VDI

REIHE 10
INFORMATIK/
KOMMUNIKATION

Fortschritt-Berichte VDI

M. Sc. Torben Miny,
Aachen

NR. 876

Konzept für die semantische Interoperabilität zwischen Informationsmodellen

BAND
1|1

VOLUME
1|1

ACPLT AACHENER PROZESSLEITTECHNIK

Lehrstuhl für
Prozessleittechnik
der RWTH Aachen

Konzept für die semantische Interoperabilität zwischen Informationsmodellen

Von der Fakultät für Georessourcen und Materialtechnik der Rheinisch-Westfälischen Technischen Hochschule Aachen

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

genehmigte Dissertation

vorgelegt von

Torben Miny, M. Sc.

Berichter: Herr Univ.-Prof. Dr.-Ing. Ulrich Epple
Herr Univ.-Prof. Dr.-Ing. Christian Diedrich
Herr Univ.-Prof. Dr.-Ing. Tobias Kleinert

Tag der mündlichen Prüfung: 21.01.2022

VDI

REIHE 10
INFORMATIK/
KOMMUNIKATION

Fortschritt-Berichte VDI

M. Sc. Torben Miny,
Aachen

NR. 876

Konzept für die semantische Interoperabilität zwischen Informationsmodellen

BAND
1|1

VOLUME
1|1

ACPLT AACHENER PROZESSLEITTECHNIK

Lehrstuhl für
Prozessleittechnik
der RWTH Aachen

VDI verlag

Miny, Torben

Konzept für die semantische Interoperabilität zwischen Informationsmodellen

Fortschritt-Berichte VDI, Reihe 10, Nr. 876. Düsseldorf: VDI Verlag 2022.

196 Seiten, 46 Bilder, 6 Tabellen.

ISBN 978-3-18-387610-5, ISSN 0178-9627,

71,00 EUR/VDI-Mitgliederpreis: 63,90

Für die Dokumentation: Semantische Interoperabilität – Modelltransformation – Object Constraint Language – Verwaltungsschale – Asset Informationsmodelle

Keywords: Semantic Interoperability – Model Transformation – Object Constraint Language – Asset Administration Shell – Asset Information Models

Die vorliegende Arbeit wendet sich an Ingenieur*innen und Wissenschaftler*innen im Umfeld von Industrie 4.0. Sie befasst sich mit der semantischen Interoperabilität zwischen digitalen Asset-Repräsentationen. Hierbei liegt der Fokus auf dem Austausch von Asset-Informationen mit Hilfe von Informationsmodellen. Derzeit werden eine Vielzahl von Informationsmodelle von verschiedenen Organisationen entwickelt. Diese enthalten vielfach semantisch identische Informationen, modellieren diese ggf. aber jeweils unterschiedlich. Kern der Arbeit ist eine neue Modelltransformationssprache zur Erstellung von Transformationsdefinitionen zwischen Informationsmodellen, um (semi-)automatisch Informationsmodelle aus anderen zu erzeugen. Die Sprache basiert auf der Object Constraint Language, ist allgemein und vollständig spezifiziert und kann in bestehende Automatisierungssystemen verwendet werden.

Bibliographische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Daten sind im Internet unter www.dnb.de abrufbar.

Bibliographic information published by the Deutsche Bibliothek (German National Library)
The Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliographie (German National Bibliography); detailed bibliographic data is available via Internet at www.dnb.de.

D82 (Diss. RWTH Aachen University, 2022)

ISBN 978-3-18-387610-5, ISSN 0178-9627

Vorwort

Die vorliegende Arbeit entstand während meiner Zeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Prozessleittechnik an der RWTH Aachen University. An dieser Stelle möchte ich mich bei allen bedanken, die mich in dieser Zeit fachlich und persönlich unterstützt haben.

Mein besonderer Dank gebührt Herrn Professor Dr.-Ing. Ulrich Epple. Seine Unterstützung für mein Promotionsvorhaben, die angenehme Arbeitsatmosphäre am Lehrstuhl und die vielen fachlichen Diskussionen und persönlichen Gespräche haben meine Arbeitsweise maßgeblich geprägt. Durch die vielfältigen Aufgaben am Lehrstuhl und die Teilnahme in diversen Gremien konnte ich mich frei entfalten und einen guten Überblick über die verschiedenen Facetten der Automatisierungstechnik und deren Schnittstelle zur Informatik erhalten.

Bei Herrn Professor Dr.-Ing. Christian Diedrich, Inhaber der Professur für Integrierte Automation an der Otto-von-Guericke-Universität Magdeburg, bedanke ich mich für die Übernahme der Rolle des Zweitgutachters. Die vielen fachlichen Diskussionen halfen mir in der Durchführung dieses Promotionsvorhabens sehr.

Zusätzlich danke ich Herrn Professor Dr.-Ing. Tobias Kleinert, Lehrstuhl-Nachfolger von Herrn Epple, für die Unterstützung in den letzten knapp 2 Jahren meiner Promotion. Herrn Professor Dr.-Ing. Herbert Pfeifer, Leiter des Instituts für Industrieofenbau und Wärmetechnik, danke ich für die Übernahme des Prüfungsvorsitz.

Ein besonderer Dank gilt meinen Kollegen am Lehrstuhl für die intensiven und teils kontroversen Diskussionen sowie den studentischen Hilfskräften und Studierenden, die bei mir eine Abschlussarbeit geschrieben haben. Besonders bedanken möchte ich mich (in alphabetischer Reihenfolge) bei Julian Grothoff, Leon Möller und Michael Thies. Ein herzlicher Dank gilt an Frau Margarete Milescu, die mich bei den diversen organisatorischen Tätigkeiten stets unterstützt hat.

Ein weiterer Dank gilt an die Mitglieder der verschiedenen Arbeitskreise, in denen ich mitarbeiten durfte (Plattform Industrie 4.0, DIN, DKE, VDI/VDE-Gesellschaft für Mess- und Automatisierungstechnik, OPC Foundation). Die vielen Gespräche haben mir einen wertvollen Einblick in das Thema Industrie 4.0 und deren praktische Anwendung gegeben.

Meiner Ehefrau Luisa Miny möchte ich für die Unterstützung, Geduld und Motivation in den vergangenen Jahren und insbesondere in der intensiven Phase bedanken. Abschließend danke ich meinen Eltern Gabi Rohde-Deppe und Torsten Deppe, die mich immer mit Tat und Rat unterstützt und mich auf diesen Weg geführt haben.

Aachen, im Januar 2022

Inhaltsverzeichnis

Abkürzungen

ADT Abstrakter Datentyp

API Application Programming Interface

ATL Atlas Transformation Language

DF Framework Digital Factory Framework

DSL Domain Specific Language

DSTL Domain Specific Transformation Language

EBNF Erweiterte Backus-Naur-Form

ETL Epsilon Transformation Language

ETSI European Telecommunications Standards Institute

GPL General Purpose Language

GPTL General Purpose Transformation Language

GUID Globally Unique Identifier

IEC International Electrotechnical Commission

IEC61360-CDD IEC 16360 - Common Data Dictionary

IEEE Institute of Electrical and Electronics Engineers

IoT Internet of Things

IRI Internationalized Resource Identifier

ISO International Organization for Standardization

IT Informationstechnologie

MOLA Model Transformation Language

OCL Object Constraint Language

OMG Object Management Group

OPC UA OPC Unified Architecture

OWL Web Ontology Language

QVT Query View Transformation

RAMI4.0 Referenzarchitekturmodell Industrie 4.0

RDF Resource Description Framework

RDFS Resource Description Framework Schema

SDK Software Development Kit

SQL Structured Query Language

TGG Triple Graph Grammatik

UML Unified Modeling Language

URI Uniform Resource Identifier

UUID Universally Unique Identifier

VIATRA Visual Automated Model Transformations

W3C World Wide Web Consortium

WoT Web of Things

ZVEI Zentralverband Elektrotechnik- und Elektronikindustrie

Kurzfassung

Im Rahmen des Zukunftsprojekts „Industrie 4.0“ der Hightech-Strategie der Bundesregierung wird das Konzept der Verwaltungsschale entwickelt. Das Ergebnis ist eine einheitliche Schnittstelle und ein Metamodell für den Zugriff auf die Informationen eines Assets. Diese Informationen werden in Informationsmodellen zusammengefasst, die jeweils einen Aspekt eines Assets darstellen und für einen konkreten Anwendungsfall verwendet werden. Durch die steigende Anzahl an kommunizierenden Geräten im industriellen Kontext, die vermehrte Nutzungen von Informationen für Mehrwertdienste (value-added services) und die Integration zu komplexen, intelligenten Maschinen und Anlagen rücken Konzepte für die Interoperabilität in den Fokus. Die semantische Interoperabilität ist ein wesentliches Ziel beim Austausch von Asset-Informationen.

Da verschiedene Stakeholder unterschiedliche Informationsmodelle benötigen, wird es eine Vielzahl dieser Informationsmodelle geben. Diese Informationsmodelle können semantisch die gleichen Informationen enthalten, jedoch anders modelliert oder zusammengestellt sein. Zusätzlich wird es verschiedene Versionen dieser Informationsmodelle geben. Dies führt zu einem Problem bei der semantischen Interoperabilität und ist durch manuelles Transformieren der Daten wegen der Vielzahl an Informationsmodellen und Assets, die digital verwaltet werden, nur noch schwer zu bewerkstelligen.

Aufgrund dessen wird in dieser Arbeit ein Konzept für die semantische Interoperabilität zwischen Informationsmodellen vorgestellt. Basierend auf einer Analyse existierender Methoden und Ansätze zur Erreichung der semantischen Interoperabilität wird das Konzept der Modelltransformation zur Lösung des Problems verwendet. Für den Asset-bezogenen Informationsaustausch werden aktuelle standardisierte Modelle miteinander verglichen und das Konzept der Verwaltungsschale als Anwendungsbeispiel herangezogen. Anhand dieses Anwendungsbeispiels wird der Unterschied zwischen syntaktischen und semantischen Transformationen vorgestellt sowie eine Klassifikation der Transformation mit Hilfe zuvor definierter Merkmale durchgeführt. Auf dieser Basis werden Anforderungen an eine Transformationssprache ermittelt und existierende Sprachen hinsichtlich ihrer Verwendbarkeit evaluiert.

Das Ergebnis der Anforderungsanalyse ist, dass bisher keine Sprache existiert, die alle Anforderungen erfüllt. Daher wird eine neue Modelltransformationssprache hergeleitet. Diese ist generisch beschrieben und wird für das Konzept der Verwaltungsschale konkretisiert. Es werden sowohl die abstrakte als auch die konkrete Syntax sowie die benötigten Syntaxregeln vorgestellt. Eine prototypische Realisierung eines Transformationssystems zeigt die Anwendung der Sprache und ermöglicht die Durchführung von Modelltransformationen zwischen beliebigen Informationsmodellen. Abschließend wird die Sprache anhand von drei ausgewählten Anwendungsfällen evaluiert.

Abstract

As part of the future project Industry 4.0 of the High-Tech Strategy of the German Federal Government, the concept of the asset administration shell is being developed. The result is a uniform interface and a metamodel for accessing the information of an asset. This information is summarized in information models, each of which represents an aspect of an asset and is used for a specific use case. Due to the increasing number of communicating devices in the industrial context, the increased use of information for value-added services and the integration to complex, intelligent machines and plants, concepts for interoperability are coming into focus. Semantic interoperability is a key goal in the exchange of asset information.

Since different stakeholders require different information models, a variety of these information models will exist. These information models may semantically contain the same information, but may be modeled or compiled differently. Additionally, there will be different versions of these information models. This leads to a semantic interoperability problem and is difficult to manage by manually transforming the data because of the large number of information models and assets that are digitally managed.

For this reason a concept for semantic interoperability between information models is presented in this thesis. Based on an analysis of existing methods and approaches to achieve semantic interoperability, the concept of model transformation is used to solve the problem. For asset-related information exchange, current standardized models are compared and the concept of the asset administration shell is used as an application example. Based on this application example, the difference between syntactic and semantic transformations is introduced and a classification of the transformation is performed using previously defined features. On this basis, requirements for a transformation language are determined and existing languages are evaluated with respect to their usability.

The result of the requirements analysis is that so far no language exists that fulfills all requirements. Therefore a new model transformation language is derived. This is described generically and is concretized for the concept of the asset administration shell. Both the abstract and the concrete syntax as well as the required syntax rules are presented. A prototypical realization of a transformation system shows the application of the language and enables the execution of model transformations between arbitrary information models. Finally, an evaluation of the language is presented based on three selected use cases.

1 Einleitung

Die Industrie unterliegt einem ständigen Wandel, welcher derzeit durch die Digitalisierung als einem der größten Treiber geprägt ist. Es gab im Laufe der Zeit neue Innovationen oder Technologien, die die Art und Weise, wie Produkte hergestellt werden, verändert haben. Erfolgt ein sprunghafter Wandel statt, wird dies industrielle Revolution genannt und ein neues Industriezeitalter wird begonnen. Bis heute fanden insgesamt drei industrielle Revolutionen statt. Die erste industrielle Revolution wurde durch die Einführung der Dampfmaschine ausgelöst, die die Nutzung von Wasser- und Dampfkraft für mechanische Produktionsanlagen ermöglichte. Durch die Nutzung von elektrischer Energie konnte die Massenfertigung und der Einsatz des Fließbands realisiert werden. Dies wird aus heutiger Sicht als zweite industrielle Revolution betrachtet. Die dritte und bisher letzte industrielle Revolution wurde mit dem Beginn der Nutzung von elektronischen Komponenten in der Automatisierungstechnik, die später auch programmierbar wurden, eingeläutet. Hierdurch konnten einzelne Arbeitsschritte, die bisher von einem Menschen erledigt wurden, durch eine Maschine übernommen werden.

Um die deutsche Industrie auf dem Weg zu einer vierten industriellen Revolution zu unterstützen, wurde im Rahmen der Hightech-Strategie der Bundesregierung das Zukunftsprojekt „Industrie 4.0“ initiiert. Durch die Informationstechnologie (IT) getrieben sollen bei der vierten industriellen Revolution die reale und die virtuelle Welt zusammenwachsen. „Die industrielle Produktion [soll] mit Hilfe modernster Informations- und Kommunikationstechnik auf intelligente Weise“ verzahnt und die Vereinigung von „Großproduktion mit individuellen Kundenwünschen, kostengünstig und in hoher Qualität“ erreicht werden [1].

Ein Ergebnis des Zukunftsprojekts ist das Konzept des Asset-bezogenen Informationszugriffs. Als Asset wird eine „Entität, die einen wahrgenommenen oder tatsächlichen Wert für eine Organisation hat und der Organisation gehört oder von ihr verwaltet wird“ [2] aufgefasst. Diese Entitäten können sowohl physische als auch virtuelle Betrachtungsgegenstände sein, wie z. B. Sensoren, Aktoren, Pläne oder Handbücher. Bisher wurden Informationen in dem IT-System abgelegt, in dem diese angefallen sind, und es findet in der Regel keine Übertragung zwischen verschiedenen IT-Systemen statt. Dies betraf vor allem den Informationsübergang zwischen zwei Gewerken, wodurch die Industrie „an fehlender Durchgängigkeit ihrer Anlagen- und Prozessdaten“ [3] leidet. Mit Hilfe des Asset-bezogenen Informationszugriffs soll zukünftig die Möglichkeit bestehen, Informationen über ein Asset über einen definierten Zugriffspunkt aus den verschiedensten Gewerken abzurufen. Ein Asset erhält somit eine digitale Repräsentanz, über die die Informationen des gesamten Lebenszyklus eines Assets abrufbar sind.

Da verschiedene Stakeholder auf die Asset-Informationen zugreifen sollen und jeweils unterschiedliche Sichtweisen auf die Modellierung und Verknüpfung dieser Informationen ha-

ben, besteht die Notwendigkeit einzelne Informationsmodelle für die jeweils benötigten Teilaspekte zu definieren. Folglich muss die Möglichkeit bestehen, ein vollständiges sowie voll umfassendes digitales Modell eines Assets durch einzelne Stakeholder-orientierte Teilmodelle bereitzustellen. Das im Rahmen von Industrie 4.0 entwickelte Konzept der Verwaltungsschale unterstützt diese Art der Modellierung [4–6]. Ähnliche Ansätze finden sich auch in aktuellen Kommunikationsprotokollen, z. B. die Companion Specifications in OPC Unified Architecture (OPC UA) [7].

1.1 Motivation und Zielsetzung

Für eine bessere Interoperabilität ist die Entwicklung eines einheitlichen Metamodells die Grundvoraussetzung. Derzeitig spezifizierte bzw. bereits existierende Modelle sind z. B. die Verwaltungsschale [4–6], das Digital Factory Framework [8–10] oder die Thing Description des World Wide Web Consortium (W3C) [11, 12]. Im nächsten Schritt sollen möglichst viele Informationsmodelle, basierend auf dem spezifizierten Datenmodell, standardisiert werden. Da jedoch die Stakeholder sowohl aus den unterschiedlichsten Gewerken Informationen als auch für die konkreten Anwendungsfälle jeweils andere Kombination bzw. Darstellung der Informationen benötigen, werden zwangsläufig sehr viele verschiedene Informationsmodelle entstehen. Dabei können sich Informationsmodelle überschneiden und ggf. mehrere Informationsmodelle für den gleichen Use Case von unterschiedlichen Organisationen existieren. Dabei können die gleichen Informationen in den jeweiligen Informationsmodellen jedoch auch unterschiedlich modelliert sein. Beispielsweise kann im Metamodell die Möglichkeit bestehen, einen Wertebereich entweder als eigenständiges Objekt oder durch zwei Objekte, die jeweils die Ober- und Untergrenze darstellen, zu modellieren. Beide Darstellungen enthalten dabei semantisch die gleichen Informationen.

Ein Vergleich bestehender Informationsmodelle aus dem Bereich der Verwaltungsschale zeigt diese Problematik in der Praxis auf. Im November 2020 wurden die ersten zwei Informationsmodelle (Teilmodell Templates) [13, 14] veröffentlicht. Dabei wurde z. B. das Merkmal „ManufacturerName“ auf unterschiedliche Weise modelliert. In [13] wird ein Property-Element mit dem Datentyp „string“ und einer Referenz auf eine semantische Beschreibung aus dem Vokabular unter der URI „admin-shell.io“ verwendet, während in [14] ein MultiLanguageProperty-Element mit dem Datentyp „langString“ sowie einer Referenz auf das Vokabular von ECLASS[1] verwendet wird. Beide Elemente stellen semantisch jedoch die gleichen Informationen dar. Da bereits bei den ersten zwei veröffentlichten standardisierten Informationsmodellen das Problem der unterschiedlichen Modellierung äquivalenter Informationen auftritt, ist davon auszugehen, dass bei weiteren Informationsmodellen - und besonders bei firmenspezifischen Informationsmodellen - dieses Problem zunimmt.

Dem Problem kann mit einer einheitlichen und gleichen Modellierung entgegengewirkt werden, indem z. B. gleiche Merkmale in übergeordnete Informationsmodelle ausgelagert werden. Aktuell erfolgt dies in der Regel durch Menschen in Harmonisierungsgruppen, z. B. bei OPC UA für die Companion Specifications [7]. Für eine geringe Anzahl von Informationsmodellen ist dies ein guter Ansatz. Steigt jedoch die Anzahl ähnlicher Informations-

[1] www.eclass.eu

modelle, nimmt die Komplexität der Harmonisierung zu und ist wahrscheinlich nicht mehr manuell realisierbar.

Zusätzlich werden im Zuge von Anforderungsänderungen neue Versionen der einzelnen Informationsmodelle entstehen. Dabei können einzelne Elemente geändert, gelöscht oder hinzugefügt werden. Spätestens hier kann der Ansatz der Harmonisierung nicht mehr genutzt werden, sondern es bedarf einer Möglichkeit Informationen automatisch aus älteren Versionen in neuere Versionen, und umgekehrt, zu überführen.

Die Integration von Komponenten in Module oder ganze Anlagen ist ein weiterer Anwendungsfall, in dem automatisierte Konzepte für die Zusammenführung von Informationen benötigt werden. Zukünftig werden sowohl für die einzelnen Komponenten als auch für das Modul bzw. die Anlage eigene Informationsmodelle vorliegen, wobei die Informationen zum Teil semantisch identisch sein werden oder durch eine Aggregation erzeugt werden können [15]. Dem Autor ist kein Ansatz bekannt, der für einen Integrator oder Betreiber diese Zusammenführung der Informationen automatisiert und generisch für alle Informationsmodelle ermöglicht.

Somit werden Konzepte benötigt, die diese Probleme softwaretechnisch lösen. Es existieren bereits erste Ansätze aus dem Bereich des Semantic Webs oder des maschinellen Lernens, die die semantische Gleichheit von zwei oder mehr Objekten über Ontologien oder sprachliche Vergleiche herausfinden sollen [16–18]. Zusätzlich besteht die Möglichkeit, Berechnungsvorschriften zwischen Objekten zu modellieren, um z. B. physikalische Zusammenhänge abzubilden. Jedoch ist zum Zeitpunkt dieser Arbeit kein Konzept bekannt, wie Informationsmodelle möglichst schnell und effizient auf Basis dieser Informationen oder durch Wissen von Fachexperten erstellt werden können.

Im Rahmen dieser Arbeit wird daher ein Konzept vorgestellt, das Fachexperten die Möglichkeit bietet, semantische Regeln für die Erstellung von Instanzen dieser Informationsmodelle auf Basis von Instanzen bestehender Informationsmodelle zu definieren. Diese Regeln können ausgeführt werden sofern ein neues bzw. angefragtes Informationsmodell benötigt wird. Hierfür wird das Konzept der Modelltransformation genutzt. Vereinfacht ermöglicht die Modelltransformation folgendes: eine oder mehrere bestehende Instanzen von vorgegebenen Informationsmodellen werden eingelesen. Auf diesen werden Regeln ausgeführt und als Ergebnis wird eine neue Instanz eines Informationsmodells erstellt. Eine Regel wird zwischen den Informationsmodellen definiert und könnte wie folgt lauten: „Erstelle ein neues Datenelement *B* im neuen Informationsmodell *IM_2* mit den Informationen aus dem Datenelement *A* des Informationsmodells *IM_1*“. Damit solche Regeln definiert werden können, müssen standardisierte Informationsmodelle[2] vorliegen, die für verschiedene Asset-Repräsentationen instanziiert werden. In der objektorientierten Modellierung können diese Informationsmodelle als Typen bzw. Templates verstanden werden. Für eine bessere Unterscheidung werden die konkreten Informationsmodelle als Informationsmodell-Instanzen und die zugehörigen standardisierten Informationsmodelle als Informationsmodell-Templates bezeichnet. Die Regeln, um aus einer oder mehreren Informationsmodell-Instanz(en) eine andere Informationsmodell-Instanz zu erzeugen, werden in sogenannten Transformations-Definitionen zusammengefasst.

[2]Firmenspezifische Informationsmodelle zählen auch als standardisierte Informationsmodelle.

Die **zentrale Fragestellung dieser Arbeit** ist, wie aus bestehenden Informationen in bereits existierenden Informationsmodell-Instanzen, die standardisierten Informationsmodell-Templates folgen, neue benötigte bzw. angefragte Informationsmodell-Instanzen teilweise bis vollständig generiert werden können.

In Abbildung 1.1 ist ein möglicher Workflow zur Erstellung von Transformations-Definitionen dargestellt. Ein Fachexperte wählt zunächst die Informationsmodell-Templates aus, zwischen denen Regeln definiert werden sollen. Danach werden diese hinsichtlich ihrer enthaltenen Informationen analysiert und die Regeln zur Erstellung einer neuen Informationsmodell-Instanz festgelegt. Hierfür können auch externe Systeme genutzt werden, die bei der Erstellung der Regeln unterstützen, z. B. mit Hilfe von Methoden der künstlichen Intelligenz oder Reasoning-Methoden für Ontologien. Diese werden anschließend in einer Transformations-Definition beschrieben. Um diese von anderen Personen oder Applikationen zu verwenden, erfolgt eine Ablage in eine Datenbank. Für eine bessere Suche können zusätzlich noch Meta-Informationen mit abgespeichert werden.

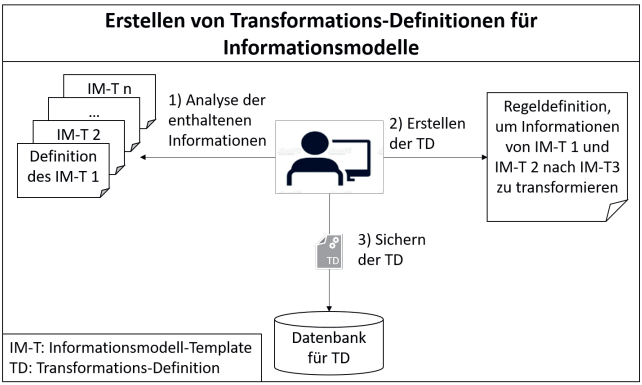

Abbildung 1.1: Erstellung von Transformations-Definitionen für Informationsmodelle (nach [19])

Abbildung 1.2 zeigt die Anwendung von Transformations-Definitionen. Als erstes werden vorhandene Informationsmodell-Instanzen des Assets (grün und rot dargestellt) geladen. Ein Nutzer oder die Applikation gibt anschließend ein Ziel-Informationsmodell-Template vor (in der Abbildung orange schraffiert dargestellt). In der Datenbank wird nach passenden Transformations-Definitionen gesucht, um diese dem Anwender oder der Applikation vorzuschlagen. Nachfolgend wird eine dieser Transformations-Definitionen ausgewählt und die (eigentliche) Modelltransformation gestartet, welche zur Erstellung der gewünschten Informationsmodell-Instanz (orange dargestellt) führt.

Um diese Workflows zu erreichen, wird eine Sprache für die Definition der semantischen Regeln innerhalb der Transformations-Definition benötigt. Im Rahmen dieser Arbeit wird die Vorgehensweise für die Definition einer neuen Sprache sowie eine konkrete Sprache für diesen Anwendungsfall vorgestellt, mit Hilfe derer diese Regeln formuliert werden. Einige der zu erfüllenden Anforderungen sind:

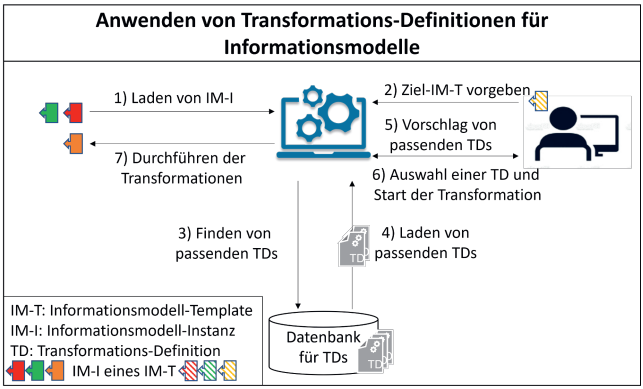

Abbildung 1.2: Anwendung von Transformations-Definitionen für Informationsmodelle (nach [19])

- Die Syntax der Sprache soll einfach zu verstehen sein.
- Die Anzahl der Sprachelemente soll so gering wie möglich, aber so komplex wie nötig sein.
- Die Sprache soll auf den aktuellen Konzepten bzw. Modellen der Wissenschaft basieren.
- Die Sprache soll die Sprachelemente des Metamodells der Informationsmodelle nutzen.

Im Zuge der Standardisierung wurden verschiedene Modellierungen für Asset-Informationen und deren Austausch entwickelt[3]. Das Konzept der Verwaltungsschale bzw. Asset Administration Shell wird derzeit in Deutschland als einer der erfolgversprechendsten Ansätze gehandelt. Aufgrund dessen wird das Konzept für die Elemente des Verwaltungsschalen-Metamodells konkretisiert und anhand diesem evaluiert.

1.2 Gliederung

In dieser Arbeit wird ein Konzept zur Lösung fehlender semantischer Interoperabilität, basierend auf unterschiedlicher Modellierung von Informationen, beschrieben. Zur Erläuterung der Grundlagen werden zunächst die Themen „Sprache“, „Modell“, „Metamodell“ und „Modellsprachen“ vorgestellt (Kapitel 2). Zusätzlich wird die Beziehung zwischen Typen und Instanzen sowie der aktuelle Stand der Identifikation von Objekten aufgezeigt.

[3]Ein Vergleich aktueller Standardisierungsvorhaben ist in Kapitel 6 gegeben.

Nachfolgend wird die Modellierungssprache Object Constraint Language (OCL) erläutert, die als Basis des vorgestellten Konzepts dient (Kapitel 3). Für die vorliegende Arbeit erfolgt zunächst eine Kurzeinführung in die relevanten Sprachelemente, bevor die abstrakte Syntax von BasicOCL erläutert wird. Aufbauend darauf werden die Stufen der Interoperabilität vorgestellt und aktuelle Ansätze diskutiert (Kapitel 4), wobei der Fokus auf der semantischen Interoperabilität liegt.

Einer dieser Ansätze ist die Nutzung der Modelltransformation, die für das Lösungskonzept dieser Arbeit die Grundlagen schafft (Kapitel 5). Zunächst wird das Grundkonzept vorgestellt. Anschließend werden Merkmale definiert, nach denen sich Modelltransformationen klassifizieren lassen. Im Anschluss erfolgen verschiedene Umsetzungsansätze, bevor Transformationssprachen und zugehörige -systeme sowie ein Vorgehen zur Entwicklung bzw. Auswahl einer solchen Sprache beschrieben werden.

Um das Konzept anwendungsnäher zu beschreiben, werden aktuelle Modellierungsansätze für Asset Information vorgestellt (Kapitel 6). Eine Eingrenzung erfolgt durch maßgeblich diskutierte Ansätze der Standardisierung und Forschung. Aus der IEC wird das Digital Factory Framework, von der Plattform Industrie 4.0 die Verwaltungsssschale und vom W3C die Thing Description beschrieben. Eine Gegenüberstellung und eine Bewertung schließen dieses Kapitel ab.

Als Resultat wird der Ansatz der Verwaltungsschale in dieser Arbeit weiterverfolgt (Kapitel 7). Der Fokus liegt auf den verschiedenen Erscheinungsarten und der Nutzung von Teilmodellen für die semantische Interoperabilität. Zum Abschluss werden offene Fragestellungen und mögliche Lösungsoptionen vorgestellt. Als ausgewählte Lösung wird die Modelltransformation verwendet.

Im nachfolgenden Kapitel 8 folgt die Anforderungsanalyse. Es werden die verschiedenen Arten der Transformation, die für die semantische Interoperabilität notwendig sind, erläutert. Darauf aufbauend wird eine Klassifikation mit den Merkmalen aus Kapitel 5 vorgenommen und die Anforderungen an eine Transformationssprache beschrieben. Im Anschluss erfolgt eine Evaluation bestehender Transformationssprachen hinsichtlich dieser Anforderungen.

Basierend auf der Anforderungsanalyse wurde ein Metamodell für eine neue Transformationssprache entwickelt (Kapitel 9). Für die einfache Nutzung im Bereich von Verwaltungsschalen wurde eine Abbildung dieser Sprache auf das Konzept der Verwaltungsschale inkl. der Nutzung der Sprache zur vereinfachten Erzeugung von Regeln definiert (Kapitel 10).

Eine Beschreibung der softwaretechnischen Umsetzung des Transformationssystems zeigt die einfache Realisierung der Sprache (Kapitel 11). Anschließend werden anhand von drei verschiedenen Anwendungsfällen das Konzept und die Umsetzung evaluiert und Empfehlungen für die Nutzung gegeben (Kapitel 12). Abschließend erfolgt eine Zusammenfassung sowie ein Aufzeigen der nächsten Schritte (Kapitel 13).

1.3 Eigene Vorveröffentlichungen

Während der Forschung zu dieser Arbeit wurden verschiedene Ergebnisse bereits publiziert. Einige der Abschnitte aus diesen Veröffentlichungen werden in dieser Arbeit eins zu eins

wiederverwendet und andere dienen als Grundlage. Aus diesem Grund folgt eine kurze Vorstellung der Veröffentlichungen:

DIN SPEC 92000 als Enabler für Plug and Produce

In dem ATP-Beitrag [20] werden die Neuerungen der DIN SPEC 92000 vorgestellt und deren Nutzung für das Konzept des *Plug and Produce*. Im Beitrag werden zunächst die Anforderungen des Konzepts Plug and Produce beschrieben. Danach folgt ein Überblick über die aktuelle Normungslandschaft für die Eigenschaftsmodellierung, welcher in Kapitel 6 übernommen wurde. Im Beitrag folgt danach eine Beschreibung der Inhalte der DIN SPEC 92000. In Form von Use Cases wird die Anwendung der neuen Konzepte für die Nutzung des Konzepts Plug and Produce aufgezeigt. Eine technische Realisierung zeigt abschließend die Umsetzbarkeit.

Konzept für die automatisierte Erstellung von Verwaltungsschalen-Teilmodellen mit Hilfe domänenspezifischer Transformationssprachelemente

Der Automation-Beitrag [19] zeigt, wie das Konzept der Modelltransformation für die automatische Erstellung von Verwaltungsschalen-Teilmodellen genutzt werden kann. Zunächst wird allgemein das Konzept der Modelltransformation auf die Begriffswelt der Verwaltungsschale konkretisiert. Danach wird der Unterschied zwischen syntaktischer und semantischer Transformation beschrieben. Dieser Abschnitt ist in Kapitel 8.1 übernommen worden. Anschließend folgt im Automation-Beitrag eine Kurzvorstellung über domänenspezifische Transformationssprachelemente. Der Beitrag schließt mit zwei Workflows ab: Einer für die Erstellung von Transformationsdefinitionen und einer für die Anwendung dieser im Kontext von Verwaltungsschalen-Teilmodellen. Eine abstraktere Beschreibung dieser Workflows für allgemeine Informationsmodelle ist bereits in Abschnitt 1.1 gegeben.

Model Transformation for Asset Administration Shells

Im IECON-Beitrag [21] wird beschrieben, wie eine Transformationssprache entwickelt wird und wie diese für das Konzept der Verwaltungsschale aussehen kann. Es wird zunächst ein Leitfaden zur Erstellung bzw. Auswahl einer Transformationssprache beschrieben. Dieser besteht aus drei Schritten: Klassifikation der Transformation, Anforderungen an die Transformationssprache und Design einer Transformationssprache. Inhalte dieser Vorgehensweise werden in Abschnitt 5.4 wiederverwendet. Anschließend wird im IECON-Beitrag diese Vorgehensweise für die Entwicklung einer Transformationssprache für Verwaltungsschalen angewendet. Diese Vorarbeit wird in Abschnitt 8.3 weiter detailliert. Abschließend wird im IECON-Beitrag ein erster Entwurf der Transformationssprache AASMTL vorgestellt. Dieser Entwurf diente als Basis für die Kapitel 9 und 10.

2 Modellierung

In diesem Kapitel werden die Grundlagen der Modellierung erklärt. Zunächst werden die Grundlagen einer Sprache und der zugehörigen Metasprache vorgestellt. Danach wird der Begriff des Modells sowie die Beziehung zur Sprache betrachtet. Es werden Analogien aufgezeigt und der Metamodellbegriff definiert. Danach wird der Begriff der Modellsprache detaillierter beleuchtet und die verschiedenen Eigenschaften, nach denen sich Modellsprachen klassifizieren lassen, beschrieben. Den Abschluss bildet eine Unterscheidung von Typen und Instanzen sowie die Identifikation von Objekten.

2.1 Sprache und Metasprache

Um Aussagen über Eigenschaften und Relationen zwischen Betrachtungsgegenständen treffen zu können, werden Begriffe und Sätze benötigt. Damit diese von verschiedenen Benutzern einheitlich verstanden werden, wird eine Sprache benötigt. Eine Sprache ist ein System von Zeichen und definiert Regeln zur Verwendung dieser [22, 23]. Wird die Sprache selbst zum Betrachtungsgegenstand, wird von Sprache der Sprache gesprochen. Um diese zu unterscheiden, werden die Begriffe der Objekt- und Metasprache eingeführt [24]. Als Objektsprache wird die zu betrachtende Sprache definiert. Die Sprache, in der die Untersuchung erfolgt, wird als Metasprache bezeichnet [25]. Dieses Konstrukt kann rekursiv angewendet werden, sodass die Metasprache wiederum zur Objektsprache wird und eine eigene Metasprache besitzt. Dieses Vorgehen kann in einem Ebenen-Diagramm dargestellt werden. Dieses beginnt auf der untersten Ebene mit der Objektsprache, darauf folgt die Metasprache, darauf die Metametasprache (s. Abbildung 2.1).

2.2 Modell und Metamodell

Für den Begriff *Modell* existieren diverse Definitionen: In [27] wird Modell als „die Abbildung von Objekten, Eigenschaften oder Relationen eines bestimmten Bereichs der objektiven Realität oder einer Wissenschaft auf einfachere, übersichtlichere materielle Strukturen desselben oder eines anderen Bereichs“ definiert. Das I40-Glossar beschreibt ein Modell als eine „schlüssige, ausreichend detaillierte Abstraktion von Aspekten in einem Anwendungsbereich“ [2]. Polke stellt in [28] heraus, dass der Modellbegriff sehr umfassend ist, aber charakteristische Merkmale vorliegen. Als charakteristisch definiert er, „dass Modelle immer vereinfachende Bilder des Eigenschafts- und Funktionsprofils des zugrundeliegenden realen Objekts sind und zwar unter einem bestimmten Blickwinkel“. In [29] ist zudem eine Übersicht über verschieden Definitionen aus dem Bereich der Softwaretechnik gegeben.

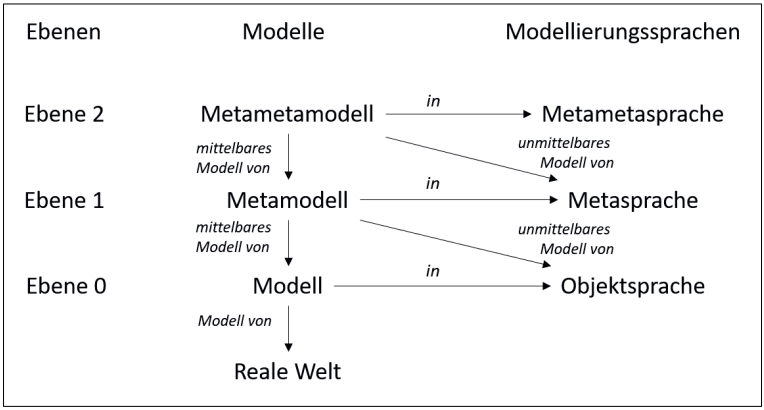

Abbildung 2.1: Sprachbasierter Metamodellbegriff nach [26]

Bereits Stachowiak hat 1973 in seinem Werk „Allgemeine Modelltheorie“ [30] festgehalten, dass eine allumfassende Definition nicht möglich ist. Er definiert aber drei Hauptmerkmale des allgemeinen Modellbegriffs:

- Abbildungsmerkmal: Ein Modell ist immer ein Abbild eines Originals, welches selbst wieder ein Modell sein kann.
- Verkürzungsmerkmal: Ein Modell erfasst nur die für den jeweiligen Modellerschaffer relevanten Attribute des Originals.
- Pragmatisches Merkmal: Ein Modell erfüllt einen Zweck und ist nur für diesen gültig (z. B. nur innerhalb bestimmter Zeitintervalle).

Da keine allgemeine Definition für ein Modell existiert, wird für diese Arbeit folgende neue Definition eingeführt:

Definition 2.1 (Modell) *Ein Modell ist immer eine Abbildung eines Originals für einen bestimmten Zweck, in dem dieses gültig ist, und erfasst nur die für den Modellerschaffer relevanten Attribute des betrachteten Originals.*

Um Modelle zu erstellen, wird eine Sprache benötigt. Da das Modell der Betrachtungsgegenstand ist, ist die Sprache eine Objektsprache. Diese wird allgemein als Modellsprache bezeichnet [31] und in Abschnitt 2.3 näher erläutert. Dabei kann die verwendete Modellsprache mit Hilfe von Beschreibungsmodellen spezifiziert werden. Diese Modelle werden als Metamodelle bezeichnet. Ein Metamodell ist demnach ein Beschreibungsmodell für ein Modell und ist nach [23] wie folgt definiert werden:

Definition 2.2 (Metamodell) *Ein Metamodell ist Modell eines Modells, wobei es sich bei dem übergeordneten Modell um ein sprachliches Beschreibungsmodell handelt, dass die Sprache, in der das untergeordnete Modell formuliert ist, abbildet.*

Wendet man das Prinzip der Ebenentheorie auf die Modellbildung an, erhält man analog dazu Metamodelle und Metametamodelle. Dieses Vorgehen der Konstruktion solcher Metamodelle wird in der Software-Entwicklung auch als Metamodellierung bezeichnet [32]. Werden die Modellsprache und die Modelle hierarchisch strukturiert und in Relation zueinander gesetzt, so ergibt sich, dass zu jedem Metamodell und Metametamodell auch eine Metasprache und eine Metametasprache existiert. Das Metamodell entspricht dabei unmittelbar der Objektsprache und durch die Vorgabe der Sprachelemente mittelbar dem Modell. Der Zusammenhang zwischen Modellen und Modellierungssprachen ist in Abbildung 2.1 dargestellt. Da verschiedene Arten von Modellierungssprachen existieren, werden deren Eigenschaften im nächsten Abschnitt genauer beschrieben.

2.3 Modellsprachen

Eine Modellsprache beschreibt die Darstellung der nutzbaren Elemente und die Beziehungen zwischen diesen in einem Modell. Dafür werden das Vokabular und die Grammatik, die die Ersteller und Nutzer bei der Modellierung eines Modells benutzen müssen, festgelegt. Dadurch wird ein einheitliches Verständnis erzeugt und Maschinen durch die in der Modellsprache definierte Semantik der Modellelemente in die Lage versetzt, die Modelle weiter zu verarbeiten [33]. Ebenso wird explizit festgelegt, welche Informationen dargestellt und welche aufgrund fehlender Konzepte in der Modellsprache nicht dargestellt werden können.

Modellsprachen können hinsichtlich ihrer Nutzer und ihrer Darstellungsform klassifiziert werden. Die Hauptnutzer können Menschen und Maschinen sein. Dies bedeutet, dass Modellsprachen entweder für Maschinen oder für eine intuitive Nutzung durch den Menschen entwickelt werden. Hinsichtlich der Darstellungsform können grafische und Zeichen-basierte Modellsprachen unterschieden werden [34].

Um eine Modellsprache zu entwickeln, muss zunächst die Syntax definiert werden. Es wird zwischen konkreter und abstrakter Syntax unterschieden [32, 35].

Die konkrete Syntax definiert die verwendbaren Symbole. Sie wird auch Notation genannt [34]. Als Beispiele für die konkrete Syntax einer textuellen Sprache können die Programmiersprachen C [36] oder Python[1] genannt werden. Dazu werden die Zeichen definiert, die nach bestimmten Mustern zu linearen Zeichenketten verknüpft werden können (s. Beispiel 2.1).

Beispiel 2.1: Verschiedene konkrete Syntaxen einer Addition

```
1 2 + 3
2 (2 + 3)
3 (+ 2 3)
4 die Summe von 2 und 3
```

Für rein grafische Sprachen werden Linien, Pfeile, Rechtecke oder andere Symbole verwendet. Diese bilden in der Regel einen Graphen. Ein klassischer Vertreter dieser Sprachkategorie ist das Petri-Netz [37, 38] (s. Abbildung 2.2). Es gibt aber auch Mischformen, die zu den

[1] https://www.python.org/

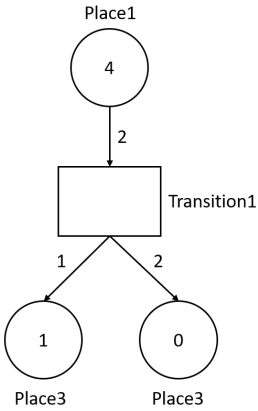

Abbildung 2.2: Konkrete Syntax eines Petri-Netzes

graphischen Symbolen textuelle Sprache verwenden, z. B. Rechtecke mit Text-Elementen, wie sie in Unified Modeling Language (UML)-Klassendiagrammen [39] verwendet werden. Die konkrete Syntax ist die Darstellung, die ein Modellierer im Modellierungswerkzeug sieht. Die abstrakte Syntax hingegen abstrahiert die konkrete Syntax auf die Modellierungskonzepte und deren Beziehungen. Sie definiert dafür abstrakt die in der konkreten Syntax nutzbaren Symbole [32]. Diese und das zugehörige Datenformat sind dem Modellierer meistens verborgen. Für textuelle Sprachen werden meistens abstrakte Grammatiken genutzt. Bei den grafischen Sprachen werden die in Abschnitt 2.2 eingeführten Metamodelle verwendet. Eine Sprache hat immer genau eine abstrakte Syntax, kann aber mehr als eine konkrete Syntax besitzen. Eine abstrakte Syntax für die konkreten Syntaxen aus dem Beispiel 2.1 ist in Beispiel 2.2 dargestellt.

Beispiel 2.2: Beispiel für abstrakte Syntax einer Addition

```
1 2 plus 3
```

Für die konkrete Syntax des Petri-Netzes aus Abbildung 2.2 könnte die abstrakte Syntax wie in Abbildung 2.3 dargestellt aussehen.

Im zweiten Schritt wird die Semantik der Sprache definiert. Diese beschreibt die Bedeutung der nutzbaren Syntaxsymbole und syntaktischer Konstrukte. Zusätzlich wird der Begriff der statischen Semantik [32, 41] definiert, welche die Wohlgeformtheitskriterien der Sprache festlegt. Diese wird in der Regel durch eine Reihe von Einschränkungen, wie z. B. des Wertebereichs oder der Beziehungen zwischen Elementen, festgelegt [32]. Betrachtet man den Zusammenhang aus Abbildung 2.1, beschreibt ein Metamodell die abstrakte Syntax sowie die statische Semantik.

Die Definition der Syntax und Semantik kann durch eine informale, semi-formale oder formale Sprache erfolgen [41]. Als formal gilt eine Sprache, die eine präzise, eindeutig festgelegte, mathematisch fundierte und somit widerspruchsfreie Syntax und Semantik besitzt.

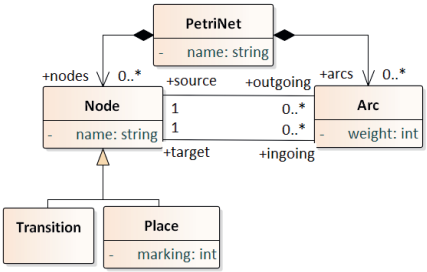

Abbildung 2.3: Abstrakte Syntax eines Petri-Netzes nach [40]

Dies wird z. B. durch kontextfreie Grammatikdefinitionen ermöglicht. Bei einer informalen Sprache ist nur die Syntax und nicht die Semantik definiert. Die Interpretation einer Sprachanwendung obliegt dem Betrachter, womit keine Prüfbarkeit möglich ist (z. B. eine natürliche Sprache). Als Mischform gilt die semi-formale Sprache, die nicht bewiesenermaßen eindeutig und widerspruchsfrei ist, aber einen nachvollziehbaren formalen Charakter hat, z. B. Metamodelle. Die Beschreibungsform kann auch zwischen der Syntax und der Semantik wechseln. Beispielsweise wird die abstrakte Syntax einer Modellsprache meistens semi-formal durch eine graphische Notation beschrieben [41]. Die zugehörige Semantik wird jedoch fast immer durch natürliche Sprache definiert, da versucht wird, die Anzahl an graphischen Symbolen möglichst gering zu halten [42].

Sprachen können in universelle (General Purpose Language (GPL)) und domänenspezifische Sprachen (Domain Specific Language (DSL)) unterschieden werden. Eine universelle Sprache definiert Sprachelemente, die nicht für eine konkrete Domäne zugeschnitten sind. Sie können somit für verschiedenste Problemstellungen genutzt werden. Klassische Vertreter sind die Programmiersprachen, wie C oder Java, Datenaustauschmodelle, wie XML oder JSON, oder Modellierungssprachen, wie UML. Dem entgegen werden domänenspezifische Sprachen für ein Anwendungsgebiet erstellt und bilden die dort benötigten Sprachelemente in der domänenspezifischen Begriffswelt ab [43, 44]. Einem Experten seines Anwendungsgebiets wird dadurch ermöglicht, sein Wissen leichter und verständlicher auszudrücken sowie zu nutzen. Klassische Vertreter sind domänenspezifische UML-Modelle oder XML-Schemata sowie die Structured Query Language (SQL) [45]. DSL können sowohl neu definiert[2] als auch auf Basis einer GPL für eine Anwendungsdomäne spezialisiert werden[3] [46]. Bei einer Neuerstellung muss die komplette Werkzeugkette (z.B. Editoren, Parser, Validatoren etc.) neu entwickelt werden. Dies ist bei der Erweiterung oder Konkretisierung einer GPL nicht notwendig, da viele Tools der GPL wiederverwendet werden können. Die in dieser Arbeit betrachteten Informationsmodelle (s. Kapitel 6) sind DSL, die auf der UML basieren.

[2]Nach [43] werden sie dann externe DSL genannt.
[3]Die GPL wird dann zur Metasprache dieser Sprache und nach [43] interne DSL genannt.

2.4 Typ und Instanz

In diesem Abschnitt werden die Begriffe *Typ* und *Instanz* für diese Arbeit sowie die Beziehungen zwischen diesen definiert. Im Bereich der Semantik und Wissenspräsentation (Informations- und Datenmodellierung) sowie der objektorientierten Modellierung werden semantische Netze aus Knoten und Kanten erstellt. Als Knoten wird ein Objekt, welches einen Begriff oder Konzept darstellt, und als Kante eine Relation zwischen Objekten verstanden. Ein klassischer Vertreter ist das Entity-Relationship-Modell. Objekte werden zusätzlich weiter in Typen und Instanzen konkretisiert. In der Literatur existieren unterschiedliche Definitionen für beide Begriffe, jedoch sind die Grundaussagen dieselben. Für diese Arbeit werden die Definitionen aus dem Industrie 4.0 Glossar [2] genutzt (wobei Objekt und Entität synonym verwendet werden[4]):

Definition 2.3 (Instanz) *Eine Instanz ist eine „konkrete Entität [bzw. Objekt], die [bzw. das] die Merkmale und deren Ausprägungen eines Typs erfüllt." [2]*

Definition 2.4 (Typ) *Ein Typ ist eine „beschreibende Entität [bzw. Objekt] gekennzeichnet durch [eine] Menge von gemeinsamen Merkmalen und deren Ausprägungen." [2]*

Eine Instanz kann dabei nicht ohne einen Typen existieren. Dies bedeutet jedoch nicht, dass der Typ immer explizit modelliert bzw. implementiert sein muss.

Um Objekte miteinander zu verbinden, werden Beziehungen zwischen diesen definiert. In Bezug auf Typen und Instanzen können diese Beziehungen in drei Arten unterschieden werden: Beziehungen zwischen Typen, Beziehungen zwischen Instanzen und Beziehungen zwischen Typen und Instanzen. An dieser Stelle werden zwei für diese Arbeit wichtige Beziehungen vorgestellt:

Is-Instance-Of-Beziehung: Die Is-Instance-Of-Beziehung ist eine Beziehung zwischen einem Typ A und einer Instanz B [48]. Die Beziehung sagt aus, dass die Instanz B ein konkretes Objekt dieses Typs A ist und alle Merkmale und deren Ausprägungen erfüllt.

Is-Subtype-Of-Beziehung: Die Is-Subtype-Of-Beziehung ist eine Beziehung zwischen einem Typ A und einem Typ B [48]. Die Beziehung sagt aus, dass der Typ A eine Spezialisierung des Typs B und somit ein Subtyp von B ist. Das bedeutet, dass der Typ A alle Merkmale und deren Ausprägungen von Typ B beibehält und zusätzlich weitere Merkmale und Ausprägungen enthalten kann.

Die Unterscheidung dieser beiden Beziehungen ist sehr wichtig, da diese auf und zwischen unterschiedlichen Ebenen in der Metamodellierung auftreten. Während die Is-Instance-Of-Beziehung immer zwischen zwei Ebenen auftritt, befindet sich die Is-Subtype-Of-Beziehung immer innerhalb einer Ebene. Ein Beispiel ist in Abbildung 2.4 dargestellt. Die Klasse *Typ* auf oberster Ebene stellt einen Typ dar. Von diesem existieren insgesamt drei Instanzen: *Fahrzeug*, *Auto* und *Motorrad*. Diese stehen wiederum in einer Beziehung zueinander: Das Auto und das Motorrad sind Spezialisierungen und somit Subtypen vom Typ Fahrzeug.

[4]Entität wird im Bereich der Datenmodellierung als Begriff benutzt (Entität und Entitätstyp) und Objekt im Bereich der objektorientierten Programmierung (Objekt und Klasse) [47].

Auf der untersten Ebene befinden sich die konkreten Instanzen dieser Typen. Es können nun folgende Aussagen getroffen werden:

- Das Objekt *Mein Auto* ist eine Instanz des Typs *Auto*. Es ist somit auch eine Instanz des Typs *Fahrzeug*. Es ist aber keine Instanz des Typs *Typ*.
- Das Objekt *Motorrad* ist eine Instanz des Typs *Typ*. Es ist aber keine Instanz von *Fahrzeug*, sondern ein Subtyp von diesem.
- Das Objekt *Fahrzeug* ist eine Instanz des Typs *Typ* und hat die zwei Subtypen *Auto* und *Motorrad* sowie die beiden Instanzen *Mein Auto* und *Mein Motorrad*.
- Das Objekt *Typ* hat drei Instanzen *Fahrzeug*, *Auto* und *Motorrad* aber keine Beziehung zu den Objekten *Mein Auto* und *Mein Motorrad*.

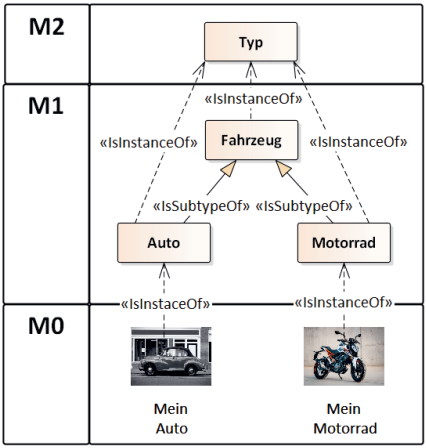

Abbildung 2.4: Is-Instance-Of- und Is-Subtype-Of-Beziehung

Weiterhin kann die Is-Instance-Of-Beziehung hinsichtlich ihrer Ausprägung unterschieden werden. Falls der Typ modelliert wird, kann dies sowohl sehr detailliert und formal als auch nur durch eine informelle Begriffsbeschreibung geschehen. Detailliert und formal wäre z. B. eine Klasse in der objektorientierten Programmierung. Die informelle Begriffsbeschreibung findet z. B. in der Klassifikation und Wissensmodellierung ihre Anwendung. Folglich ist auch die Instanziierung der Instanz eines Typs unterschiedlich. Während bei einer formalen Beschreibung des Typs (z. B. Klasse in C++) die Instanz genau alle Attribute enthält, die der Typ definiert, ist dies bei einer reinen Begriffsbeschreibung nicht der Fall. Hier werden implizit Eigenschaften beschrieben, die zwischen Typ und Instanz gleich sind, um Instanzen zu klassifizieren.

2.5 Identifikation von Objekten

Um Objekte eindeutig zu identifizieren, werden Namensräume benötigt. Ein Namensraum definiert einen Bereich, innerhalb dessen jeder Bezeichner nur einmal auftreten darf. Jedes Objekt besitzt einen solchen Bezeichner und ist somit einem Namensraum zugewiesen. Bei den Bezeichnern können zwei Arten unterschieden werden: lokal eindeutiger und global eindeutiger Bezeichner. Ein lokal eindeutiger Bezeichner ist nur innerhalb seines Namensraums eindeutig. Ein global eindeutiger Bezeichner ist hingegen weltweit eindeutig. Dies wird dadurch realisiert, dass global eindeutige Namensräume existieren. Um global eindeutige Bezeichner festzulegen, können verschiedene Standards angewendet werden. In dieser Arbeit werden drei Standards genutzt: *ISO 29002-5: Industrielle Automatisierungssysteme und Integration - Austausch von Merkmaldaten - Teil 5: Identifikationsschema* [49], *RFC 3986: Uniform Resource Identifier (URI)* [50] und *RFC 4122: A Universally Unique Identifier (UUID)*[5] [51]. Die Erstellung eines global eindeutigen Bezeichners für ein lokales Objekt kann durch die Aneinanderreihung der einzelnen Bezeichner aller Vaterobjekte erfolgen. Das letzte Vaterobjekt muss in der Kette einen global eindeutigen Bezeichner aufweisen. Diese Bezeichner können je nach Anzahl der überlagerten Namensräume sehr lang werden, sodass man für die konkrete Nutzung auf lokal eindeutige Bezeichner wechselt.

[5] Auch bekannt als Globally Unique Identifier (GUID).

3 Object Constraint Language

Graphische Notationen für Modellsprachen gewinnen zunehmend an Bekanntheit, da diese im Vergleich zu textuellen Notationen meistens verständlicher sind. Jedoch gibt es Schwierigkeiten bei der Modellierung von Bedingungen bzw. Einschränkungen (Constraints). Für häufig auftretende Constraints, wie z. B. Kardinalitäten von Assoziationen, wurden graphische Abkürzungen eingeführt [29]. Allerdings lassen sich diese nicht verallgemeinern und sind ausschließlich für einen kleinen Satz sehr generischer Constraints möglich. Aus diesem Grund müssen weitere Bedingungen in einer zugehörigen textuellen Sprache definiert werden. Dies kann durch natürliche Sprache erfolgen (vgl. Abschnitt 2.3). Da natürliche Texte meistens mehrdeutig und zudem nur schwer maschinenverarbeitbar sind, sollte möglichst eine formale Sprache bevorzugt genutzt werden [52]. Mit OCL existiert eine Sprache, die es ermöglicht Bedingungen bzw. Einschränkungen formal zu beschreiben.

OCL ist eine universelle, textuell semi-formale Sprache, um unter anderem Invarianten oder Vor- und Nachbedingungen von Methoden in objektorientierten Modellen formal zu beschreiben. Die Sprache wurde 1995 ursprünglich von IBM entwickelt und 1997 in die Modellierungssprache UML integriert [42]. Zudem wurde OCL als Standard von der Object Management Group (OMG) veröffentlicht und liegt zum Verfassungszeitpunkt dieser Arbeit in der Version 2.4 [53] vor.

Zunächst wurde die Sprache für die Definition von Constraints in UML genutzt. Schnell wurde das Potenzial der Sprache erkannt, wodurch OCL zu einer Hauptkomponente in vielen modellgetriebenen Engineering-Techniken [42] wurde. OCL wird unter anderem in domänenspezifischen Sprachen oder für die Code-Generierung mit Hilfe von Templates genutzt. Aufgrund der Möglichkeit durch OCL formal komplexe Abfragen auszudrücken, ist die Sprache mittlerweile auch in vielen Modelltransformationssprachen integriert, wie z. B. in Query View Transformation (QVT). Der Vorteil von OCL ist, dass die Sprache auf der Prädikatenlogik aufbaut und diese erweitert [54]. Für die Definition der Sprachelemente werden aber keine mathematischen Symbole verwendet. Vielmehr werden Elemente einer natürlichen Sprache genutzt, weswegen auch Nicht-Mathematiker oder -Informatiker die Sprachelemente verstehen und nutzen können. Zusammengefasst kann OCL für eine Vielzahl von Anwendungen und Arten von Ausdrücken verwendet werden.

Die OCL-Spezifikation [53] definiert die abstrakte Syntax sowie die Semantik der Sprache. Die Definitionen sind in Form eines Metamodells sowie durch natürliche Sprache beschrieben. Ergänzt wird die Definition durch formal definierte Regeln und Operationen[1]. Zusätzlich werden in der Spezifikation auch eine konkrete Syntax der Sprache sowie mögliche Anwendungsbeispiele gezeigt. Die Sprache ermöglicht die Definition von konkreten Ausdrücken, die zur Laufzeit ausgewertet werden können. OCL definiert hierfür ein

[1]Die Definitionen der Regeln und Operationen werden mit Hilfe der eigenen Sprachelemente definiert.

eigenes Typ-System, bestehend aus einfachen Datentypen und Sammlungen[2]. Das Ergebnis der Auswertung eines Ausdrucks ist immer konform zu einem dieser Typen. Aufgrund dessen wird OCL als eine typisierte Sprache aufgefasst. Zusätzlich entspricht die Oberklasse *Class* den Typen des Metamodells, für welches Ausdrücke spezifiziert werden sollen. Mit Hilfe dieser Ausdrücke können formale und komplexe Abfragen auf eigenen Modellen formuliert werden. Lediglich das Anlegen von Objekten eigens definierter Typen ist nicht möglich. Des Weiteren ermöglichen die Sprachelemente die Definition von Ausdrücken dahingehend, dass diese keine Seiteneffekte bei der Ausführung mit sich bringen. Das bedeutet, dass bei der Ausführung eines Ausdrucks keine Modifikationen an bestehenden Objekten erfolgen. Für die einfache Erstellung von Ausdrücken sind die Sprachelemente in deklarativer Form definiert[3]. Da nicht für alle Anwendungsfälle alle Sprachelemente benötigt werden, definiert die Spezifikation zusätzlich noch BasicOCL, welches ein Minimalset an Sprachelementen enthält. Um die Sprache nutzbar zu machen, z. B. in einer konkreten Implementierung, wird neben der abstrakten Syntax zusätzlich eine konkrete Syntax benötigt. Die Spezifikation [53] definiert selbst eine konkrete Sprache in textueller Form. Eine konkrete Syntax in graphischer Form ist in [52] vorgeschlagen.

In den nächsten Abschnitten folgt eine Einführung in BasicOCL, da die dort beschriebenen Elemente für die vorliegende Arbeit ausreichend sind. Für eine vollständige Beschreibung aller Elemente und Operationen wird auf den Standard [53] verwiesen[4]. Das Kapitel unterteilt sich in drei Teilabschnitte. Zunächst werden Constraints erklärt und beschrieben, wie die Navigation auf Klasseneigenschaften und Operationen mit OCL funktioniert (Abschnitt 3.1). Danach wird in Abschnitt 3.2 die abstrakte Syntax von BasicOCL beschrieben. Diese unterteilt sich in die Vorstellung des Typ-Systems und der OCL-Ausdrücke. Abschließend wird die konkrete Syntax von OCL kurz beschrieben, wobei der Fokus auf den für diese Arbeit benötigten Sprachelementen liegt (Abschnitt 3.3).

3.1 Anwendung von OCL

OCL stellt Sprachelemente zur Definition von Ausdrücken zur Verfügung. Diese können unterschiedlichster Art sein, z. B. Variablenausdrücke, IfThenElse-Ausdrücke oder Ausdrücke zum Zugriff auf Attribute oder Operationen von Objekten[5]. Sofern eine Einschränkung in einem UML-Modell benötigt wird, können die OCL Sprachelementen genutzt werden. Diese Stellen definieren zwangsläufig auch die Semantik für das Ergebnis des Ausdrucks. Der Ausdruck ist wiederum durch die Stelle einem konkreten Kontext zugewiesen, innerhalb dessen er auszuwerten ist. OCL definiert einige Standardstellen, an denen OCL-Ausdrücke genutzt werden können. In Abbildung 3.1 sind zwei dieser Stellen stellvertretend gezeigt: die Definition einer Invariante für eine Klasse und die Definition eines Anfangswerts von einem Attribut.

Um die Stellen textuell zu beschreiben, führt OCL weitere Sprachelemente ein. Die Definition des Kontextes einer Stelle erfolgt durch das Schlüsselwort *context <classifier>*

[2]Eine Auflistung der verschiedenen Arten von Datentypen und Sammlungen ist in 3.2 beschrieben.
[3]Eine genauere Beschreibung von deklarativ und dem Unterschied zu imperativ ist Abschnitt 5.2.1 gegeben.
[4]Eine gute Einführung in die Grundkonzepte von OCL wird in [42] gegeben.
[5]Die möglichen Arten werden in Abschnitt 3.2 vorgestellt.

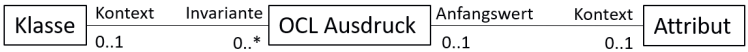

Abbildung 3.1: Beispiele von Standardstellen, an denen OCL-Ausdrücke genutzt werden können

und bezieht sich auf ein konkretes UML-Modellelement (Klasse, Attribut oder Methode), z.B. *context Person*, *context Person::age* oder *context Person::getAge()*. Zusätzlich kann der Kontext auch Variablendeklarationen beinhalten, die innerhalb dieses Kontextes genutzt werden können. Dies wird über folgende Syntax definiert:

context <v_name_1> : <v_type_1> , ... , <v_name_n> : <v_type_n>

Damit besteht die Option, diese Variablen innerhalb des Kontextes zu nutzen, beispielsweise um Invarianten für verschiedene Typen innerhalb eines Kontextes zu definieren oder um auf die Werte dieser Variablen zuzugreifen[6]. Wird nur eine Klasse oder Methode als Kontext angegeben, dann gelten die angegebenen Constraints für alle Instanzen dieser Klasse oder Methode.

Es gibt verschiedene Arten von Constraints, die durch ihre Schlüsselwörter unterschieden werden. Der Constraint an sich wird durch OCL-Ausdrücke formuliert. Beispielhaft werden die beiden Constraints Invariante und Anfangswert kurz beschrieben[7]:

Invariante (Invariant)

Eine Invariante ist eine Einschränkung, die zu jeder Zeit für eine Instanz des UML-Modellelements, welches durch den Kontext festgelegt wird, gelten muss. Die Auswertung einer Invariante ergibt einen booleschen Wert, der stets wahr sein muss, damit die Einschränkung erfüllt ist. Eine Invariante wird mit *inv: <Boolean OCL expression>* definiert. Der Ausdruck muss für den kompletten Lebenszyklus des Objekts gelten, also „wahr“ ergeben. Nachfolgend ist ein Beispiel für das Attribut *age* der Klasse *Person* gegeben. Das Alter der Klasse Person muss immer größer gleich als Null sein:

```
context Person
  inv: age >= 0
```

Anfangswert (Initial Value)

Mithilfe des Constraints *Anfangswert* kann der Wert eines Attributs festgelegt werden, der initial vorliegen muss. Bei der Auswertung muss der Typ des Anfangswerts dem Typen des Attributs, für den der Constraint gilt, entsprechen. Um den Anfangswert eines Attributs festzulegen, wird die Syntax *init: <OCL expression>* verwendet. Im nachfolgenden Beispiel wird ein Constraint deklariert, der aussagt, dass das Attribut *isMarried* der Klasse *Person* initial den Wert *false* haben muss.

```
context Person::isMarried: Boolean
  init: false
```

[6]Dies wird in dem Konzept dieser Arbeit verwendet.

[7]Für die Beschreibung weiterer bereits definierter Constraints wird auf die entsprechende Spezifikation [53] verwiesen.

Um auf die verschiedenen Attribute und Methoden eines Objekts sowie auf die Attribute und Methoden einer Sammlung von Objekten zuzugreifen, gibt es den Punkt- und den Pfeil-Operator. Um Attribute oder Methoden eines einzelnen Objekts zu referenzieren, wird der Punkt-Operator verwendet. Für den Zugriff auf Attribute oder Methoden einer Sammlung von Objekten wird der Pfeil-Operator genutzt. Im nachfolgenden Beispiel werden beide Varianten gezeigt.

Das Beispiel enthält drei Invarianten: Das Alter soll stets größer gleich 0 sein, der Aufruf der Funktion *getName* soll dem Wert des Attributs *name* entsprechen und die Anzahl an Autos der Person soll kleiner gleich 2 sein. Die Attribute *age*, *name* und *car* sowie die Methode *getName()* werden über den Punktoperator referenziert, da diese zur Instanz der Klasse *Person* gehören, die ein Einzelobjekt darstellt. Die Methode *size()* hingegen gehört zum Attribute *car*. Dieses stellt eine Menge von Objekten dar, genauer gesagt eine Menge von Instanzen der Klasse *Car*. Aus diesem Grund wird der Pfeil-Operator verwendet. Definiert der Kontext nur eine Klasse oder Methode, kann für die Navigation innerhalb dieser Klasse das Schlüsselwort *self* genutzt werden. Mit *self* wird die Instanz der Klasse des Kontexts referenziert, hier eine Instanz der Klasse *Person*.

```
context Person
  inv: self.age >= 0
  inv: self.getName() = self.name
  inv: self.car->size() <= 2
```

3.2 Abstrakte Syntax von BasicOCL

Die abstrakte Syntax von BasicOCL definiert ein Typ-System und die OCL-Ausdrücke. Nachfolgend wird die in BasicOCL verwendete abstrakte Syntax für beide Bestandteile vorgestellt.

Typ-System

OCL definiert ein eigenes Typ-System mit zugehörigen Operationen, die auf dem jeweiligen Typ ausgeführt werden können. Beispielsweise können für Objekte des Typs *Integer* die Operationen *+*, *-* oder *abs()*[8] und für Objekte des Typs *Collection* die Operationen *collect()*[9] oder *forAll()*[10] ausgeführt werden. In Abbildung 3.2 ist das Typ-System von BasicOCL dargestellt. Die in weiß dargestellten Typen sind aus UML [39] entnommen. Nachfolgend werden die einzelnen Typen kurz vorgestellt, ohne auf die zugehörigen Operationen einzugehen. Für interessierte Leser wird auf die Spezifikation [53] verwiesen.

Alle Typen erben von der UML Klasse *Type*[11]. *Type* ist eine abstrakte Klasse, die ein typisiertes Element beschreibt. Auf der ersten Ebene werden die Klassen *InvalidType*, *VoidType*, *DataType*, *AnyType*, *Class* und *TemplateParameterType* definiert. Die Klasse

[8] Berechnung des absoluten Wertes.
[9] Anwendung einer Mapping-Funktion auf alle Elemente.
[10] Iteration über alle Elemente dieser Collection.
[11] In BasicOCL wird nicht die Klasse *Classifier* als Basisklasse genutzt, sondern die Klasse *Type*. Folglich muss jeder Verweis auf die Klasse *Classifier* als Verweis auf die Klasse *Type* uminterpretiert werden [53].

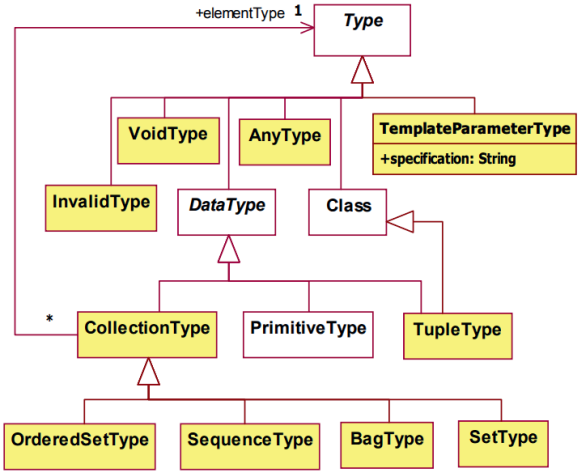

Abbildung 3.2: BasicOCL Typenmodell [53]

InvalidType stellt einen Typen für ungültige Werte dar, welche z. B. bei der Anwendung einer Operation auf Typen, für die diese Operation nicht definiert ist, entstehen. Der Typ ist vergleichbar mit *Exceptions* aus Programmiersprachen. Für die Darstellung eines undefinierten Werts wird die Klasse *VoidType* definiert. Diese tritt immer dann auf, wenn ein Ausdruck als undefiniert evaluiert wird, z. B. wenn auf ein Attribut oder ein Objekt zugegriffen wird, das nicht existiert. In Programmiersprachen wird oft das Schlüsselwort *null* verwendet. Die Klasse *AnyType* stellt einen verallgemeinerten Typen dar, dem alle anderen Typen entsprechen. Mithilfe dieser Klasse können Operationen definiert werden, die für alle Typen gelten. Die UML-Klasse *Class* ist die Metaklasse einer Klasse und kann zur Beschreibung von eigenen Klassen genutzt werden, wie es heutzutage oftmals in Klassendiagrammen erfolgt. Die Klasse *TemplateParameterType* stellt einen parametrierbaren Typen dar und wird beispielsweise bei wiederverwendeten Ausdrücken benötigt. Als letzte Klasse auf der oberen Ebene wird die Klasse *DataType* aus UML genutzt. Diese Klasse beschreibt einen Typen, dessen Instanzen durch einen Wert dargestellt werden. Von dieser Klasse werden drei verschiedene Unterklassen abgeleitet: *CollectionType*, *PrimitiveType* und *TupleType*. Die abstrakte Klasse *CollectionType* beschreibt dabei eine Liste von Elementen eines bestimmten Typs, der über die Assoziation *elementType* festgelegt wird. Dabei gibt es keine Einschränkungen, so können z. B. auch verschachtelte Collections definiert werden. Die abstrakte Klasse hat vier Unterklassen: *OrderedSetType*, *SequenceType*, *BagType* und *SetType*. Die Klassen stellen dabei immer eine Sammlung von Elementen dar und unterscheiden sich in zwei Punkten:

1. Elemente dürfen nur einmal oder mehrfach in der Sammlung vorhanden sein und
2. Elemente liegen in einer Reihenfolge vor oder sind ungeordnet.

In Tabelle 3.1 sind die Ausprägungen den einzelnen Unterklassen zugeordnet.

Tabelle 3.1: Unterscheidung der Subklassen von CollectionType

Kriterium	OrderedSetType	SequenceType	BagType	SetType
Anzahl	Einmal	Mehrfach	Mehrfach	Einmal
Reihenfolge	Geordnet	Geordnet	Ungeordnet	Ungeordnet

Auf der zweiten Ebene wird die UML-Klasse *PrimitiveType* wiederverwendet. Diese fasst einfache Datentypen ohne Unterstrukturen zusammen. Insgesamt werden fünf einfache Datentypen definiert: *Boolean*, *Integer*, *Real*, *String* und *UnlimitedNatural*[12]. Die letzte Klasse *TupleType* beschreibt eine Menge von verschiedenen Typen. Dadurch ist es möglich, eigene Datenstrukturen innerhalb von OCL zu erstellen. In Programmiersprachen wird dies oft als *struct* bezeichnet.

OCL-Ausdrücke

Um Constraints zu definieren, werden Ausdrücke benötigt. In den vorherigen Beispielen wurden diese dargestellt, ohne näher auf die Art der Ausdrücke einzugehen. In Abbildung 3.3 ist eine vereinfachte Darstellung des Metamodells der OCL-Ausdrücke von BasicOCL aus der Spezifikation [53] dargestellt. Das vollständige Modell sowie die Semantik sind in der Spezifikation ausführlich beschrieben.

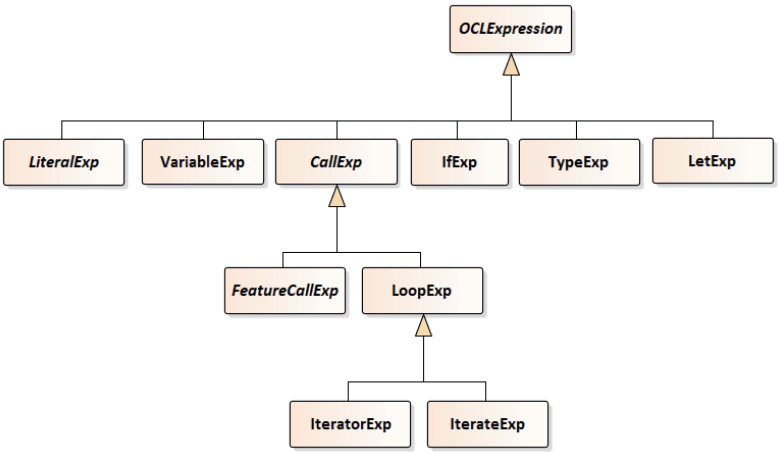

Abbildung 3.3: BasicOCL Metamodell nach [21]

Auf oberster Ebene werden insgesamt sechs verschiedene Klassen definiert. Die Klasse *LiteralExp* definiert einen Ausdruck, welcher ein gegebenes Literal eines primitiven Typs

[12]Instanzen der Klasse *UnlimitedNatural* sind Werte der natürlichen Zahlen (0,1,2,...,*). Das Zeichen * stellt den Wert „unendlich“ dar. Zumeist findet sich dieser Typ in der Beschreibung von Multiplizität einer Assoziation.

auswertet und als Ergebnis zurückgibt, z. B. String oder Integer. Durch die Klasse *VariableExp* können Variablen ausgewertet und das entsprechende Ergebnis verwendet werden. Die referenzierte Variable kann vorher explizit (z. B. mit *LetExp*) oder implizit (z. B. *self*) definiert sein. Die abstrakte Klasse *CallExp* stellt einen Ausdruck dar, welcher sich auf ein Attribut, eine Methode oder auf einen vordefinierten Iterator für eine Collection bezieht. Das Ergebnis ist der Wert des Attributs, der Methode oder der Iteration. Die Unterklasse *FeatureCallExp* wird für die Auswertung von Attributen und Methoden verwendet, während die Unterklasse *LoopExp* für die Auswertung einer Iteration über eine Collection genutzt wird. Die beiden Subklassen *IteratorExp* und *IterateExp* iterieren über die Elemente einer Collection, werten einen vorgegebenen Ausdruck für jedes Element aus und geben das Ergebnis (z. B. eine neue Collection oder einen booleschen Wert) zurück. Die IteratorExp ist eine Kurzform der IterateExp für bestimmte Methoden, z. B. *forAll*, *select* und *sortedBy*. Die Klasse *IfExp* ist ein Ausdruck, welcher eine boolesche Bedingung auswertet und je nach Ergebnis einen von zwei gegebenen Ausdrücken zurückgibt. Um einen Typen zu referenzieren, kann die Klasse *TypeExp* genutzt werden. Abschließend kann mit der Klasse *LetExp* ein Ausdruck definiert werden, welcher eine neue Variable eines Typs erstellt und mit einem gegebenen Wert initialisiert. Der Wert der Variable kann danach nicht mehr geändert werden. Um die verschiedenen Ausdrücke zu verdeutlichen, wird in Abbildung 3.4 folgende Invariante mit Hilfe der vorgenannten OCL-Klassen dargestellt:

```
context machine
  inv: self.nextjobs->select(j | j.prio = 1)->size() <= 1
```

Die Invariante sagt aus, dass zu jeder Zeit in der Liste der nächsten Aufträge einer Maschine maximal ein Auftrag enthalten sein darf, der die Priorität 1 hat. Der abstrakte Syntaxbaum beginnt mit einer Vergleichsoperation auf der obersten Ebene und gliedert sich anschließend unter Nutzung der OCL-Klassen weiter auf.

3.3 Konkrete Syntax von BasicOCL

In der OCL Spezifikation wird neben der abstrakten Syntax auch eine konkrete Syntax beschrieben. Diese ist in der erweiterten Backus-Naur-Form (EBNF) formuliert. Zusätzlich ist das Mapping zu den Elementen der abstrakten Syntax angegeben. Für die Unterscheidung zwischen der abstrakten und konkreten Syntax wird bei den Elementen der konkreten Syntax der Term *CS*[13] als Suffix angehängt.

Für die Erstellung von neuen Syntaxelementen (Wörtern bzw. Symbolfolgen) müssen Produktionsregeln formuliert werden. Diese geben an, wie Syntaxelemente mit Hilfe der Kombination von anderen Syntaxelementen definiert (produziert) werden. Eine Produktionsregel besteht aus einem linken und einem rechten Teil, die durch das Syntaxelement `::=` getrennt werden. Der linke Teil stellt dabei das zu erstellende Syntaxelement dar und der rechte Teil die Anleitung zur Produktion dieses Syntaxelements.

Bei den Syntaxelementen wird in Terminal- und Nichterminalsymbole unterschieden. Ein Terminalsymbol beschreibt ein Symbol, welches nicht durch andere Symbole ersetzt werden

[13] CS steht für Concrete Syntax.

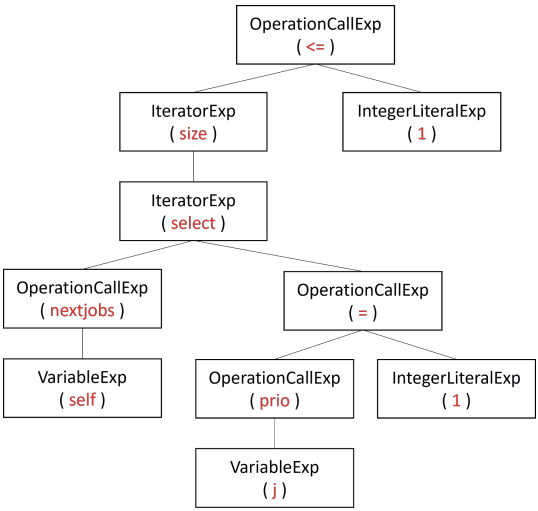

Abbildung 3.4: Darstellung der Invariante durch OCL-Klassen

kann. Ein Nichtterminalsymbol ist ein Symbol, für das eine Produktionsregel existiert. Folglich kann das Symbol durch die Symbole der Produktionsregel ersetzt werden.

In OCL existieren allgemeingültige und vordefinierte Terminalsymbole, um komplexere Produktionsregeln zu erstellen. Die für diese Arbeit relevanten Symbole werden nachfolgend vorgestellt: Durch die Nutzung von Klammern können zusammengehörige Symbole gruppiert werden. Um optionale Symbole oder Symbolgruppen zu kennzeichnen, wird das Fragezeichen-Symbol an das Symbol oder die Symbolgruppe angehängt. Falls Symbole innerhalb einer Produktionsregel mehrfach benutzt werden, können diese durch die Verwendung einer Nummer in eckigen Klammern als Suffix unterschieden werden. Kommentare können zeilenweise durch die doppelte Nutzung des Terminalsymbols „-“ oder paragraphenweise zwischen den Terminalsymbolen `/*` und `*/` hinzugefügt werden. Zur Veranschaulichung der Verwendung dieser Symbole wird die Produktionsregel für die *variableDeclarationListCS* beschrieben. Diese erzeugt eine Liste von Variablen, die beispielsweise in einem Tupel benötigt wird. Eine *variableDeclarationListCS* besteht immer aus mindestens einem Nichtterminalsymbol *variableDeclarationCS* gefolgt von einer Gruppe, bestehend aus dem in dieser Produktionsregel definierten Terminalsymbol in Form eines Kommas und einer weiteren *variableDeclarationListCS*. An dieser Stelle kann rekursiv verfahren werden, sodass weitere Variablen durch ein Komma getrennt angehängt werden. Das Fragezeichen am Ende der Gruppe definiert die Gruppe als optional. Somit kann die Gruppe auch entfallen, was zum Ende der Rekursion führt.

```
variableDeclarationListCS[1] ::= VariableDeclarationCS
        (‘,’variableDeclarationListCS[2] )?
```

Wichtige Produktionsregeln für diese Arbeit

Für diese Arbeit sind sieben Produktionsregeln wichtig, da diese in Kapitel 9 in der konkreten Syntax verwendet werden. An dieser Stelle wird nur auf die Semantik dieser Produktionsregeln eingegangen. Die formale Beschreibung der Produktionsregeln kann in der Spezifikation [53] nachgeschlagen werden.

Die erste Produktionsregel ist die Erstellung des Syntaxelements *simpleNameCS*, welche es ermöglicht einen Namen als String zu formulieren. Es können beliebige Zeichen verwendet werden, wobei immer mit einem Buchstaben, einem Unterstrich oder dem $-Zeichen begonnen werden muss. Die nutzbaren Unicode-Zeichen sind in [53] festgelegt. Danach können beliebig weitere dieser Zeichen und zusätzlich auch die Ziffern 0 bis 9 folgen.

Eine weitere wichtige Produktionsregel ist die für das Syntaxelement *OCLExpressionCS*, welche wiederum aus anderen Produktionsregeln besteht. Dabei ist es möglich, die zugehörigen Produktionsregeln für die Sub-Klassen der OCLExpression aus der abstrakten Syntax zu nutzen, z. B. *CallExpCS*, *VariableExpCS* oder *LiteralExpCS*. Für weitere Informationen zu diesen Produktionsregeln wird auf die Spezifikation [53] verwiesen.

Für die Erstellung von String-Literalen ist die Produktionsregel des Syntaxelements *StringLiteralExpCS* definiert. Mit dieser ist es möglich, ein String-Literal zu erzeugen, welches durch eine Sequenz aus Zeichen oder Escape-Sequenzen beschrieben wird. Eine Escape-Sequenz ermöglicht die Darstellung von Sonderfunktionen in Text-Zeichen, z. B. $\backslash t$ für ein Tabulator oder $\backslash n$ für eine neue Zeile.

Mit dem Syntaxelement *CollectionLiteralExpCS* können Sammlungen von Objekten dargestellt werden. Dafür muss in der Produktionsregel zunächst der Typ der Sammlungen Set, Bag, Sequence, Collection oder OrderedSet angegeben werden. Danach folgen die eigentlichen Elemente.

Für die Definition eines Typnamens in einem Ausdruck kann das Syntaxelement *typeCS* verwendet werden. Für die Definition werden andere Produktionsregeln wiederverwendet. Zum Beispiel *primitiveTypeCS* für die Erstellung von einfachen Datentypen, wie String oder Boolean, oder *pathNameCS*, welche eine Sequenz von Strings definiert, zur Beschreibung eines Pfadnamens.

Die letzten zwei Produktionsregeln dienen der Definition einer Liste von Parametern *paramtersCS* oder einer Liste von Argumenten *argumentsCS*. Die erstgenannte Regel ist eine Sequenz von Variablendeklarationen, die durch ein Komma getrennt werden. Jede dieser Variablendeklarationen besteht aus einem Namen *simpleNameCS* und kann optional den Typ *typeCS* sowie einen Ausdruck für den Initialwert *OCLExpressionCS* festlegen. Die Produktionsregel zur Erstellung des Syntaxelements *argumentsCS* ist durch eine Sequenz von *OCLExpressionCS*, getrennt durch ein Komma, definiert.

4 Interoperabilität

Damit sich zwei Entitäten untereinander verständigen können, brauchen diese ein gemeinsames Verständnis über die ausgetauschten Daten. Als Beispiel wird folgendes Szenario betrachtet: Zwei Menschen aus unterschiedlichen Ländern wollen miteinander interagieren. Beide kommen aus unterschiedlichen kulturellen, politischen sowie ethnischen Kreisen und sprechen jeweils eine andere Sprache, die zusätzlich auf unterschiedlichen Alphabeten basieren. Diese Personen können sich zunächst nicht mittels einer gesprochenen Sprache unterhalten. Aus diesem Grund muss entweder einer von beiden Personen die jeweils andere Sprache lernen oder beide lernen eine gemeinsame ggf. einfachere oder weit verbreitetere Sprache[1]. Angenommen beide Personen haben eine gemeinsame Sprache in ihren Grundformen gelernt. Das Erlernen ist jedoch unterschiedlich erfolgt, sodass die beiden Personen ein unterschiedliches Vokabular in der Sprache besitzen, wobei das Haupt-Vokabular gleich ist. Ab diesem Zeitpunkt ist es den Personen möglich, sich mit Hilfe der gemeinsamen Sprache über Dinge, die im Haupt-Vokabular enthalten sind, auszutauschen. Nutzt jedoch eine Person ein Wort aus dem Vokabular, das die andere Person nicht versteht, ist zunächst wiederum keine 100%ige Kommunikation möglich. Der Kommunikationspartner kann jedoch ggf. die Bedeutung des fehlenden Wortes durch die anderen Wörter im Satz vermuten. Dabei tritt ein weiteres Problem auf: die Semantik der Wörter. Wörter können in verschiedenen Kontexten unterschiedliche Bedeutungen haben. Der Kommunikator muss anhand des Gesprächskontextes ermitteln, welche Bedeutung den Begriffen gerade zugeordnet werden. Nur wenn dies korrekt erfolgt, können sich zwei Personen verstehen. Ein weiteres Problem tritt auf, wenn die Grammatik nicht korrekt ist. Menschen können dies durch Verständnis des Kontextes herausfiltern und die Informationen dennoch verstehen.

Bei der Übertragung dieses Szenarios auf die Kommunikation zwischen Maschinen bzw. Software-Applikationen treten dieselben Probleme auf. Damit Applikationen interagieren können, bedarf es zunächst einer gemeinsamen Sprache[2]. Dies ist in der Regel ein Datenformat. In einem nächsten Schritt müssen die im Datenformat enthaltenen Daten von beiden Seiten gleich analysiert (z. B. Einheit, Bedeutung, Datentyp) und anschließend korrekt interpretiert werden (z. B. Messwert vom Sensor X an Anlage A ist zu hoch). Erst wenn diese Voraussetzungen gegeben sind, kann von einer Interaktion zwischen Applikationen gesprochen werden. Ein Unterschied ist jedoch, dass eine Maschine zunächst kein Kontext-Wissen besitzt und dementsprechend mit fehlerbehafteten Datenformaten oder Informationen nicht umgehen kann. Es existieren bereits Ansätze, die sich mit dieser Thematik beschäftigen, auf die nachfolgend verwiesen wird.

Um die verschiedenen Arten der Kommunikationsfähigkeit zwischen Systemen, Applikationen oder auch Organisationen zu klassifizieren, existiert der Begriff *Interoperabilität*. Dabei

[1] Heutzutage ist dies oftmals Englisch.
[2] Genau genommen einer gemeinsamen Syntax.

liegen verschiedene Definitionen von Interoperabilität vor. Einige Zusammenfassungen sind in [54, 55] gegeben. Nachfolgend sind einige Definitionen aufgelistet.

- *Duden:* Fähigkeit unterschiedlicher Systeme, möglichst nahtlos zusammenzuarbeiten
- *Plattform I4.0 Glossar:* Fähigkeit zur aktiven, zweckgebundenen Zusammenarbeit von verschiedenen Komponenten, Systemen, Techniken oder Organisationen
- *Oxford Dictionary:* The ability of computer systems or programs to exchange information
- *Cambridge Dictionary:* The degree to which two products, programs, etc. can be used together, or the quality of being able to be used together
- *ISO/IEC[56]:* The ability of two or more systems or applications to exchange information and to mutually use the information that has been exchanged
- *IEEE[57]:* The ability of two or more systems or components to exchange information and to use the information that has been exchanged

Diese Arbeit handelt von der Interaktion zwischen Software-Applikationen. Dafür wird eine Möglichkeit zur Zusammenarbeit von verschiedenen Software-Applikationen von verschiedensten Herstellern vorgestellt. Interoperabilität wird in dieser Arbeit wie folgt definiert:

Definition 4.1 (Interoperabilität) *Fähigkeit von Systemen und Applikationen, Informationen untereinander auszutauschen und diese für eine aktive und zweckgebundene Zusammenarbeit zu nutzen.*

Interoperabilität lässt sich in verschiedene Stufen einteilen. Welche Stufen existieren und wie diese aufeinander aufbauen, wird im Abschnitt 4.1 dargestellt. In dem darauffolgenden Abschnitt 4.2 werden die derzeitigen Probleme, die in den einzelnen Stufen auftreten, detaillierter beschrieben und aktuelle Ansätze zur Lösung dieser Probleme dargestellt.

4.1 Stufen der Interoperabilität

Es existieren mehrere Modelle, die Interoperabilität in verschiedene Stufen einteilen. Nachfolgend werden einige von diesen vorgestellt und im Anschluss in Bezug zu dieser Arbeit gesetzt.

In Abbildung 4.1 ist die vierstufige Einteilung nach [58] dargestellt. Diese beginnt auf der untersten Stufe mit der *technischen Interoperabilität*, die die Übertragung von Daten sicherstellt. Aufbauend darauf befindet sich die *syntaktische Interoperabilität*, die aussagt, dass zwei Systeme das gleiche Verständnis von Zeichen und Formaten haben. Auf der dritten Stufe (*semantischen Interoperabilität*) wird festgelegt, „aus welchen inhaltlichen Feldern ein Datensatz in welcher Reihenfolge besteht und mit welchen Codes die Daten in den einzelnen Feldern erzeugt werden“ [58]. Wie die Daten weiterverarbeitet werden, z. B. durch abgestimmte Workflows, legt die höchste Stufe, die *organisatorische Interoperabilität*, fest.

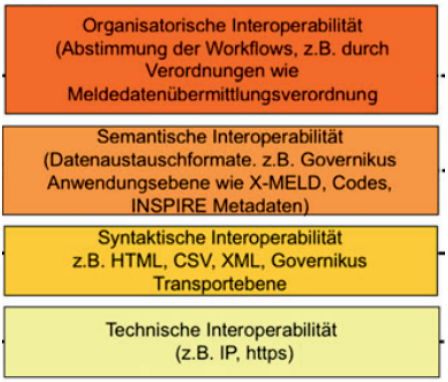

Abbildung 4.1: Interoperabilitätsstufen nach Kubicek [58]

In [59] werden sechs Stufen eingeführt: *no connection* (keine Interoperabilität zwischen Systemen), *technical* (Basis- und Netzwerk-Konnektivität), *syntactical* (Datenaustausch ist gegeben), *semantic* (Verständnis der ausgetauschten Daten), *pragmatic/dynamic* (Anwendbarkeit der Daten in einem Kontext) und *conceptual* (geteiltes Wissen domänenübergreifend). Pantsar Syvaniemie et al. definieren in [60] eine ähnliche Klassifikation mit folgenden sechs Stufen: *connection*, *communication*, *semantic*, *dynamic*, *behavioural* und *conceptual*.

Im militärischen Umfeld hat die NATO mit der System-Interoperabilitäts-Richtlinie (NATO C3 System Interoperability Directive: NIC) eine fünf-stufige Klassifikation entwickelt [61]. In Stufe 0 muss der Anwender die Systeminteroperabilität herstellen (Isolated Interoperability). In Stufe 1 können Systeme bereits Daten untereinander austauschen (Connected Interoperability). Die Weiterverarbeitung der ausgetauschten Daten zu sinnvollen Informationen erfolgt in Stufe 2, z. B. durch definierte Schemata in JSON oder XML (Functional Interoperability). In der nachfolgenden Stufe 3 können die Informationen durch die jeweiligen Systeme automatisiert interpretiert werden, ohne dass vorherige Absprachen zwischen den Systemen notwendig sind (Domain Interoperability). Die Informationen gehören dabei einer bestimmten Domäne an. Werden die Informationen über Domänengrenzen hinweg verstanden, ist die 4. Stufe der Interoperabilität erreicht (Enterprise Interoperability).

Noura, Atiquzzaman und Gaedke beschreiben in [62], dass die Klassifikation im Bereich von Internet of Things (IoT) nicht in Stufen, sondern aus verschiedenen Perspektiven betrachtet werden sollte. In Abbildung 4.2 sind die fünf verschiedenen Perspektiven dargestellt: *Device Interoperability*, *Network Interoperability*, *Syntactical Interoperability*, *Semantic Interoperability* und *Platform Interoperability*. Unter *Device Interoperability* wird eine Fähigkeit verstanden, mit der heterogene Geräte Informationen austauschen können. Es können verschiedene Arten von Geräten von Low-End bis High-End und zudem mit unterschiedlichen Kommunikationsprotokollen vorliegen. Außerdem soll die Integration von neuen Geräten in eine IoT-Plattform möglich sein. Bei *Network Interoperability* geht es darum, dass Geräte aus unterschiedlichen Netzwerken miteinander interagieren können und

dass der Informationsaustausch zwischen den Netzwerken möglich ist. Das Datenformat und die Datenstruktur der ausgetauschten Informationen werden in der *Syntactial Interoperability* festgelegt. Der Fokus liegt auf den Kodierungs- und Dekodierungsregeln der sich verständigenden Systeme. In *Semantic Interoperability* wird die Semantik der übertragenen Daten betrachtet. Dazu muss das Wissen über die Informationen ausgetauscht werden, z. B. über definierte Informationsmodelle. Abschließend wird die *Platform Interoperability* betrachtet, bei der die Plattform-Abhängigkeit von Systemen und deren Schnittstellen im Fokus steht. Verschiedenste Plattformen haben unterschiedliche Anforderungen oder eigene Programmiersprachen, sodass Systeme nicht zwingend auf jeder Plattform ausführbar sind.

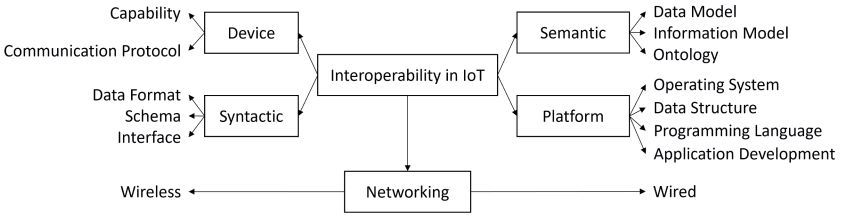

Abbildung 4.2: Perspektiven der Interoperabilität nach [62]

In den ISO/IEC 21823-1 und ISO/IEC 19941 Standards [56, 63] werden fünf Facetten von Interoperabilität beschrieben (siehe Abbildung 4.3): *Transport*, *Syntactic*, *Semantic*, *Behavioural* und *Policy*. Die *Transport Interoperability* umfasst die Kommunikationsinfrastruktur, die benötigt wird, um Daten zwischen zwei Entitäten auszutauschen. Hierzu gehören das physische Medium und die Transportmechanismen (die ersten 4 Layer im ISO/OSI Schichtenmodell [64]). Die detaillierte Beschreibung ist in Teil 2 des ISO/IEC 21823-Standards [65] gegeben. Die *Syntactic Interoperability* beschreibt die Fähigkeit, mit der Systeme oder Geräte Informationen basierend auf ihrer Syntax austauschen können. Beispiele sind: Web Ontology Language (OWL), XML, Resource Description Framework Schema (RDFS) oder JSON. Bei der *Semantic Interoperabilty* steht die Bedeutung des Datenmodells innerhalb eines Kontextes im Fokus, z. B. wie die Daten für eine konkrete Domäne zu interpretieren sind. In Teil 3 des ISO/IEC 21823-Standards [66] ist beschrieben, dass dies durch die Nutzung von Ontologien erfolgen soll. Die *Behavioural Interoperability* befasst sich mit der Nutzung der ausgetauschten Informationen. Verschiedene Entitäten sind für verschiedene Zwecke konzipiert und verfolgen mit den ausgetauschten Informationen unterschiedliche Absichten. Dies beeinträchtigt die anderen Facetten der Interoperabilität nicht. Vielmehr ist die Behavioural Interoperability in den Schnittstellenbeschreibungen definiert. Das bedeutet, es wird geprüft, ob die erwartenden Ergebnisse beim Aufruf einer Operation mit den tatsächlichen Ergebnissen übereinstimmen. Abschließend beschreibt die *Policy Interoperability* die Fähigkeit von Entitäten, innerhalb von rechtlichen, organisatorischen und politischen Rahmenbedingungen zusammenarbeiten zu können.

Die Modelle sind sich im Grunde ähnlich, unterscheiden sich aber in der Anzahl und Interpretation der Stufen. Drei Stufen jedes Modells können dabei als identisch interpretiert

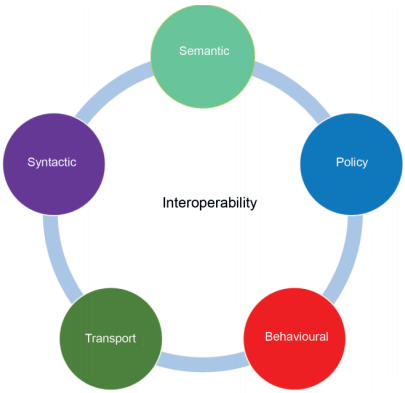

Abbildung 4.3: Facetten der Interoperabilität nach [56]

werden. Die Zuordnung der Stufenbegriffe ist nur bei der Richtlinie der NATO [61] nicht eindeutig, wird aber grob wie folgt angenommen und entspricht den Stufen eins bis drei in Tabelle 4.1:

- Die erste Stufe befasst sich mit dem Transport der Daten und definiert, dass die Komponenten über eine Verbindung zum Austausch von Daten verfügen.
- In der zweiten Stufe wird der syntaktische Austausch mit Hilfe vorgegebener Datenstrukturen gefordert.
- Die dritte Stufe beschreibt die semantische Interpretation der Daten, sodass ein gemeinsames Verständnis über die ausgetauschten Daten vorliegt.

Während die Stufe eins in [58] und [59] *technische Interoperabilität* heißt, wird in [60] der Begriff *connection*, in [61] der Begriff *Connected*, in [62] der Begriff *Network* und in [56, 63] der Begriff *Transportinteroperabilität* verwendet. In Stufe zwei wird hauptsächlich der Begriff *syntaktisch* genutzt. Lediglich in [60] wird stattdessen der Begriff *Communication* und in [61] der Begriff *Functional* verwendet. Bei Stufe drei ist der Hauptbegriff *Semantisch*. Nur in der Richtlinie der NATO [61] wird stattdessen der Begriff *Domain* genutzt.

Aufbauend auf diesen drei Stufen gibt es meistens Stufen, die sich mit dem Verhalten beschäftigt, z. B. Behavioural Interoperability in [56, 60, 63], oder die organisatorischen, rechtlichen und politischen Interoperabilitätskriterien beschreiben sowie Aussagen zu den verwendeten Geräten bzw. Plattformen machen. Im Rahmen dieser Arbeit soll der Fokus auf der semantischen Interoperabilität liegen, weshalb die vorgestellte Einordnung ausreichend ist.

Tabelle 4.1: Vergleich der verschiedenen Stufensysteme

Stufe	[58]	[59]	[60]	[61]	[62]	[56, 63]
0	—	No connection	—	Isolated	Device	—
1	Technisch	Technical	Connection	Connected	Network	Transport
2	Syntaktisch	Syntactical	Communication	Functional	Syntactical	Syntactic
3	Semantisch	Semantic	Semantic	Domain	Semantic	Semantic
4	Organisatorisch	Pragmatic / Dynamic	Dynamic	Enterprise	Platform	Behavioural
5	—	Conceptual	Behavioural	—	—	Policy
6	—	—	Conceptual	—	—	—

4.2 Aktuelle Ansätze für Interoperabilität

In [67] wurden 15 der aktuellen Horizon2020-Projekte analysiert und aufgezeigt, für welche Stufen der Interoperabilität diese jeweils Lösungen bereitstellen. Ein detaillierter Vergleich ist in [62] gegeben, indem 30 verschiedene Lösungen für verschiedene Stufen der Interoperabilität miteinander verglichen wurden. Die meisten Ansätze beschränken sich auf die Transport- und die syntaktische Interoperabilität. Lediglich sieben der 30 Lösungsvorschläge betrachten zudem die semantische Interoperabilität. Das lässt sich dadurch erklären, dass zunächst Lösungen für die unteren Schichten entwickelt werden sollten, bevor Lösungen für die oberen Schichten erforscht werden. Die semantische Interoperabilität wird aktuell in vielen Forschungsaktivitäten betrachtet und es entstehen zunehmend Lösungsansätze. In [62, 67–72] werden verschiedenste Ansätze und Forschungsaktivitäten diskutiert.

Für die Transport-Interoperabilität existieren Ansätze basierend auf Adaptern bzw. Gateways oder virtuellen Netzwerken bzw. Overlay-basierten Lösungen und in Form von universalen Netzwerk-Technologien. Im Bereich der syntaktischen Interoperabilität werden ebenfalls Adapter bzw. Gateways genutzt. Zusätzlich werden OpenAPI-Beschreibungen, Metamodell-getriebene Ansätze und Middleware-Konzepte verwendet. Auch aktuelle Projekte und Initiativen in Deutschland versuchen die syntaktische Interoperabilität zu verbessern. Beispielsweise die Plattform Industrie 4.0 mit dem Konzept der Verwaltungsschale oder das BaSys4.0- bzw. BaSys4.2-Projekt mit der Entwicklung eines neuen Middleware-Konzepts.

Zunächst können die Ansätze für die semantische Interoperabilität hinsichtlich ihrer Herkunft unterschieden werden. Während sich [62, 67–70] vorwiegend auf Ansätze im Bereich des IoT konzentrieren, werden in [71, 72] zusätzlich Lösungskonzepte aus dem Bereich von Industrie 4.0 bzw. aus der Automation-Community betrachtet. Aus dem IoT-Kontext ist häufig das Semantic Web [73] mit seinen Ontologien vertreten, während im Bereich von Industrie 4.0 vorwiegend standardisierte Informationsmodelle (wie z. B. Teilmodelle in Verwaltungsschalen) diskutiert werden.

So verschieden die Ansätze auf den einzelnen Schichten sind, so gleich sind diese in ihren übergeordneten Konzepten. Zwei grundlegende Ansatzkonzepte, die häufig genutzt werden,

sind die Standardisierung und das Mapping.

Der offensichtlichste Schritt zur 100%igen Interoperabilität ist die vollständige Standardisierung. Dies bedeutet, es existiert für jede Schicht genau eine Lösung, die von allen Systemen umgesetzt werden muss. Am Beispiel der semantischen Interoperabilität bedeutet dies, dass ein definiertes Modell vorliegt, mit dem alle Informationen Domänen- und Branchen-übergreifend abgebildet werden können. Dies ist aktuell nicht gegeben und liegt unter anderem daran, dass durch die freie Marktwirtschaft in den letzten Jahren Umsatz durch unterschiedliche und nicht zusammen nutzbare Modelle der einzelnen Firmen erwirtschaftet wurde. Zudem wurden aufgrund unterschiedlicher Ziele und Interessen individuelle Lösungen für einzelne Domänen oder Firmen entwickelt [74]. Das Ergebnis ist ein heterogenes Feld an Lösungen, wobei jeder Entwickler seine Lösung am zielführendsten hält. Das nur eine übergreifende Lösung existiert, ist aus Sicht des Verfassers utopisch. Aus diesem Grund werden nachfolgend alternative Lösungen diskutiert.

Eine Möglichkeit, die aktuell am stärksten vorangetrieben wird, sind standardisierte Informationsmodelle für einzelne Domänen bzw. Branchen durch Standardisierungsorganisationen. Unter einem Informationsmodell wird „im mathematisch-algebraischem Sinne (...) ein zusammengesetzter Abstrakter Datentyp (ADT) mit mehreren Grundmengen (Sorten), Variablen und Axiomen, Regeln und Funktionen zwischen den Sorten (verstanden)(...). Die Anleitungen zur Erstellung von Informationsmodellen sind vielfältig und reichen von Glossaren und Thesauri über objektorientierte Klassifikationen (z.B. AutomationML) bis hin zu Modellen auf formaler Logik (z. B. Ontologien)“ [72]. Ein ADT kann in Form von Datenmodellen mit semantischer Bedeutung oder in Form von Netzen in Ontologien dargestellt werden. Unter Datenmodell mit semantischer Bedeutung werden konkrete Informationsmodelle für einen Anwendungsfall verstanden, bei dem die Semantik der einzelnen Elemente spezifiziert wird. In sogenannten Templates oder Spezifikationen werden diese festgehalten, zur Laufzeit instanziiert und mit aktuellen Daten befüllt. Beispiele sind die OPC UA Companion Specifications, die anwendungsfallspezifisch die Eigenschaften zusammengehörig mit den Elementtypen des OPC UA Metamodells modellieren, um diese in konkreten OPC UA Server Instanzen zu nutzen und mit Werten zu füllen. Des Weiteren entwickeln PROFINET, CAN, IOLINK und EthernetIP entsprechende Geräteprofile und die Home Gateway Initiative (HGI) veröffentlichte das Smart Device Template (SDT) für die Gerätemodellierung. Die im Zuge der Industrie 4.0-Initiative entwickelte Verwaltungsschale definiert ebenfalls Teilmodelle für diesen Zweck. Im Bereich der Ontologien entstehen ebenfalls standardisierte Modelle, wie z. B. oneM2M, ETSI SAREF, W3C SSN, IBM Watson, SenML, NGSI-LD, die auf den Semantic Web Technologien, wie z. B. Resource Description Framework (RDF), OWL und SPARQL aufbauen.

Durch die Entwicklung dieser Informationsmodelle durch verschiedene Domänen- und Fachexperten können gleiche Informationen in verschiedenen Informationsmodellen enthalten sein. Die Informationen müssen dabei nicht zwangsweise gleich modelliert sein. Um die Informationsmodelle gemeinsam zu nutzen, wird vielfach ein Mapping genutzt. Beim Mapping werden einzelne Elemente eines Informationsmodells mit Elementen eines anderen Informationsmodells in Relation gesetzt. Auf syntaktischer Ebene werden in der Regel die Datenmodelle ineinander transformiert. Dies kann durch den Anwender von Hand geschehen, mit Hilfe von Modelltransformation (semi-) automatisch oder mit Methoden der

künstlichen Intelligenz. Ein Beispiel sind die Arbeiten von Christiansen [75, 76], in denen verschiedene Teil-Topologie-Modelle in ein übergeordnetes größeres Topologie-Modell zusammengeführt und anschließend je nach Anwendungsfall wieder in Teilmodelle zerlegt werden. In [77] wird das Konzept der semantischen Transformation für Geodaten dargestellt. Im Bereich der künstlichen Intelligenz gibt es ebenfalls erste Ansätze, wie z. B. mit Hilfe von natürlicher Sprachverarbeitung (Natural Language Processing (NLP)) [16, 17]. Bei Ontologien wird das Mapping durch Festlegung von OWL Relationen realisiert. So können z. B. Elemente als *gleich* oder *ist SubType von* bezeichnet werden, sodass die einzelnen Netze zusammengeführt werden können. Dies kann bei großen Netzwerken jedoch sehr komplex werden. Ein Konzept zur Erstellung von neuen Informationsmodellen basierend auf existierenden Informationsmodellen wird nur sehr selten betrachtet.

Aufgrund dessen wird in dieser Arbeit eine Lösung für die semantische Interoperabilität aufgezeigt, die zum Ziel hat, angeforderte Informationsmodelle aus existierenden Informationsmodellen zu erstellen. Für ein besseres Verständnis wird das Konzept der Verwaltungsschale als Anwendungsbeispiel genutzt. Die Verwaltungsschale verwendet zur Darstellung der Semantik sog. Teilmodelle (vgl. Abschnitt 6.3.2), die Informationsmodelle basierend auf einem festgelegten Metamodell darstellen. Diese werden in sogenannten Teilmodell-Templates spezifiziert. Für das Metamodell existieren Serialisierungsformate, sodass eine syntaktische Interoperabilität hergestellt werden kann. Bei der semantischen Interoperabilität muss neben dem syntaktischen Mapping[3] auch ein semantisches Mapping[4] erfolgen. Um das syntaktische Mapping möglichst automatisiert durchzuführen, wurde das Konzept der Modelltransformation eingeführt. In IEC 21823-4 [78] wird ebenfalls die Modelltransformation als Ansatz für die Herstellung der syntaktischen Interoperabilität verwendet. Dadurch können die Ansätze auch für die Lösung der Probleme bei der semantischen Interoperabilität förderlich sein. Daher wird das Konzept der Modelltransformation für die Herstellung der semantischen Interoperabilität zwischen verschiedenen Informationsmodellen im Zuge dieser Arbeit vorgestellt. Im nächsten Kapitel wird zunächst das Konzept der Modelltransformation näher beschrieben.

[3] Regeln für die Transformation von Elementen des einen Datenformats in Elemente des anderen Datenformats.

[4] Welches Element entspricht semantisch welchem anderen Element.

5 Modelltransformation

Aufgrund von unterschiedlichen Anforderungen werden immer mehr Modelle spezifiziert. Diese enthalten vielfach die gleichen semantischen Informationen, sind jedoch für den jeweiligen Anwendungsfall unterschiedlich modelliert. Ein Ziel ist, neue Instanzen dieser Modelle zum Teil oder vollständig aus anderen Modell-Instanzen zu erstellen, um z. B. Fehler bei der Dateneingabe zu verhindern.

Teilweise können Modelle nicht losgelöst voneinander betrachtet werden, da gegebenenfalls Änderungen an einem Modell zu Anpassungen an anderen Modellen führen. Somit ist ein automatisiertes Änderungsmanagement sinnvoll [79].

Die beiden vorgestellten Anwendungsfälle im Bereich der Nutzung von Modellen können mit Hilfe der Modelltransformation gelöst werden. Gerade im Bereich der modellgetriebenen Softwareentwicklung sind Modelltransformationen ein wichtiger Bestandteil [46, 80].

Aufgrund dessen erfolgt in diesem Kapitel zunächst eine kurze Einführung in die Begriffswelt der Modelltransformation (Abschnitt 5.1). Anschließend werden in Abschnitt 5.2 die Eigenschaften einer Modelltransformation betrachtet, bevor in Abschnitt 5.3 verschiedenen Modell-zu-Modell Transformationsansätze diskutiert werden. Abschließend werden in Abschnitt 5.4 verschiedene Arten von Transformationssprachen sowie die Erstellung einer solchen erläutert.

5.1 Begriffswelt der Modelltransformation

Eine Transformation ist die Wandlung von Form, Struktur oder Gestalt von einem Ausgangs- in einen Zielzustand, die mittels Regeln in einem Regelwerk festgelegt wird [81].

Definition 5.1 *Eine Transformation T ist eine Menge von Regeln (Regelwerk), die für beliebige x aus einer Menge X, die Quelle oder Definitionsbereich genannt wird, ein oder mehrere y aus einer Menge Y, die Ziel oder Wertebereich genannt wird, zuordnet.*

Bei der Modelltransformation werden die Mengen X und Y konkretisiert. Sie bestehen aus Modellelementen. Bei der Ausführung einer Modelltransformation wird eine Instanz eines Quellmodells in eine Instanz eines Zielmodells überführt. Als Quellmodell wird das Modell bezeichnet, aus dem die Daten gewonnen werden. Ein Zielmodell ist das Modell, welches erstellt oder in dem Änderungen vorgenommen werden. Als Grundvoraussetzung bleibt die Semantik bei einer Transformation zwischen dem Quell- und dem Zielmodell erhalten, sofern dies durch die Sprache des Zielmodells ermöglicht wird [81].

Definition 5.2 *Eine Modelltransformation T_M ist eine Menge von Regeln (Regelwerk), die für beliebige x aus einer Menge von Modellelementen X, die Quellmodelle genannt werden, ein oder mehrere y aus einer Menge von Modellelementen Y, die Zielmodelle genannt werden, unter Beibehaltung der Semantik zuordnet, sofern dies die Sprache des Zielmodells zulässt.*

Die Überführung erfolgt durch die Ausführung der definierten Transformationsregeln. Eine Transformationsregel beschreibt dabei, wie Elemente des Quellmodells in Elemente des Zielmodells transformiert werden sollen.

Definition 5.3 *Eine Transformationsregel definiert, wie ein oder mehrere Elemente des Quellmodells in ein oder mehrere Elemente des Zielmodells transformiert werden.*

Zusätzlich werden die nutzbaren Sprachelemente in einer Transformationssprache (vgl. 5.4) definiert.

Definition 5.4 *Eine Transformationssprache definiert die nutzbaren Sprachelemente zur Beschreibung von Transformationsregeln.*

Transformationsregeln werden in einer Transformationsdefinition zusammengefasst, die anschließend von einem Transformationssystem ausgeführt werden [81].

Definition 5.5 *Eine Transformationsdefinition ist eine Menge von Transformationsregeln, die beschreiben, wie ein Quellmodell in ein Zielmodell transformiert wird.*

Für die Ausführung der Transformationsdefinition wird ein Kontrollalgorithmus benötigt. Dieser Algorithmus führt die Auswahl und die Anwendung geeigneter Regeln innerhalb der Transformationsdefinition aus. Ein Transformationssystem übernimmt diese Aufgaben. Es liest die benötigten Quellmodelle ein, wendet die Regeln innerhalb der Transformationsdefinition an und erstellt die Zielmodelle.

Definition 5.6 *Ein Transformationssystem ist eine Applikation, die ein oder mehrere Quellmodelle einliest, die Transformationsdefinition ausführt und damit die Zielmodelle erstellt.*

Die Beziehungen zwischen Transformationsdefinition, Transformationssystem, Modell, Metamodell und Sprache sind in Abbildung 5.1 dargestellt.

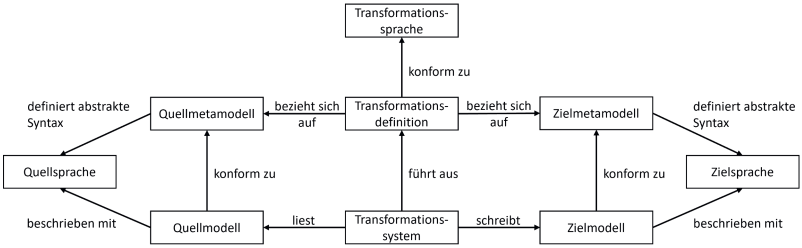

Abbildung 5.1: Beziehungen zwischen Transformation, Modell, Metamodell und Sprache nach [82, 83]

5.2 Merkmale von Modelltransformationen

Für einen Vergleich von Modelltransformationen werden Unterscheidungsmerkmale benötigt. In [84–86] wurden diverse Modelltransformationen hinsichtlich potenzieller Unterscheidungsmerkmale untersucht. Das Ergebnis ist eine heterogene Menge an Unterscheidungsmerkmalen. Diese wurden im Rahmen der Vorveröffentlichung [21] in ihrer Gesamtheit analysiert, geordnet und schließlich in vier Kategorien zusammengefasst: „Allgemeine Merkmale“, „Quell- und Ziel(meta)modelle“, „Transformationsregeln“ und „Regelnutzung“. Diese vier Kategorien inklusive ihrer Merkmale sind in Abbildung 5.2 dargestellt. Die folgenden Abschnitte beschreiben jede der vier Kategorien und ihre Merkmale im Detail.

5.2.1 Allgemeine Merkmale

Allgemein kann bei einer Modelltransformation unterschieden werden, welcher Beschreibungsmechanismus innerhalb einer Transformationsdefinition genutzt, welche Transformationsrichtung unterstützt wird und welche Inkrementalität vorliegt. Diese Merkmale werden in der ersten Kategorie zusammengefasst.

Beschreibungsmechanismus

Modelltransformationen können hinsichtlich ihres verwendeten Mechanismus in *deklarative* und *operationale/imperative* Ansätze unterschieden werden.

Bei einem *deklarativen* Ansatz wird beschrieben, wie der Start- und wie der Endzustand auszusehen haben. Es wird dabei nicht vorgeschrieben, wie der Endzustand erreicht wird. Die Vorteile sind, dass die Navigation innerhalb des Modells, das Anlegen von neuen Modellelementen und die Reihenfolge der Regelausführung nicht durch den Regelersteller erfolgt. Daher sind die Regeln schneller zu formulieren und einfacher zu verstehen. Mit geringem Aufwand kann zusätzlich durch ein Transformationssystem auch eine bidirektionale Transformation durchgeführt werden. Typische deklarative Sprachen sind LISP, ML, Haskell oder PROLOG.

Bei einem *operationalen/imperativen* Ansatz wird beschrieben, wie das Endergebnis zu erreichen ist. Es werden die einzelnen, durchzuführenden Aktionen formuliert. Der Vorteil

Abbildung 5.2: Merkmale der Modelltransformation

ist eine verbesserte Lesbarkeit, wenn z. B. Variablen zur Laufzeit verändert oder Schleifen verwendet werden sollen. Die Nachteile sind, dass diese Ansätze in der Regel im Vergleich zu deklarativen Ansätzen länger und nicht reversibel sind, da das Quellmodell nicht aus dem Zielmodell generierbar ist. Typische imperative Sprachen sind C, JAVA oder Python.

Zusätzlich existieren auch *hybride* Ansätze, die deklarative Sprachelemente für das Erstellen, Setzen, Löschen und Lesen von Modellelementen nutzen und imperative Sprachelemente für die Ablaufsteuerung, wie z. B. Vergleiche, Schleifen oder Variablen.

Transformationsrichtung

Modelltransformationen können unidirektional oder bidirektional erfolgen. Bei einer unidirektionalen Transformation erfolgt die Transformation in eine Richtung, meistens vom Quell- zum Zielmodell. Eine Ausführung in die andere Richtung ist nicht möglich. Bei bidirektionalen Transformationen können die Transformationen in beide Richtungen erfolgen. Dies kann entweder erreicht werden, indem nur bidirektionale Transformationsregeln verwendet werden (z. B. durch den Einsatz eines deklarativen Beschreibungsmechanismus)

oder indem Regelpaare definiert werden, die einmal eine Transformation vom Quell- zum Zielmodell und einmal vom Ziel- zum Quellmodell beschreiben.

Inkrementalität

Die Inkrementalität beschreibt, wie Änderungen in bestehenden Zielmodellen vorgenommen werden dürfen. Sie definiert was wann und wie geändert werden darf. Es werden drei Arten unterschieden:

- Ziel-Inkrementalität: Bei der ersten Erstellung wird alles neu angelegt. Danach folgt die Speicherung. Bei Änderungen in den Quellmodellen werden die entsprechenden Regeln ausgeführt. Nicht betroffene Elemente im Zielmodell bleiben erhalten.
- Quell-Inkrementalität: Das Transformationssystem analysiert, welche Regeln von einer Änderung im Quellmodell betroffen sind und führt diese aus. Dies ist vor allem bei großen Quellmodellen interessant.
- Erhaltung von Benutzer-Editierungen im Zielmodell: Bei einem Update des Zielmodells bleiben Änderungen, die zuvor durch einen Benutzer erfolgten, unberührt.

5.2.2 Merkmale der Quell- und Ziel-(meta-)modelle

Des Weiteren können Transformationen hinsichtlich der Quell- und Zielmodelle und deren Metamodellen unterschieden werden. Zu dieser Kategorie zählen Art, Anzahl, die Abstraktionsebene der Modelle und in welcher Beziehung die Modelle zueinanderstehen.

Art der Quell- und Zielmodelle

Bei einer Modelltransformation können zwei Arten von Zielmodellen erzeugt werden: Textartefakte oder Modellartefakte. Die Generierung von Text wird als Modell-zu-Text Transformation und die Generierung von Modellen als Modell-zu-Modell Transformation bezeichnet [87]. Bei den Modellen kann zusätzlich nach der internen Struktur unterschieden werden, z. B. ob diese baumartig ist oder eine Graphstruktur aufweist. Für beide Transformationen sind verschiedene Ansätze entwickelt worden. Eine detaillierte Beschreibung der Modell-zu-Modell Transformationsansätzen erfolgt in Abschnitt 5.3.

Anzahl der Quell- und Zielmodelle

Neben der Art des Zielmodells kann nach der Anzahl der Quell- und Zielmodelle unterschieden werden. Insgesamt existieren vier Kombinationen:

1:1 Modelltransformation: Ein Quellmodell wird in genau ein Zielmodell transformiert. Beispiel: Ein Modell nach dem ACPLT/KS-Metamodell wird in ein Modell nach dem OPC UA Metamodell umgewandelt.

1:M Modelltransformation: Ein Quellmodell wird in mehrere Zielmodelle transformiert. Beispiel: Ein Plattform-unabhängiges Modell wird in verschiedene Plattform-abhängige Modelle transformiert.

M:1 Modelltransformation: Mehrere Quellmodelle werden in ein Zielmodell transformiert. Beispiel: Mehrere Quellmodelle, die unterschiedlich entwickelt wurden, werden in einem Zielmodell kombiniert oder zusammengeführt.

M:N Modelltransformation: Mehrere Quellmodelle werden in mehrere Zielmodelle transformiert. Beispiel: Mehrere Modelle werden untereinander synchronisiert.

Abstraktionsebene der Quell- und Zielmodelle

Die Quell- und Zielmodelle können in verschiedenen Abstraktionsebenen der Originale liegen. Aus diesem Grund kann in eine Transformation zwischen Modellen, die auf einer Abstraktionsebene liegen, und in eine Transformation zwischen Modellen, die auf unterschiedlichen Abstraktionsebenen liegen, unterschieden werden. Die erste wird als horizontale Transformation und die zweite als vertikale Transformation bezeichnet. Beispiele für eine horizontale Transformation sind die Restrukturierung und die Migration. Die Code-Generierung ist ein Beispiel einer vertikalen Transformation.

Beziehung zwischen Quell- und Zielmodell

Einige Ansätze befassen sich mit der Erstellung von neuen Zielmodellen ohne dabei die Quellmodelle zu verändern. Andere Ansätze ändern als Ergebnis die Quellmodelle. Im zweitgenannten Fall, wenn das Zielmodell dem Quellmodell entspricht, wird dies als eine In-Place-Transformation bezeichnet. Dabei kann weiterhin unterschieden werden, ob ein reines Ersetzen oder auch ein Update vorliegt. Bei einem Update muss festgelegt werden, welche Modellelemente geupdatet werden dürfen, z. B. nur neue Modellelemente oder nur bestimmte Klassen von Modellelementen.

Art der Metamodelle

Neben den Quell- und Zielmodellen kann auch die Art der Metamodelle, nach denen diese formuliert sind, betrachtet werden. Es wird in endogene und exogene Transformationen unterschieden. Bei einer endogenen Transformation sind die Metamodelle der Quell- und Zielmodelle gleich. Sind die Metamodelle verschiedenen, liegt eine exogene Transformation vor. In [88] wird eine endogene Transformation zusätzlich als Rephrasing und eine exogene Transformation als Translation bezeichnet. Typische Beispiele für eine endogene Transformation sind die Optimierung, die Restrukturierung, die Vereinfachung oder die Normalisierung des Quellmodells. Die Synthese, das Reverse Engineering oder die Migration sind hingegen klassische Beispiele einer exogenen Transformation.

5.2.3 Merkmale der Transformationsregeln

Viele der vorher genannten Merkmale für Modelltransformationen gelten auch explizit für Transformationsregeln, wie z. B. die Art der Metamodelle, der Beschreibungsmechanismus oder die Transformationsrichtung. Zusätzlich existieren weitere Merkmale, die speziell für Transformationsregeln gelten. Die Merkmale dieser Kategorie werden in diesem Abschnitt vorgestellt.

Genutzte Konstrukte

Transformationsregeln können hinsichtlich der genutzten Konstrukte - Variablen, Muster und Logik - unterschieden werden.

Eine Möglichkeit ist die Nutzung von Variablen, um Werte von Elementen zu speichern und wiederzuverwenden. Dies bedingt einen imperativen Beschreibungsmechanismus. Die Variablen werden in der Regel als Metavariablen bezeichnet, um sie von den Element-Variablen bzw. Element-Attributen, die Teile der zu transformierenden Modelle sind, zu unterscheiden.

Zusätzlich können Muster verwendet werden. Muster sind Modellfragmente und bestehen aus beliebig vielen Variablen. Zudem können auch Ausdrücke und Aussagen der Metasprache verwendet werden. Dies können sowohl als eine Zeichenkette, als ein Begriff oder als ein Graph dargestellt werden.

Mit Hilfe von Logik können Berechnungen oder Beschränkungen ausgedrückt werden. Diese können verschiedenen Paradigmen folgen und in verschiedenen Beschreibungsmechanismen genutzt werden. Zusätzlich kann die Logik ausführbar, z. B. mit Hilfe von OCL-Abfragen, oder nicht ausführbar sein, z. B. durch die Festlegung von Einschränkungen.

Die Konstrukte können ferner in *untypisiert*, *syntaktisch typisiert* und *semantisch typisiert* unterschieden werden. Untypisierte Konstrukte sind z. B. textuelle Templates. Variablen, die einem Metamodell-Element zugeordnet werden und Elemente dieser Metamodell-Element-Klasse verwalten können, werden syntaktisch typisiert genannt. Semantisch typisierte Konstrukte erlauben stärkere Eigenschaften, wie z. B. die Wohlgeformtheit (statische Semantik) oder die Definition des Verhaltens (dynamische Semantik).

Syntaktische Separation

Eine syntaktische Separation liegt vor, wenn eine Trennung der Transformationsregeln bezüglich der Modelle existiert. Zugehörige Ansätze ermöglichen die Definition von Regeln, die nur auf dem Quell-Modell oder nur auf dem Ziel-Modell anzuwenden sind. Klassische Vertreter sind z. B. die Left-Hand-Side (LHS-) und die Right-Hand-Side (RHS-)Regeln. Die LHS-Regeln werden nur auf die Quell-Modelle und die RHS-Regeln nur auf die Ziel-Modelle angewendet.

Ausführungsbedingung

Einige Regeln können Ausführungsbedingungen enthalten. Erst wenn diese erfüllt sind, werden die Regeln ausgeführt. Je nach Art der Ausführungsbedingungen kann das Scheduling der Regelanwendung unterschiedlich ausfallen (siehe Abschnitt 5.2.4). Ein Beispiel sind when-Bedingungen.

Parametrierung

Transformationsregeln können parametrierbar sein. Dies bedeutet, dass generische Regeln definiert werden können, die mit Hilfe von Parametern konkretisiert werden. Beispielsweise könnten innerhalb einer Regel mehrere Alternativen implementiert sein und mit Hilfe eines Parameters ausgewählt werden. Andere Optionen zur Spezialisierung einer Regel sind z. B. die Übergabe von Datentypen oder Modelltypen.

5.2.4 Merkmale der Regelnutzung

Abschließen wird die Kategorie Regelnutzung betrachtet. Darunter werden die Merkmale *Steuerung der Regelanwendung* und *Regelorganisation* zusammengefasst. Die Steuerung der Regelanwendung beantwortet die Fragen bezüglich des Zeitpunkts und auf welche Bereiche die Regeln ausgeführt werden sollen. Die Regelorganisation beschreibt, wie die Regeln geordnet abgelegt werden können.

Steuerung der Regelanwendung

Die Steuerung der Regelanwendung wird in Standortbestimmung und Scheduling untergliedert. Die Standortbestimmung beschreibt auf welche Bereiche im Quell- und Zielmodell die Regeln angewendet werden sollen. Dies ist vor allem interessant, wenn eine Regel auf mehrere Bereiche bzw. Elemente anwendbar ist. Wird diese Regel auf alle ausgeführt oder nur auf einzelne und wenn ja, auf welches Element, z. B. das erste gefundene oder das am tiefsten gefundene. Je nachdem wie die Standortbestimmung festgelegt wird, kann das Verhalten in *deterministisch*, *nicht deterministisch* und *interaktiv* unterschieden werden.

Das Scheduling definiert zu welchem Zeitpunkt die Regeln ausgeführt werden sollen und kann in vier Untermerkmale unterschieden werden:

- *Erstellung von Steuerung der Regelanwendung:* Der Ersteller der Transformationsdefinition kann entweder den Planungsalgorithmus, also die Reihenfolge der Regelausführung, selbst definieren (explizites Scheduling) oder diesen aktiv beeinflussen, z. B. durch die Vorgabe das Regeln der Auslöser für andere Regeln sein können (internes explizites Scheduling). Alternativ wird der Algorithmus vom Transformationstool vorgegeben (internes Scheduling).

- *Regelauswahl:* Die Regelauswahl kann entweder durch explizite Bedingungen, durch eine nicht deterministische Auswahl, durch einen Konfliktlösungsalgorithmus oder interaktiv durch den Benutzer erfolgen. Bedingungen (sog. Application Constraints) können sowohl negativ einschränkend, Negative Application Constraint (NAC), als auch positiv einschränkend, Positive Application Constraint (PAC), formuliert sein. Bei NACs werden Elemente festgelegt, die nicht in der gesuchten Quellmodellstruktur vorkommen dürfen. Ist dies der Fall, wird die Regel ausgeführt. Die PACs hingegen beschreiben Elemente, die zwingend in der gesuchten Quellmodellstruktur vorkommen müssen, damit die Regel ausgeführt wird. Zusätzlich existieren auch einfachere Bedingungen, wie z. B. Prioritäten.

- *Regelwiederholung:* In einigen Anwendungsfällen sollen die Regeln oder der gesamte Regelsatz mehrfach ausgeführt werden. Dadurch ergeben sich ggf. neue Eingangsbedingungen (vor allem bei In-Place-Transformationen (vgl. Abschnitt 5.2.2)). Dafür können Schleifen, Rekursionen oder Fixpunktiterationen genutzt werden.

- *Unterstützung von Phasen:* Dieses Unterscheidungsmerkmal liegt vor, wenn eine Transformation in verschiedene Phasen unterteilt wird. In jeder Phase können ausgewählte Regeln ausgeführt werden. Beispielsweise könnte eine Phase zum Erstellen und eine Phase zum Setzen von Werten definiert werden.

Regelorganisation

Die Regelorganisation beschreibt, wie Regeln geordnet abgelegt werden, um diese wiederzuverwenden. Eine Übersicht der Wiederverwendungsmechanismen wird in [89] aufgeführt. Die Regelorganisation wird in drei Arten unterschieden:

- *Modularisierung:* Regeln können in Modulen zusammengefasst werden. Einzelne Module können wiederverwendet werden. Sowohl in der aktuellen als auch in andere Transformationsdefinitionen wird dies ermöglicht, indem die Module importiert werden.
- *Wiederverwendung:* Regeln können mit Hilfe zuvor definierter Regeln erstellt werden. Dies kann z. B. durch das Zusammensetzen bestehender Regeln erfolgen oder durch Prinzipien der Vererbung, wie die Erweiterung oder die Spezialisierung.
- *Strukturierung:* Regeln können in die Elemente des Quellmodells, des Zielmodells oder frei organisiert sein. Das bedeutet, dass Regeln z. B. bestimmten Metaklassen zugeordnet werden. Dies ist beim Quellmodell schwierig, sofern mehrere Metaklassen vorhanden sind.

5.3 Modell-zu-Modell Transformationsansätze

In Abschnitt 5.2.2 wurde eine Unterscheidung in Modell-zu-Text und Modell-zu-Modell Transformationen eingeführt. Diese Unterscheidung basiert auf der Forschung von Czarnecki und Helsen [84], die verschiedene Transformationssysteme untersucht und die verwendeten Ansätze kategorisiert haben. In [87] wurde bestätigt, dass diese Kategorisierung derzeit noch relevant ist.

Für die Modell-zu-Text Transformation existieren zwei Ansätze: Besucher- und Vorlagen-Ansatz. Für die Modell-zu-Modell Transformation existieren verschiedene Ansätze, wobei folgende vier Ansätze in der Regel verwendet werden: der imperative/operationale, der relationale/deklarative, der Graph-basierte und der hybride Ansatz. Da in dieser Arbeit eine Modell-zu-Modell Transformation durchgeführt werden soll, werden in diesem Abschnitt die Ansätze für die Modell-zu-Modell Transformation detailliert vorgestellt und diskutiert.

5.3.1 Imperativer/Operationaler Ansatz

Der imperative/operationale Ansatz basiert auf einer imperativen Sprache und adressiert, *wie* und *wann* eine Transformation ausgeführt werden soll. Dazu erstellt der Anwender eine Liste von Anweisungen bzw. Regeln, die anschließend auf die Modelle angewendet werden. Die imperativen Sprachkonstrukte sind mit den Programmiersprachen vergleichbar, erweitern diese aber um spezielle Funktionalitäten, die für Modelltransformationen benötigt werden, wie z. B. die Rückverfolgung. Ein Beispiel ist die Kombination einer Abfrage-Sprache, wie z. B. OCL, mit imperativen Konstrukten einer Programmiersprache. Dabei

können die Extra-Funktionen über Bibliotheken zur Verfügung gestellt werden. Des Weiteren hat die OMG einen solchen Ansatz in Form des QVT Operational Mappings entwickelt und veröffentlicht. Ein guter Einblick ist in [90] gegeben. 16 weitere Umsetzungen, wie z. B. der von Kermeta [91], sind in [87] aufgelistet.

Ein Sonderfall des operationalen Ansatzes stellt der *Direkte-Manipulations-Ansatz* dar. Bei diesem Ansatz liegt eine interne Modellrepräsentation vor, auf die mittels eines imperativen Application Programming Interface (API) zugegriffen werden kann. Mit Hilfe dieser Schnittstelle kann das Modell schrittweise verändert werden. Das Framework, welches diese API anbietet, ist in der Regel objektorientiert implementiert. Typische imperative Sprachen, die verwendet werden, sind JAVA, C++ oder Python. Der Nutzer formuliert die Regeln in Form von Anweisungen (Aufrufen von API-Operationen), um das Modell zu verändern. Er ist aber auch zusätzlich für die richtige Reihenfolge und die korrekte Ausführung der Regeln/Aufrufe verantwortlich.

Vorteile des imperativen/operationalen Ansatzes sind die sehr gute Performance zur Laufzeit, die Programmiersprachen mitbringen, sowie die komplette Kontrolle über die Durchführung der Transformationsschritte. Dies ermöglicht unter anderem die einfache Integration von User-spezifischen Erweiterungen, wie z. B. ein spezielles Logging oder eine spezielle Rückverfolgungsstechnik. Für die einen ist es ein Vorteil, für die anderen ein Nachteil. So gehört die Kenntnis der Sprachelemente und der Aufwand zur Kontrolle für einen Software-Entwickler zum Arbeitsalltag, für einen Domänenexperten als reinen Anwender von Modellen, meistens nicht. Ein Domänenexperte möchte auf der Semantikebene möglichst schnell einfach zu verstehende Regeln formulieren.

5.3.2 Relationaler/Deklarativer Ansatz

Im Gegensatz zum imperativen Ansatz wird beim relationalen/deklarativen Ansatz nicht festgelegt, wie ein Objekt transformiert wird, sondern nur wie Quell- und Zielobjekt aussehen. Eine Definition, wie das System diese Transformation umsetzt, erfolgt nicht. Das Hauptkonzept dieses Ansatzes sind mathematische Relationen. Die Grundidee basiert auf der Definition von Relationen zwischen Elementen im Quell- und Zielmodell mit Hilfe von Constraints in Form von deklarativen Regeln (z. B. [92]). Eine Regel definiert, welches Element bzw. Muster im Quellmodell gefunden werden muss und wie dieses danach im Zielmodell auszusehen hat. Zusätzlich legt die Regel fest, wann sie ausgeführt werden soll.

Vorteile dieses Ansatzes sind die einfache Definition von Regeln, die Vermeidung von Seiteneffekten bei der Ausführung und direkte Erstellung der Zielelemente. Aufgrund der deklarativen Beschreibungen sind die Regeln zunächst nicht ohne weiteres ausführbar. Dazu sind logische Programmiersprachen oder operationale Ansätze notwendig.

Es existieren bereits etablierte Umsetzungen wie die QVT Relations der OMG [93]. Eine größere Auflistung befindet sich unter anderem in [84, 87].

5.3.3 Graph-basierter Ansatz

Viele Meta-Modelle werden typischerweise in UML-Notation erstellt und können damit als Graphen angesehen werden. Aus diesem Grund verwenden einige Ansätze für die Modelltransformation eine Graph-Grammatik. Dabei werden die Modelle als typisierte Graphen betrachtet. Transformationsregeln bestehen bei diesem Ansatz aus einer linken (LHS) und einer rechten Seite (RHS). Die LHS beschreibt, welche Quellmodellelemente ersetzt, und die RHS, durch welche Elemente diese Quellmodellelemente ersetzt werden sollen. Bei der Ausführung einer Regel wird der Ausgangsgraph (Quellmodell) nach einer Übereinstimmung der LHS durchsucht. Wird eine Übereinstimmung gefunden, wird der Graph durch die RHS ersetzt. Dies erfolgt so lange, bis keine Übereinstimmungen mehr gefunden werden.

Vorteile sind, dass der Ansatz auf einer theoretischen Grundlage aufbaut, eine In-Place-Transformation ermöglicht und die Transformationen meistens unidirektional sind. Der Ansatz ist nicht deterministisch und wird daher in der Praxis selten angewendet.

Um M:N Transformationen, anstelle von In-Place-Transformationen, zu ermöglichen, wurde von Schürr die Triple Graph Grammatik (TGG) [94] entwickelt. Er führte einen dritten simultan erzeugten Korrespondenzgraph ein und verwendet kontextsensitive Graph-Grammatiken. Ein Überblick über die Entwicklungen der TGGs wird in [95] gegeben.

Es gibt bereits mehrere Umsetzungen dieses Ansatzes, z. B. AGG [96] oder UMLX [97]. Diese Umsetzungen waren einer der ersten Realisierungen und werden bis heute genutzt. Des Weiteren existieren in der Automatisierungstechnik erste Ansätze [79, 98]. Weitere Umsetzungen sind in [84, 87] untersucht worden.

5.3.4 Hybrider Ansatz

Die zuvor vorgestellten Ansätze haben Vor- und Nachteile. Vielfach sind diese komplementär. So ermöglichen imperative Ansätze eine sehr effiziente Implementierung für komplexe Transformationen, die jedoch mit größeren Transformationsdefinitionen einhergehen. Dies kann für einen Ersteller und Nutzer zu Problemen bei der Lesbarkeit und Wartbarkeit führen. Relationale Ansätze sind präziser und einfacher zu verstehen, da die Definition der Transformation auf einer höheren Ebene mit weniger Implementierungsdetail erfolgt. Allerdings können Einschränkungen bezüglich der expliziten Definition des Kontrollflusses, vor allem bei komplexeren Transformationen, vorliegen. Der Graph-basierte Ansatz basiert zwar auf einer theoretischen Grundlage, ist aber nicht deterministisch. Aus diesen Gründen werden hybride Ansätze entwickelt, die die Vorteile der verschiedenen Ansätze kombinieren. Diese können in zwei Kategorien unterschieden werden:

- Systeme, die mehrere Ansätze unterstützen, aber diese auf Regelebene trennen
- Systeme, die mehrere Ansätze unterstützen und diese innerhalb von Regeln ermöglichen

Zur ersten Kategorie gehört z. B. QVT. QVT unterstützt drei verschiedenen Ansätze: Relations, Operational mappings und Core. Die drei verschiedenen Ansätze können nicht innerhalb einer Regel genutzt werden. Eine Kombination von Regeln, die jeweils einen anderen der drei Ansätze nutzen, ist möglich. Atlas Transformation Language (ATL), Model Transformation Language (MOLA) und Visual Automated Model Transformations (VIATRA) sind Beispiele der zweiten Kategorie. Bei ATL wird eine Kombination des relationalen und des imperativen Ansatzes [99], bei MOLA die Kombination des Graph-basierten und des imperativen Ansatzes [100] und bei VIATRA die Kombination aus relationalem, imperativem und Graph-basiertem Ansatz verwendet [101].

5.4 Transformationssprache und -system

Zur Definition der Transformationsregeln und der Regelsteuerung, wird eine Sprache, die Transformationssprache, benötigt. Um die Transformationssprache interpretieren und damit Modelltransformationen auszuführen zu können, wird ein passendes Transformationssystem vorausgesetzt. Die Transformationssprache und das Transformationssystem bedingen einander und müssen somit immer gemeinsam entwickelt werden.

Für die Entwicklung von Transformationssprachen wurden zunächst universelle Programmiersprachen, wie C++, Java oder Python genutzt. Diese haben den Nachteil, dass sie nicht explizit für diesen Anwendungsfall entworfen wurden. Das hat zur Folge, dass regelmäßig auftretenden Aufgaben nur umständlich umgesetzt werden können [46], da ein tiefes Verständnis der Programmiersprache notwendig ist [83]. Aus diesem Grund werden konkrete Sprachen für Modelltransformationen entwickelt. Je nach verwendetem Ansatz unterstützen diese deklarative oder imperative Sprachelemente. Zusätzlich können die Sprachen hinsichtlich ihrer Verwendung in generische und domänenspezifische Transformationssprachen unterschieden werden.

5.4.1 Generische und domänenspezifische Transformationssprachen

Analog zu Modellsprachen werden auch Transformationssprachen in generische und domänenspezifische Sprachen unterschieden. Generische Transformationssprachen, häufig auch General Purpose Transformation Language (GPTL) genannt, ermöglichen die Erstellung von Transformationsregeln zwischen beliebigen Quell- und Ziel-Metamodellen und für beliebige Anwendungsfälle. Sie sind dadurch vielfältig anwendbar. Um dies zu erreichen, sind die Sprachelemente allgemein gehalten, wenig typisiert und enthalten somit nur in einem geringen Maße eine Semantik. Das hat zur Folge, dass die Erstellung von Regeln nur durch ein tiefes Verständnis der Sprache ermöglicht wird und meistens von entsprechendem Fachpersonal erfolgt. Typische Beispiele sind QVT [90, 93], ATL [99], Epsilon Transformation Language (ETL) [102] oder VIATRA [101]. Sollen hingegen Domänenexperten Transformationsregeln erstellen, ist diese Art der Sprachen nur begrenzt nutzbar.

Um diesem Problem zu entgegnen, wurden domänenspezifische Transformationssprachen (Domain Specific Transformation Language (DSTL)) entwickelt [103–105]. Diese enthalten

Sprachelemente, die an die konkrete Syntax der jeweiligen domänenspezifischen Modellierungssprache (Metamodell) angepasst werden [106]. Dadurch müssen Domänenexperten nur die Syntax der zusätzlichen Transformationssprachelemente erlernen und können diese somit schneller nutzen. Im Gegenzug können diese Sprachen dafür ausschließlich für die Modellsprache genutzt werden, für die sie entwickelt wurden.

Zusätzlich besteht die Option, die Transformationssprache an den jeweiligen Anwendungsfall anzupassen, sodass die Sprache deutlich spezifischer definiert werden kann. Dies führt dazu, dass die Sprache durch den Wegfall von nicht-benötigten Funktionen vereinfacht wird. In den letzten Jahren wurden vermehrt domänenspezifische Transformationssprachen entwickelt [107]. Immer mehr Arbeiten zeigen die Notwendigkeit auf [108, 109]. Tools zur Erstellung von domänenspezifischen Transformationssprachen [106, 110, 111] wurden entwickelt. Eine ausführliche Analyse der Literatur wird in [112] gegeben.

Wenn beliebige Metamodelle transformiert werden und eine generische Sprache für den vorliegenden Anwendungsfall anwendbar ist, sollte eine generische Sprache genutzt werden. Erfolgen die Transformationen innerhalb einer Modellierungssprache, sollen Domänenexperten in die Lage versetzt werden, die Regeln zu erstellen, und ist die Anwendung in der Domäne hoch, sollte hingegen eine domänenspezifische Transformationssprache genutzt werden.

Aktuelle Forschungen definieren Meta-Transformationssprachen inkl. passenden generischen Transformationssysteme. Diese können mit domänenspezifischen Sprachelementen erweitert werden, sodass sie als domänenspezifische Transformationssprachen gelten und dasselbe generische Transformationssystem nutzen. Dadurch kann die Entwicklung auf die domänenspezifischen Sprachelemente reduziert werden.

5.4.2 Erstellung von Transformationssprachen

Für die Erstellung von Transformationssprachen existieren verschiedene Ansätze: Nutzung oder Anpassung einer bestehenden, Generierung einer neuen oder vollständige Neuentwicklung einer Transformationssprache. In der Vorveröffentlichung [21] wurde ein Leitfaden für die Erstellung einer Transformationssprache entwickelt. Dieser Leitfaden besteht aus drei Schritten:

- Klassifikation der durchzuführenden Modelltransformation und der Quell/Ziel-Metamodelle
- Allgemeine Anforderungen an die Modelltransformationssprache und Festlegung der notwendigen abstrakten Sprachelemente
- Entwurf der Modelltransformationssprache

Klassifikation der durchzuführenden Modelltransformation und der Quell/Ziel-Metamodelle

Zunächst werden die Eigenschaften der Modelltransformation beschrieben, um eine passende Transformationssprache auszuwählen oder zu entwickeln. Nach Vorgabe des Leitfadens ist zunächst eine Klassifikation hinsichtlich der allgemeinen Merkmale sowie der Merkmale zu den Quell- und Ziel(meta)modellen, die in Abschnitt 5.2 definiert wurden,

durchzuführen. Dies wird darin begründet, dass diese Merkmale bereits durch den gegebenen Anwendungsfall klassifizierbar sind. Die Merkmale für die Transformationsregeln und für die Regelnutzung formulieren Anforderungen an die Funktionalitäten der Sprache und werden daher im zweiten Schritt spezifiziert. Zusätzlich soll das Meta-Meta-Modell der Quell- und Zielmodelle analysiert werden, da die Transformationssprache die Typen der Modellelemente unterstützen muss.

Allgemeine Anforderungen an die Modelltransformationssprache und Festlegung der notwendigen abstrakten Sprachelemente

In diesem Schritt werden Anforderungen an die Sprache spezifiziert, z. B. wie die Regeln formuliert werden sollen. Die in diesem Schritt ermittelten Anforderungen müssen von der Syntax der Transformationssprache erfüllt werden. Die Merkmale aus Abschnitt 5.2 hinsichtlich der Transformationsregeln und der Regelnutzung gehören in diesen Schritt. Hierzu zählen die zu unterstützenden Konstrukte, ob eine syntaktische Separation erfolgen soll, wie die Regeln ausgeführt werden sollen sowie ob und wie die Regeln wiederverwendet werden sollen. Zusätzlich werden in diesem Schritt die abstrakten Sprachelemente festgelegt, die eine Sprache unterstützen muss.

Entwurf der Modelltransformationssprache

Beim Entwurf der Modelltransformationssprache wird zunächst geprüft, ob bereits eine existierende Modelltransformationssprache verwendet werden kann, die die zuvor definierten Klassifikationen und Anforderungen erfüllt. Dies kann sowohl eine bestehende domänenspezifische als auch eine generische Transformationssprache sein. Ist dies nicht der Fall, wird geprüft, ob bei einem bestehenden Modelltransformations-Framework die Sprache und die Tools so angepasst werden können, sodass dieses den Anforderungen genügt. Dabei kann z. B. eine generische Transformationssprache genutzt und durch kleine Änderungen in der Syntax (z. B. durch Einführung von weiteren Sprachelementen oder durch Restriktion auf einen kleineren Umfang der Sprachelemente) für die geforderte Domäne angepasst werden. In [113] wird eine Anpassung einer generischen Transformationssprache gezeigt. Dies ist nur bei minimalen Anpassungen sinnvoll.

Kann weder eine existierende Sprache direkt genutzt noch angepasst werden, muss eine neue Sprache entwickelt werden. Es sollten existierende Tools für die (semi-)automatische Erstellung von Modelltransformationssprachen hinsichtlich ihrer Nutzbarkeit analysiert werden, um den Aufwand der Neuentwicklung zu reduzieren. Für einige Transformationsprobleme wurden bereits solche Frameworks erstellt. Meistens stellen diese Frameworks Anforderungen an die Metamodelle, wie z. B. dass diese in einer formalen Sprache definiert sein müssen, oder die zu erfüllenden Aufgaben müssen in einer formalen Darstellung definiert werden.

Existiert kein nutzbares Generierungs-Framework, muss eine komplette Neuentwicklung erfolgen. In [114] und [115] sind Beispielprozeduren für die vollständige Erstellung von neuen Transformationssprachen gegeben. Es ergeben sich drei Schritte für die Erstellung:

- Erstellung einer abstrakten Syntax und Definition der statischen Semantik[1],

[1]Die Bedeutung der Elemente für die Transformation sowie die Definition von Invarianten, vor allem für Typen.

- Entwurf einer zugehörigen konkreten Syntax und
- Implementierung eines Parsers, Checkers und Interpreters (auch Transformationssystem genannt).

Bei der Definition der abstrakten und konkreten Syntax sollte betrachtet werden, ob Sprachkonstrukte von anderen Sprachen, wie z. B. Transformations-, Programmier- oder Expression-Sprachen, wiederverwendet werden können. Vor allem bei der abstrakten Syntax sollte auf eine umfassende Spezifikation von Expressions geachtet werden, um nicht betrachtete Randfälle später abbilden zu können. In [111] ist eine Übersicht über typische Elemente einer Modelltransformationssprache gegeben. Für den Schritt der Implementierung existieren bereits Generatoren, die die entsprechenden Werkzeuge mit einer gegebenen Sprache automatisiert erzeugen. Diese stellen ähnlich den Generierungs-Frameworks Anforderungen an die Sprache, für die Werkzeuge erstellt werden können. In [108] wird ein Generator beschrieben, der automatisiert Code-Templates aus einer formalen Modelltransformationssprache erstellt.

6 Modellierung und Austausch von Asset-Informationen

Die Modellierung und der Austausch von Asset-Informationen haben in den letzten Jahren eine hohe Bedeutung im Bereich der Forschung an Hochschulen und in der Industrie erlangt. Dies liegt vor allem an der Einführung der Begriffe *Digitaler Zwilling* und *Verwaltungsschale*. In diesem Kontext existieren heutzutage zusätzliche Begriffe, wie *Digitaler Schatten* oder *Digitaler Engel* [116]. Im Rahmen dieser Arbeit wird keine weitere Unterscheidung zwischen den Begriffen getroffen und der Fokus liegt auf der Modellierung sowie dem Austausch von Asset-Informationen.

Erstmals wurde der Begriff des Digitalen Zwillings 2003 von Grieves [117] eingeführt. 2010 verwendete die NASA den Begriff des Digitalen Zwillings zur Bezeichnung eines Simulationsmodells eines physischen Raumfahrzeugs [118]. Anschließend entstanden unterschiedliche Definitionen, die alle eines gemeinsam haben: Sie beschreiben den Digitalen Zwilling als ein informationstechnisches Abbild einer physischen Entität. In [119] wurde eine ausführliche Literaturrecherche durchgeführt, die aufzeigt, dass vier Forschungsgebiete im Bereich des Digitalen Zwillings existieren: Die Modellierung von Digitalen Zwillingen (Informationsmodell), der Umgang mit Daten (Datenfusion), die Interaktion und Kollaboration zwischen Digitalen Zwillingen sowie die Modellierung, das Auffinden und die Integration von Diensten Digitaler Zwillinge. Die Bedeutung des Digitalen Zwillings zeigt sich durch die Technologietrends, die jedes Jahr von Gartner veröffentlicht werden [120–122]. So war laut Gartner der Digitale Zwilling von 2017 bis 2019 einer der fünf wichtigsten Technologietrends. In Deutschland wurde zeitgleich der Begriff der Verwaltungsschale [123] geprägt. Dieser erweitert den Anwendungsbereich des Begriffs des Digitalen Zwillings dahingehend, dass nicht nur physische, sondern auch virtuelle Entitäten berücksichtigt werden. Diese physischen und virtuellen Entitäten werden Assets genannt.

In [119] wird aufgezeigt, dass zwar verschiedene Ansätze für die Datenmodellierung existieren, jedoch ein Konsens zwischen diesen zu diesem Zeitpunkt nicht gegeben ist. Sie fordern generische Modellierungsmethoden für die einzelnen Forschungsgebiete. Bestehende internationale Normen wurden soweit erkennbar nicht betrachtet. Aus diesem Grund erfolgt zunächst ein Auszug aus der Vorveröffentlichung [20], in der die aktuelle Normungslandschaft im Bereich der Eigenschaftsmodellierung aufgezeigt wird. Dem interessierten Leser wird zusätzlich [124] empfohlen, in dem ebenfalls aktuelle Modellierungen, auch industrielle, für Digitale Zwillinge vorgestellt und verglichen werden.

Nach dem Kurzüberblick werden drei vielversprechende, standardisierte Ansätze für die Modellierung und den Austausch von Asset-Informationen detaillierter vorgestellt. Das gemeinsame Ziel dieser Ansätze ist der Weg von der System-/Gewerke-orientierten zu einer

Asset-orientierten Ablage der Informationen. Dieser Wandel ist vor allem in der Prozessindustrie noch nicht erfolgt [3]. Dadurch soll der Zugriff auf Asset-Informationen deutlich vereinfacht werden. Am Ende dieses Kapitels erfolgt ein Vergleich der Ansätze sowie eine Schlussfolgerung, wie die Vorteile der einzelnen Ansätze kombiniert werden können.

6.1 Aktuelle Normungslandschaft für Eigenschaften

Der folgende Abschnitt wurde in Kapitel 2 der Vorveröffentlichung [20] abgedruckt und stellt die aktuelle Normungslandschaft im Bereich der Eigenschaftenmodellierung vor. Dabei wird herausgestellt, dass bereits nationale und internationale Normen existieren, die einen Beitrag leisten.

„Die Normen können in vier Bereiche eingeteilt werden (siehe Abbildung 6.1):

1. *Normen, die sich mit der Modellierung von Eigenschaftsbeschreibungen befassen,*
2. *Normen, die Regeln und Anleitungen für die Erstellung von Eigenschaftsbeschreibungen, Klassifikationen und Lexika bereitstellen,*
3. *Normen, die Referenzmodelle (Klassifikationen und Eigenschaftsbeschreibungen) definieren und*
4. *Normen, die Informationsmodelle für den Datenaustausch von Informationen über Eigenschaften spezifizieren.*

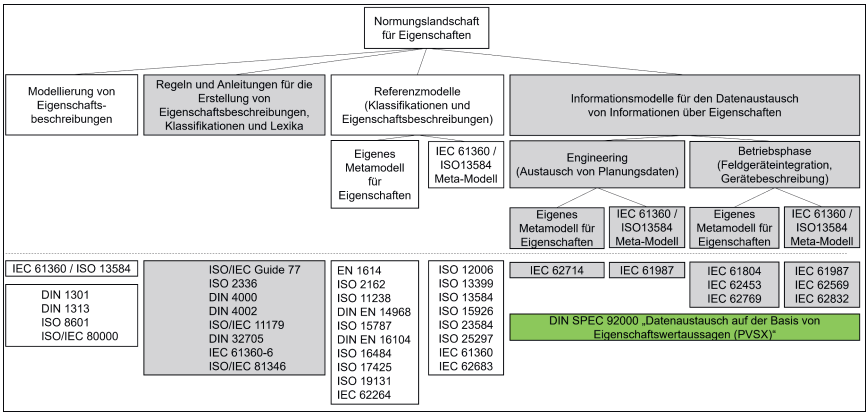

Abbildung 6.1: Normungslandschaft für Eigenschaften nach [20]

Um Eigenschaften einheitlich zu beschreiben, wird ein weltweit einheitliches Modell benötigt. Beide internationale Normungsorganisationen, IEC und ISO, entwickelten je ein solches Modell (ISO 13584-42[125] und IEC 61360-1[126]) und führten aufgrund der hohen Ähnlichkeit und der Interoperabilität diese zu einem gemeinsamen Informationsmodell

in der EXPRESS-Sprache [127] zusammen (ISO 13584-42 und IEC 61360-2 [128]), deren zugehörigen Erweiterungen ebenfalls synchronisiert werden. Um auch bei der Definition der Einheiten und des Datentyps einer Eigenschaftsbeschreibung interoperabel zu bleiben, werden die entsprechenden Normen hierfür verwendet (z. B. ISO 8601 oder ISO/IEC 80000).

Aufbauend auf dem Modell für Eigenschaftsbeschreibungen müssen Regeln und Anleitungen für die Erstellung von konkreten Eigenschaftsbeschreibungen, deren Klassifikationen und Lexika erstellt werden[1]*. Basierend auf diesen wurden Modelle für Eigenschaftsbeschreibungen definiert und in domänenspezifischen Klassifikationen (z. B. ISO 13399 oder IEC 62683) zusammengeführt. In einigen Domänen existieren bereits eigene Modelle für Eigenschaftsbeschreibungen (z. B. ISO 11238 oder IEC 62264). Diese müssen beim Informationsaustausch in das oben definierte Modell transformiert werden.*

Zum Austausch von Informationen über Eigenschaften müssen die einzelnen Eigenschaften eindeutig definiert und in standardisierten Eigenschafts-Lexika hinterlegt sein. Beispiele für existierende Eigenschafts-Lexika sind das IEC 16360 - Common Data Dictionary (IEC61360-CDD)[2] *oder ECLASS*[3]*.*

Die auszutauschenden Informationen treten in allen Phasen des Produkt-Lebenszyklus auf. Dies beginnt mit der Auswahl, der Bestellung und dem Einkauf von Produkten (Procurement) mit Hilfe von Informationen über den Produkt-Typen. Diese werden in der Engineering-Phase weiterverwendet und in der Auslieferungsphase um Instanz-Eigenschaften erweitert. Während des Betriebes stehen vor allem Parametrierungen sowie Ist-Werte der Instanzen im Fokus und in der Instandhaltung werden ebenfalls Informationen über den Typen und die jeweilige Instanz benötigt. Aus diesem Grund haben sich verschiedene Informationsmodelle für die unterschiedlichen Lebensphasen entwickelt. So wird für den Austausch von Planungsdaten von Anlagen das Informationsmodell AutomationML (IEC 62714 [129]) genutzt, während für Gerätedaten die Merkmalslisten aus der IEC 61987 [130] verwendet werden. In der Geräteintegration wird die Beschreibungssprache EDDL[4] *(IEC 61804 [131]), die Schnittstelle FDT*[5] *(IEC 62453 [132]) oder das Konzept FDI*[6] *(IEC 62769 [133]) genutzt. Für die Verwaltung von Asset-Informationen wird das Informationsmodell der IEC 62832 [134] oder zukünftig die Verwaltungsschale verwendet.“*[7]

6.2 Digital Factory Framework - International Electrotechnical Commission

Das Digital Factory Framework (DF Framework) ist ein internationaler Standard der IEC für die Modellierung von Produktionssystemen und wurde 2016 als technische Spezifikati-

[1] Die zugehörigen Normen sind in Bild 6.1 aufgelistet.
[2] `https://cdd.iec.ch/`
[3] `https://www.eclass.eu/`
[4] Electronic Device Description Language
[5] Field Device Tool
[6] Field Device Integration
[7] Kapitel 2 der Vorveröffentlichung [20].

on und 2019 als offizieller Standard veröffentlicht. Der Standard besteht aktuell aus drei Teilen. Der erste Teil [8] stellt eine allgemeine Einführung sowie einen Überblick über die Modelle und Konzepte bereit. Im zweiten Teil [9] erfolgt die detaillierte Vorstellung der einzelnen Modellelemente und in Teil 3 [10] werden die Regeln zur Nutzung dieser Elemente formuliert. Zusätzlich werden in den Anhängen der Teile Mappings zu Technologien gegeben. In dem nachfolgenden Abschnitt werden das Ziel und der Anwendungsbereich dieses Standards vorgestellt. Anschließend wird das Informationsmodell beschrieben.

6.2.1 Ziel und Anwendungsbereich

Ziel des Standards ist die Modellierung von Produktionssystemen. Hierfür wird ein Informationsmodell definiert, um Assets im Bereich von Produktionssystemen, die Beziehungen zwischen verschiedenen Assets und den Informationsfluss zwischen den Assets zu modellieren. Als Asset wird ein „physisches oder logisches Objekt, das sich im Besitz einer Organisation befindet oder unter dem Gewahrsam einer Organisation steht, und entweder einen wahrgenommenen oder tatsächlichen Wert für die Organisation hat“ verstanden [8]. Eine Rolle wird explizit als Asset ausgeschlossen. Der Standard legt zusätzlich Regeln für die Nutzung der einzelnen Modellelemente fest (z. B. für die Erstellung von Bibliotheken). Das vorgestellte Modell gilt für alle Produktionsarten (kontinuierlich, diskret und Batch), für alle Branchen des industriellen Sektors sowie für alle Phasen im Lebenszyklus von Produktionssystemen. Es soll die Möglichkeit gegeben werden, zu jeder Zeit Informationen über ein Produktionssystem hinzuzufügen, zu löschen, zu ändern oder zu erhalten. Da bereits viele Vorarbeiten im Bereich der Modellierung durch Standardisierungsgremien (z. B. ISO oder IEC), Klassifikationskonsortien (z. B. ECLASS) und Datenlieferanten entwickelt wurden, zeigt der Standard einen Weg zur Integration und Nutzung dieser Arbeiten auf und nennt diese drei Gruppen explizit als Stakeholder. Als Hauptstakeholder wird ein Unternehmen angesehen, welches Produktionssysteme besitzt. Dem Unternehmen soll mit Hilfe des Standards die Möglichkeit gegeben werden, die eigenen Produktionssysteme in der Informationswelt abzubilden (siehe Abbildung 6.2).

Explizit ausgeschlossene Anwendungsbereiche sind Gebäudekonstruktionen sowie jegliche Arten von Produkten, die auf dem Produktionssystem verarbeitet werden (z. B. Eingangsmaterial, Verbrauchsmaterial oder Endprodukte). Ebenfalls nicht im Standard enthalten sind Anforderungen oder eine Spezifikation einer softwaretechnischen Umsetzung. Vielmehr ist das Ziel, das Informationsmodell in bestehende Austauschformate oder Kommunikationsstandards zu integrieren, wie z. B. in AutomationML (IEC 62714[129]) oder OPC UA (IEC 62541[135]). Entsprechende Mappings sind im Anhang von [10] enthalten.

6.2.2 Informationsmodell

Das Informationsmodell basiert auf der objektorientierten Modellierung. Folglich existieren Modellelemente, um sowohl Instanzen als auch Typen zu modellieren (vgl. Abschnitt 2.4). Zudem wurden Elemente, um ein Begriffswörterbuch zu erstellen, definiert.

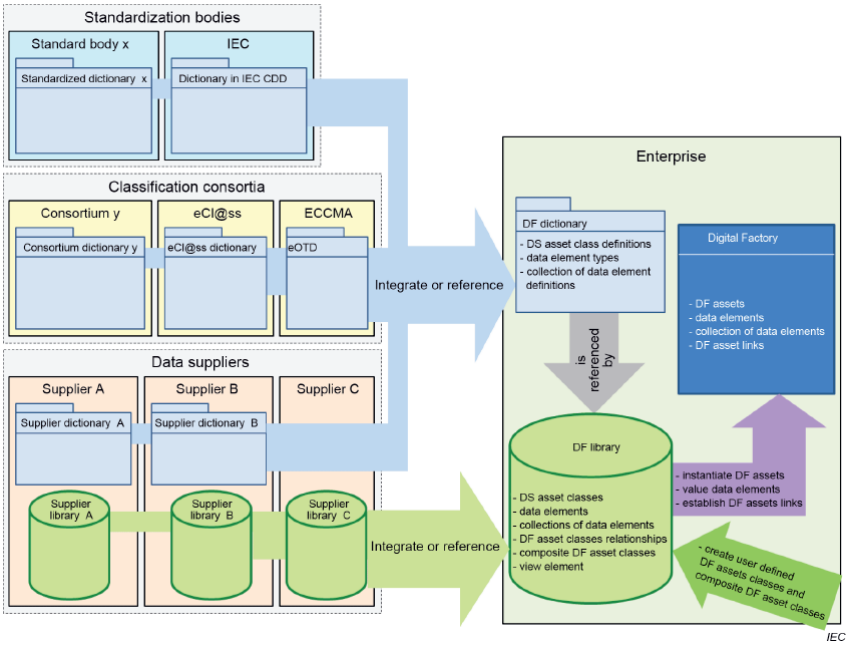

Abbildung 6.2: Überblick über das Digital Factory Framework [8]

Um konkrete Produktionssysteme und deren Teile zu modellieren, existieren insgesamt fünf Modellelemente (*DigitalFactory*, *DFasset*, *DataElement*, *CollectionOfDataElements (CDEL)* und *DFassetLink*). Abbildung 6.3 zeigt, wie ein Produktionssystem aus der realen Welt in der Informationswelt durch ein DigitalFactory-Objekt und ein PSasset durch ein DFasset-Objekt modelliert wird. Sowohl ein DigitalFactory-Objekt als auch ein DFasset-Objekt können wiederum aus mehreren DFasset-Objekten bestehen. Beide Objekttypen bestehen aus einem Header und einem Body. Im Header werden administrative Informationen, wie z. B. der Zweck des Produktionssystems oder die Informationen zur Identifikation, beschrieben. Im Body werden Informationen über die Eigenschaften, den Aufbau und die internen Beziehungen des Produktionssystems bzw. des DFasset modelliert. Dies erfolgt mit Hilfe der drei anderen Modellelemente: DataElement, CDEL und DFassetLink. Mit DataElement-Objekten können Informationen über einzelne Eigenschaften inkl. deren Werte modelliert werden, wie z. B. die Beschreibung, der Name oder der Identifier. Diese können in Listen zusammengefasst und als CDEL modelliert werden. Das letzte Element ist der DFassetLink. Mit diesem können Beziehungen zwischen zwei oder mehreren PSassets modelliert werden.

Da die objektorientierte Modellierung angewendet wird, müssen Modellelemente definiert werden, um Typen zu modellieren. Die Modellelemente *DFassetClass* und *DFassetClassAssociation* erfüllen diese Anforderung und dienen als Typen für DFasset und DFas-

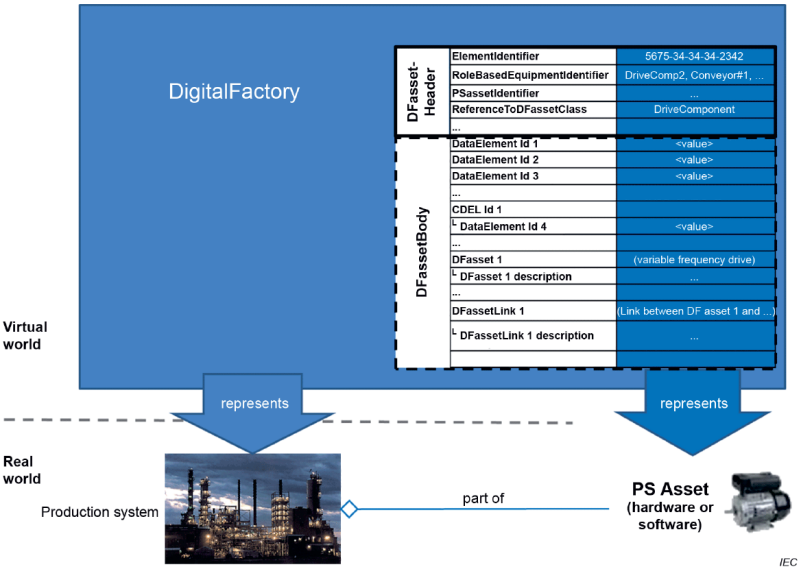

Abbildung 6.3: Modellelemente für die Darstellung von Produktionssystemen [10]

setLink. Die Verbindung zwischen dem jeweiligen Typ und der Instanz wird durch eine Referenz dargestellt. Die Typen können in Bibliotheken mit Hilfe der *Library*-Objekten zusammengefasst werden. Zusätzlich wurde das Modellelement *ViewElement* eingeführt, um die Möglichkeit von Filterung innerhalb eines Library-Objekts oder eines DigitalFactory-Objekts zu realisieren.

Um den Objektinstanzen eine semantische Bedeutung zuzuschreiben, werden Elemente für die Definition von Begriffswörterbüchern festgelegt. Dies erfolgt analog zu den bereits standardisierten Merkmalsbibliotheken (IEC 61360). Ein Begriffswörterbuch wird in diesem Standard durch ein *ConceptDictionary*-Objekt dargestellt. Wie in Abbildung 6.2 dargestellt, existieren für verschiedene Stakeholder verschiedene Begriffswörterbücher. Dies wurde in der Modellierung durch abgeleitete ConceptDictionary-Objekte berücksichtigt. Innerhalb eines Begriffswörterbuchs kann anhand der Modellelemente *DFassetClassDefinition*, *DataElementType* und *CDELdefinition* die begriffliche Festlegung dieser Konzepte getätigt werden. Somit können z. B. konkrete Asset-Beschreibungen erfolgen. Die vorher beschriebenen Objekte (z. B. DFassetClass oder DataElement) können jeweils eine Referenz auf einen dieser Begriffe angeben, um damit die Semantik des jeweiligen Objekts maschinenverarbeitbar festzulegen.

Zusammengefasst wird sowohl die objektorientierte Modellierung verwendet als auch die semantische Referenzierung im Modell angewendet. Es wurden Elemente für die Modellierung von Produktionssystemen und deren Teilen sowie die Möglichkeit der Klassifikation und Begriffsfestlegung definiert.

6.3 Asset Administration Shell - Plattform Industrie 4.0

Die Verwaltungsschale bzw. Asset Administration Shell (AAS) ist ein Konzept für die digitale Darstellung von Informationen und den herstellerübergreifenden Informationsaustausch [136] und wurde 2015 erstmals in [137] eingeführt. Das Konzept und dessen Struktur wurden in der DIN SPEC 91345 [138] 2016 als deutscher Standard veröffentlicht und international eingebracht. Eine erste funktionsfähige Modellierung sowie eine zugehörige quelloffene Referenzimplementierung wurde 2017 vom Zentralverband Elektrotechnik- und Elektronikindustrie (ZVEI) entwickelt [139]. Die Plattform Industrie 4.0 veröffentlichte 2018 ein Technologie-neutrales Informationsmodell der Verwaltungsschale in UML inklusive verschiedener Serialisierungsformate in [140]. Aktuell liegt die Version 3.0RC01 [4] vor. Weitere Arbeiten beschäftigen sich mit der Definition von Schnittstellen [5] sowie Infrastruktur-Elementen [6]. Im November 2019 wurde zusätzlich eine internationale Working Group initiiert, die eine internationale Standardisierungsreihe dieser Arbeiten als Ziel hat. Der erste Teil dieser Reihe IEC 63278-1 [141] wurde im November 2020 veröffentlicht.

6.3.1 Ziel und Anwendungsbereich

Ziel des Konzepts sind Modellierung, Zugriff und Austausch von Informationen und Funktionalitäten für ein beliebiges Asset [123]. Als Asset wird analog zum DF Framework ein „physisches oder logisches Objekt, das sich im Besitz einer Organisation befindet oder unter dem Gewahrsam einer Organisation steht, und entweder einen wahrgenommenen oder tatsächlichen Wert für die Organisation hat“ verstanden [4]. Zusätzlich wird auch eine Rolle als Asset betrachtet. Für die in [123] genannten Ziele wurde das Konzept der Verwaltungsschale eingeführt, welches die digitale Repräsentation eines Assets in der Informationswelt für eine Organisationseinheit darstellt. Die Verwaltungsschale bietet als Hauptfunktionalität einen einheitlichen Zugriff auf die Informationen und Funktionalitäten des Assets in allen Phasen des Lebenszyklus [136]. Das Konzept und die Modelle sind dennoch so generisch gehalten bzw. technologieunabhängig definiert, dass ein Asset aus einer beliebigen Branche oder Domäne stammen kann. Damit das Konzept auch für die Interaktion zwischen Maschinen angewendet werden kann, liegt der Hauptfokus auf der semantischen Annotation der Modellelemente. Es wird auf bestehenden Standards und Bibliotheken aufgebaut sowie die Möglichkeit der Erstellung von eigenen Begriffsdefinitionen gegeben. Die Verwaltungsschale ist nicht nur ein Informationsmodell. Stattdessen umfasst das Konzept auch eine vollständige Architektur, bestehend aus einem Informationsmodell, Interaktionsmodellen sowie Infrastrukturkomponenten. Für die Umsetzung wurden konkrete Serialisierungsformate in JSON (IETF RFC 8259 [142]), XML (W3C XML [143]), AutomationML (IEC 62714[129]), OPC UA (IEC 62541[135]) und RDF (W3C RDF [144]) sowie Schnittstellendefinitionen veröffentlicht. Aktuell sind das Informationsmodell und die Serialisierungsformate in [4] sowie eine generische Schnittstellendefinition in [5] standardisiert. Die anderen Konzeptteile bzw. Modelle werden aktuell in verschiedenen Standardisierungsgremien diskutiert.

6.3.2 Informationsmodell

Das Informationsmodell der Verwaltungsschale wird von einer Arbeitsgruppe der Plattform Industrie 4.0 entwickelt und liegt aktuell in der Version 3.0RC01 [4] vor. Es beschreibt die möglichen Objekttypen sowie die Beziehungen zwischen diesen. In Abbildung 6.4 ist ein Überblick über die Modellelemente gegeben.

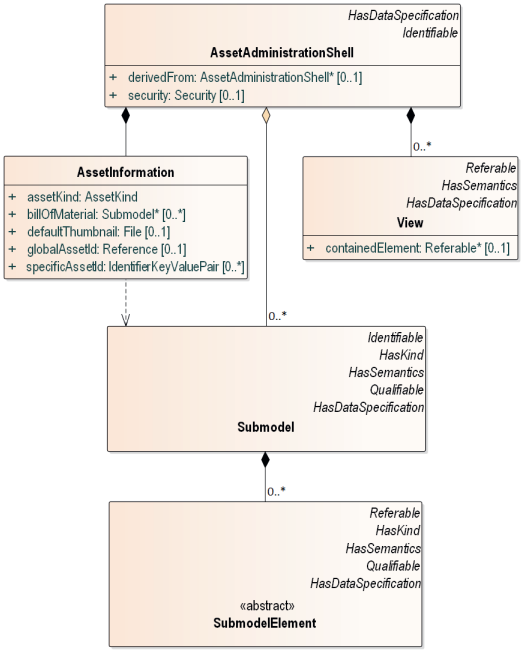

Abbildung 6.4: Überblick über das Metamodell der Verwaltungsschale nach [4]

Das *AssetAdministrationShell*-Objekt ist das Hauptobjekt und dient als Darstellung der Verwaltungsschale. Es stellt die digitale Repräsentation genau eines Assets dar und verwaltet digitale Modelle zu verschiedenen Aspekten des Assets (Teilmodelle). Aus diesem Grund besitzt jedes AssetAdminstrationShell-Objekt ein *AssetInformation*-Objekt, welches zur Darstellung der Meta-Informationen eines Assets genutzt wird, z. B. ob ein Asset-Typ oder eine Asset-Instanz vorliegt (siehe Abschnitt 2.4). Zusätzlich besitzt das AssetAdminstrationShell-Objekt Referenzen zu *Submodel*-Objekten, die zur Darstellung eines digitalen Modells des Assets dienen. Diese Objekte spiegeln jeweils einen Aspekt des Assets wider und bieten zusammenhängende Informationen innerhalb eines Modellverständnisses an, um alle benötigten Informationen für einen Anwendungsfall maschinenverarbeitbar zu machen. Ein Submodel-Objekt besteht aus *SubmodelElement*-Objekten. Diese lassen sich in verschiedene Untertypen unterscheiden und sind innerhalb des jeweiligen Submodels eindeutig identifizierbar (siehe Abbildung 6.5).

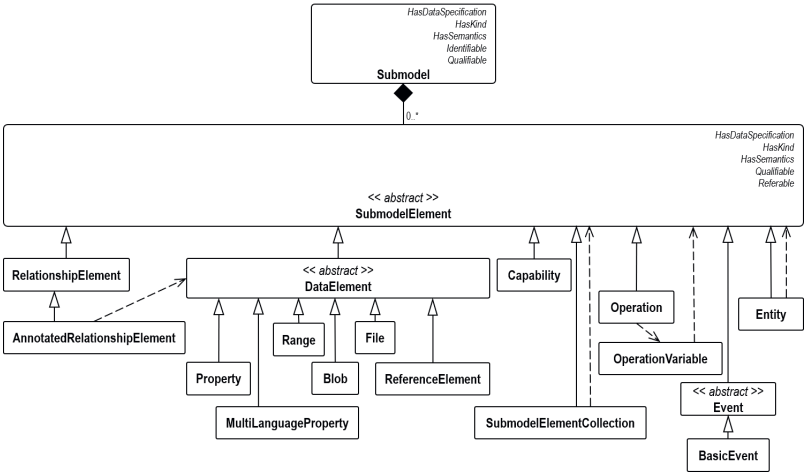

Abbildung 6.5: Modellelemente eines Submodel-Objekts (nach [4])

- *RelationshipElement*: Ein Objekt, welches eine Beziehung zwischen zwei anderen Objekten definiert.
- *AnnotatedRelationshipElement*: Ein Objekt, welches eine Beziehung zwischen zwei anderen Objekten definiert und zusätzliche Informationen mit Hilfe von Datenelementen ermöglicht.
- *Property*: Ein Datenelement, welches einen Einzelwert besitzt.
- *MultiLanguageProperty*: Ein Datenelement, welches eine Menge von Zeichenketten in verschiedenen Sprachen besitzt.
- *Range*: Ein Datenelement, welches einen Wertebereich mit Hilfe eines Minimal- und eines Maximalwerts beschreibt.
- *Blob*: Ein Datenelement, welches ein Blob[8]-Objekt speichern kann.
- *File*: Ein Datenelement, welches eine Adresse zu einer Datei mit Hilfe des Pfads und des Dateinamens inkl. der Dateiendung besitzt.
- *ReferenceElement*: Ein Datenelement, welches eine logische Referenz zu einem anderen referenzierbaren Objekt besitzt.
- *Capability*: Ein Objekt, welches eine Referenz auf eine Fähigkeitsbeschreibung besitzt.
- *SubmodelElementCollection*: Ein Objekt, welches eine beliebige Anzahl an SubmodelElement-Objekten besitzt.

[8]Hier ist Blob als Datentyp aus der Informationstechnik gemeint.

- *Operation*: Ein Objekt, welches Ein- und Ausgabevariablen (Argumente) besitzt und damit eine Funktion beschreibt.
- *BasicEvent*: Ein Objekt, welches eine Referenz zu einem beobachteten Objekt hält.
- *Entity*: Ein Objekt, welches eine Entität beschreibt und Aussagen zu dieser speichert. Eine Entität kann als CoManaged oder SelfManaged spezifiziert werden. CoManaged bedeutet, dass kein Asset im Sinne der Verwaltungsschale vorliegt, z. B. eine Schraube. SelfManaged hingegen bedeutet, dass ein Asset existiert und folglich eine Referenz zu diesem angegeben werden muss.

Sowohl Submodel- als auch SubmodelElement-Objekte sind vom Typ *HasSemantics*. Somit können diese Objekte eine Referenz auf eine semantische Beschreibung besitzen. Diese Beschreibungen sind für eine Kommunikation zwischen Maschinen unabdingbar. Das Konzept zur Verwendung dieser semantischen Beschreibungen ist im Bereich der Produktbeschreibung bereits Stand der Technik. Die IEC 61360 [126] definiert ein Informationsmodell für die Erstellung dieser Konzeptbeschreibungen. Es wurden bereits mehrere Begriffsbibliotheken erstellt, wie z. B. IEC61360-CDD[9] oder ECLASS[10] (vgl. auch Abschnitt 6.1).

Für Submodell-Objekte sind diese Konzeptbeschreibungen nicht geeignet, da sie für einfache Datenelemente entwickelt wurden. Dafür sollen Submodel-Templates erstellt werden, die anschließend zur Laufzeit instantiiert werden. Die Unterscheidung zwischen einer Submodel-Instanz und einem Submodel-Template wird über das Attribute *modellingKind* angegeben. Außerdem besitzt die Verwaltungsschale *View*-Objekte. Diese ermöglichen verschiedene Sichten durch Referenzen auf SubmodelElement-Objekte. Die Objekte der Klassen AssetAdministrationShell und Submodel können eigene Lebenszyklen haben. Aufgrund dessen ist die Beziehung zwischen ihnen mittels Aggregation modelliert. Für die Referenzierung besitzen diese einen weltweit eindeutigen Identifizierer.

Zusammengefasst beschreibt [4] ein Informationsmodell mit allen relevanten Objekten und deren Beziehungen zur Erstellung und Nutzung von Verwaltungsschalen. Das Konzept nutzt bestehende internationale Normen, wie Konzeptbeschreibungen nach der IEC 61360. Zusätzlich wird Wert auf die semantischen Annotationen sowie die Möglichkeit der Trennung von Informationen für verschiedene Anwendungsfälle durch Teilmodelle gelegt. Auch wird das Thema Standardisierung bei Teilmodellen durch die Erstellung von Templates berücksichtigt.

6.4 Thing Description - Web of Things

Die Thing Description ist ein Modell zur Beschreibung von Metadaten und Schnittstellen einer Entität. Das Modell wurde vom W3C 2017 zunächst als Working Draft [11] veröffentlicht bevor 2020 die Recommendation des Standards in [12] folgte. Das Modell ist in eine Gesamtarchitektur eingebunden, die ebenfalls 2020 als Recommendation [145]

[9] https://cdd.iec.ch/
[10] https://www.eclass.eu/

veröffentlicht wurde. Das Modell beschreibt ein Set von Interaktionen für die Integration von verschiedenen Geräten im Web of Things (WoT) und ermöglicht verschiedenen Applikationen, miteinander zu interagieren.

6.4.1 Ziel und Anwendungsbereich

Ziel des Standards ist, mit einem möglichst einfachen Vokabular den Zugriff auf die Informationen eines Dings zu beschreiben. Als Ding (Thing) wird eine „Abstraktion einer physischen oder virtuellen Entität, dessen Metadaten und Schnittstellen mit Hilfe einer WoT Thing Description beschrieben wird“ [145], verstanden. Eine virtuelle Entität stellt dabei eine Komposition aus einem oder mehreren Dingen dar und ist somit immer noch eine physische Entität. Eine weitere Einschränkung hinsichtlich der Dinge, z. B. bezüglich der Domäne, liegt nicht vor. Die Thing Description soll der zentrale Baustein in der Gesamtarchitektur des WoT sein und den Zugriffspunkt zu einem beliebigen Ding darstellen.

Das Modell ermöglicht die Definition von Metadaten sowie die Beschreibung von Zugriffsmöglichkeiten auf Eigenschaften, Funktionen und Events des Dings. Dabei liegen die Werte der Eigenschaften, die konkreten Funktionen oder Events nicht im Modell vor, sondern das Modell ermöglicht die Modellierung von Schnittstellen zu diesen. Diese Schnittstellen beinhalten konkrete Technologie-Endpunkte, deren mögliche Interaktions-Pattern sowie semantische Annotationen. Die Nutzung von bestehenden URI Schemata, z. B. bei der Definition von Protokoll Bindings[11] oder Mediatypen [**?**] wird fokussiert, um die Konformität zu bestehenden Web-Standards zu wahren. Als Serialisierungsformat wird JSON (IETF RFC 8259 [142] genutzt. Ziel ist es, die Beschreibung der Verortung der Daten des Dings sowie deren Zugriffsmöglichkeiten getrennt von der Datenhaltung zu modellieren.

6.4.2 Informationsmodell

Die aktuelle Version des Informationsmodells der Thing Description wurde 2020 in [12] als Recommendation des W3C veröffentlicht. Das Informationsmodell ist in Abbildung 6.6 vereinfacht dargestellt.

Ein *Thing*-Objekt dient in dem Modell als das Beschreibungsobjekt eines Dings. Es enthält Metadaten wie den Identifizierer, den Namen, eine Beschreibung, die Version oder Informationen über den Ersteller und das Erstelldatum. Zusätzlich besteht das Thing-Objekt aus bis zu sechs ausgezeichneten Objekten. Das Security Schema (*SecurityScheme*-Objekt) muss angegeben werden. Es beschreibt, welcher Security-Mechanismus beim Zugriff auf die Daten, Funktionen oder Events verwendet wird, z. B. ohne Security, Basic (unverschlüsselter Benutzername und Passwort) oder OAuth2 [146]. Mit Hilfe des *Link*-Objekts können Verbindungen zu anderen Thing-Beschreibung erstellt werden. Die Objekte *PropertyAffordance*, *ActionAffordance* und *EventAffordance* dienen zur Beschreibung der Interaktion mit einer Eigenschaft, einer Funktion oder einem Event. Für eine Eigenschaft wird beschrieben, wie auf den Wert zugegriffen werden darf. Als Auswahl steht lesend und schreibend zur Verfügung sowie die Möglichkeit diesen Wert zu beobachten und bei Änderungen

[11] `https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml`

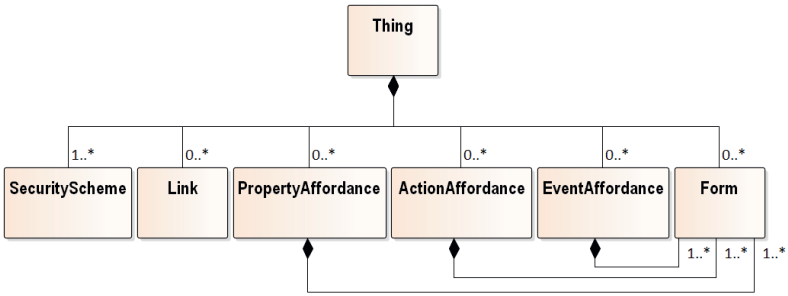

Abbildung 6.6: Informationsmodell der Thing Description

informiert zu werden. Bei den Funktionen können die Input- und Output-Schemata beschrieben und angezeigt werden. Zusätzlich kann hinterlegt werden, ob die Funktion sicher (d. h. keine Zustandsänderungen durch Aufruf) und somit idempotent ist. Für Events kann beschrieben werden, wie eine Subscription angelegt und gelöscht werden kann und welche Daten das Event bei Auslösung übermittelt. Die konkreten Zugriffsadressen und Interaktionspattern werden im *Form*-Objekt festgelegt. Dieses Objekt wird auch Protokoll-Binding genannt. Es beinhaltet die Arten der semantischen Interaktion (z. B. readproperty, writeproperty oder invokeaction), einen Ziel-Internationalized Resource Identifier (IRI), den ContentType des Medientypes (z. B. text/plain oder image/jpeg) und optional das Encoding des Inhalts, das SubProtokol, den Security-Mechanismus und den Antworttyp, falls dieser abweichend vom Anfragetyp ist. Wird ein Form-Objekt im Thing-Objekt verwendet, gilt dies für alle untergeordneten Objekte, solange kein eigenes Form-Objekt dort vorliegt. Liegt eines vor, ist dieses verbindlich und das höhergelegene Form-Objekt ist nicht mehr gültig.

Neben dem vorgestellten Kernmodell besteht zusätzlich noch die Möglichkeit semantische Annotationen an die Elemente anzuhängen. Dies erfolgt über das *ContextExtension*-Objekt. Damit können andere Datenschemas, Ontologie-Einträge oder Konzeptbeschreibungen referenziert werden. Auch das Konzept der Templates wird unterstützt, sodass für bestimmte Ding-Typen Thing-Templates erstellt werden können, die anschließend zur Laufzeit für ein konkretes Ding eines Ding-Typs instanziiert und mit konkreten Werten befüllt werden.

Es wurde ein Informationsmodell mit geringem Vokabular entwickelt, welches die Möglichkeit bietet, die Interaktion mit den Eigenschaften, Funktionen und Events eines Dings zu beschreiben. Es baut auf bestehenden Standards auf und nutzt diese in den verschiedensten Attributen mittels Referenzen. Zusätzlich besteht die Möglichkeit der semantischen Annotationen sowie die Template-Erstellung.

6.5 Vergleich

In diesem Abschnitt werden die drei vorgestellten Standards für die Modellierung von Asset-Informationen miteinander verglichen. Der Fokus liegt zunächst auf der Modellierung von Assets. Aus diesem Grund wird im nachfolgenden Abschnitt allgemein die Bedeutung des Assets und die konkreten Definitionen innerhalb der einzelnen Spezifikationen betrachtet. Danach werden die drei Spezifikationen bzgl. Ziel, Anwendungsbereich und Informationsmodell verglichen.

6.5.1 Asset-Begriff

Für den Begriff des Assets existieren in der Literatur unterschiedliche Definitionen. Diese variieren innerhalb der Branchen sowie in den jeweiligen Domänen. Aus diesem Grund wird nachfolgend ein Vergleich der Definitionen im Kontext von Industrie 4.0 und der vorgestellten Spezifikationen durchgeführt.

Im Referenzarchitekturmodell Industrie 4.0 (RAMI4.0) [123] wird ein Asset als ein „Gegenstand, der einen Wert für eine Organisation hat“, definiert. Diese Definition wird im Industrie 4.0 Glossar[12] erweitert, sodass ein Asset als eine „Entität, die einen wahrgenommenen oder tatsächlichen Wert für eine Organisation hat und der Organisation gehört oder von ihr individuell verwaltet wird“ verstanden wird. Im Englischen wird *Entität* durch „physisches oder logisches Objekt“ ersetzt. Die beiden vorgestellten Spezifikationen, IEC 62832 DF Framework (DF) [8] und Details of the Asset Administration Shell (DotAAS) [140], nutzen jeweils diese Definition (vgl. Abschnitt 6.2 und 6.3). Die Spezifikationen unterscheiden sich jedoch in der Interpretation dieser Entität, da in DF die „Rolle“ als Asset in der Begriffsdefinition zunächst explizit ausgeschlossen ist. Allerdings werden zusätzlich die Begriffe „ProductionSystemAsset“ und „DigitalFactoryAsset“ eingeführt, in denen auch die Rolle mit einbezogen wird. In DF wird jedoch immer der Bezug zu einem Produktionssystem gefordert. Dies ist in DotAAS nicht der Fall, da alle Domänen und Branchen berücksichtigt werden.

In der dritten Spezifikation Thing Description (TD) [11] wird der Begriff Asset nicht explizit verwendet. Jedoch wird „eine physische oder virtuelle Entität, deren Metadaten und Schnittstellen durch eine WoT Thing Description beschrieben werden, wobei eine virtuelle Entität die Zusammensetzung eines oder mehrerer Things ist“, angenommen. Dies kann im Sinne der Modellierung und der anderen Spezifikationen als Asset aufgefasst werden.

Wie zu erkennen ist, existieren verschiedene explizite und implizite Definitionen von Assets. Diese sind im Kern gleich, unterscheiden sich aber in ihrem Betrachtungsrahmen zum Teil stark. Während TD lediglich physische Entitäten betrachtet[13], wird die Definition in DF auf logische Objekte erweitert, jedoch mit Bezug zu dem Bereich Produktionssysteme. Als logische Objekte sind vor allem die Typen von physischen Entitäten zu nennen. Der größte Betrachtungsraum ist in DotAAS gegeben, da ein Versuch zur Betrachtung aller Domänen und Entity-Arten unternommen wird.

[12] https://www.plattform-i40.de/PI40/Navigation/DE/Industrie40/Glossar/glossar.html
[13] Virtuelle Entitäten sind lediglich zusammengesetzte physische Entitäten.

6.5.2 Ziel, Anwendungsbereich und Informationsmodell

Hinsichtlich Ziel und Anwendungsbereich unterscheiden sich die drei vorgestellten Standards auf den ersten Blick wenig. Alle drei ermöglichen die Beschreibung der Informationen einer Entität bzw. Asset. Sie unterscheiden sich aber in der Art der Entität. Während TD ein rein physisches Objekt ohne weitere Einschränkungen betrachtet, erweitert DF dieses um virtuelle Entitäten mit der Einschränkung, dass diese Entitäten einen Bezug zu einem Produktionssystem haben müssen. Dies beinhaltet auch die Beschreibung von Asset-Typen, während mit dem ersten Standard lediglich Asset-Instanzen modelliert werden können. DotAAS erlaubt als offenster Standard alle Arten von Entitäten, solange diese einen Wert für ein Unternehmen haben[14].

In allen drei Spezifikationen wird Wert auf die Wiederverwendung bestehender Standards gesetzt. Während sich DotAAS und DF eher auf die internationalen Normungsgremien IEC und ISO beziehen, bezieht sich TD eher auf die RFC Dokumente der IETF. Dies ist dem Ursprung der Standards geschuldet. Die ersten beiden stammen aus dem Ingenieurswesen und haben daher einen Bezug zu technischen Fragestellungen, wohingegen der dritte Standard aus der Informationstechnik stammt und daher aus dieser Sicht versucht, Fragestellungen zu beantworten. Dies spiegelt sich auch in den Informationsmodellen wider.

Die Zielvorstellungen der drei Standards unterscheidet sich bezüglich der semantischen Annotationen. Diese werden von allen drei Standards unterstützt. In DotAAS und DF ist dies als ein Hauptkonzept fest im Ziel verankert und soll als Erweiterung und Abgrenzung zu anderen bestehenden Standards dienen. In TD werden semantische Annotationen ausschließlich als Erweiterung betrachtet und daher nicht im Ziel direkt berücksichtigt.

Alle Standards definieren neben dem Informationsmodell auch zugehörige Serialisierungsformate. Während TD eine JSON-Serialisierung festlegt, werden in DF zwei Mappings zu AutomationML und OPC UA gegeben, die beide aus der Automatisierungstechnik stammen. In DotAAS werden die meisten Serialisierungsformate angeboten: JSON, XML, AutomationML, OPC UA und RDF.

Um die Konzepte der Standards sinnvoll nutzen zu können, wird eine zugehörige Architektur benötigt. Diese wird sowohl in DotAAS als auch in TD beschrieben und standardisiert. DF trifft hierzu keine Aussagen.

In Bezug auf die Informationsmodelle sind die Spezifikationen DF und DotAAS ähnlich. Beide definieren Objekte, mit denen die Eigenschaften eines Assets inklusive deren Werte modelliert werden können. DotAAS erweitert dieses Modell um die Möglichkeit, auch Funktionen und Events zu beschreiben. Zusätzlich werden die Datenelemente in konkrete Elementtypen, wie Property- oder Range-Objekte, untergliedert. Beide Modelle unterstützen in den Objekten die Vorhaltung der Daten. Über die Datenhaltung wird keine weitere Aussage getätigt. Die Daten könnten beispielsweise durch einen Client von einer Datenquelle bereitgestellt werden. Für eine reine Beschreibung zur Abrufbarkeit der Daten, z. B. durch die Definition eines speziellen proprietären Protokolls, müssen die zur Verfügung stehenden Objekten genutzt werden. Hierzu existieren derzeit keine Beispiele oder Modellierungsempfehlungen. Das ist der Hauptunterschied zu TD. Das in TD beschriebene Informationsmodell definiert, im Gegensatz zu den anderen beiden Informationsmodellen, keine konkreten

[14]Eine detailliertere Betrachtung ist in Abschnitt 6.5.1 gegeben.

Werte der Eigenschaften, Funktionen oder Events, sondern bietet die Möglichkeit, den Zugriff auf diese zu beschreiben (mit den Form-Objekten). Somit kann eine Applikation diese Informationen über die Thing Description abrufen und anschließend den konkreten Wert über das angegebenen Web-Protokoll eigenständig abrufen. Beide Modellierungen haben ihre Vor- und Nachteile. Wie diese ggf. sinnvoll zusammen genutzt werden können, ist in Abschnitt 6.6 beschrieben.

6.6 Schlussfolgerung

Aus Modellierungssicht sind zunächst alle drei Ansätze sinnvoll. Bei einer detaillierten Betrachtung ist zu erkennen, dass DF und DotAAS ähnlich sind. Dies liegt an der zeitlichen Historie. DF entstand zunächst, worauf DotAAS anschließend aufbaute und viele der Konzepte übernommen hat. Zukünftig sollte versucht werden, eine Harmonisierung oder Zusammenführung der beiden Standards zu erreichen. Da DotAAS umfänglicher ist, wird in dieser Arbeit ausschließlich diese Veröffentlichung weiter betrachtet.

Im Vergleich zu TD weisen die beiden anderen Spezifikationen ähnliche Konzepte auf, unterscheiden sich jedoch in der Zielsetzung. Ein Versuch könnte sein, die Spezifikationen für verschiedene Anwendungsfälle zu nutzen und somit zu kombinieren. Möglich wäre z. B. die Verwaltungsschale für die Modellierung von Asset-Typen zu nutzen, da die Verwaltungsschale ursprünglich für diesen Zweck entwickelt wurde[15]. Dieser Bereich ist in aktuellen Forschungsprojekten und Anwendungsszenarien evaluiert und funktionsfähig. Das Informationsmodell bietet für die Ablage der Typ-Informationen die Möglichkeit, statische Werte direkt abzulegen und abrufbar zu machen. Für die Modellierung von Asset-Instanzen ergeben sich besondere Anforderungen in Bezug auf den Zugriff der Ist-Daten. Es müssen eigene Zugriffsmöglichkeiten über verschiedene proprietäre Protokolle möglich sein. Dies ist aktuell in DotAAS noch nicht modelliert. So fehlt die Möglichkeit anzugeben, *wo* die Daten liegen und *wie* auf diese zugegriffen werden kann. Dass die Verwaltungsschale diesen Zugriff immer besitzt, ist eher unwahrscheinlich, da die Security betrachtet werden muss. Darf die Verwaltungsschale auf die Anlagen zugreifen und die aktuellen Ist-Werte abrufen und vor allem auch manipulieren? Genau das wiederum könnte über TD ermöglicht werden, da das verwendete Konzept die Beschreibung der Funktionen und Ist-Daten logisch vom physischen Zugriff trennt. Ein Vorschlag ist daher, diese beiden Modellierungen zusammenzubringen und für Asset-Typen die Verwaltungsschale und für Asset-Instanzen eine Mischung aus Verwaltungsschale und Thing-Description zu verwenden. Dazu muss die Thing-Description bei den Protokoll-Bindings dahingehend erweitert werden, dass neben Web-Protokollen auch andere im industriellen Umfeld auftretende Protokolle, wie z. B. OPC UA, modelliert werden können.

Für diese Arbeit ist die Datenablage im Hintergrund zunächst nicht relevant, da die semantische Interaktion und der Zugriff auf die Informationen im Vordergrund stehen. Da der aktuelle Fokus in der Automatisierungs-Community auf der Verwaltungsschale liegt, wird diese als Anwendungsbeispiel für die Vorstellung des Konzepts genutzt. Aus diesem Grund wird der Informationsaustausch zwischen Verwaltungsschalen im nächsten Kapitel genauer betrachtet.

[15] Es sollen Verwaltungsinformationen zu einem Asset dargestellt werden.

7 Informationsaustausch bei Verwaltungsschalen

Im vorherigen Kapitel wurden die Ziele und Anwendungsbereiche sowie die Informationsmodelle von drei Standards für die Modellierung und den Austausch von Asset Informationen vorgestellt und anschließend miteinander verglichen. Als Ergebnis wird die Verwaltungsschale als Anwendungsbeispiel für das vorgestellte Konzept dienen. Aus diesem Grund wird in diesem Kapitel der Informationsaustausch bei Verwaltungsschalen genauer betrachtet. Zunächst werden die Erscheinungsformen von Verwaltungsschalen beschrieben. Anschließend wird die Nutzung von Verwaltungsschalen-Teilmodellen beim Informationsaustausch vorgestellt, aktuelle Probleme bzw. offene Fragestellungen analysiert sowie mögliche Lösungsoptionen aufgezeigt.

7.1 Erscheinungsformen

In [147] werden erstmals verschiedene Erscheinungsformen von Verwaltungsschalen beschrieben. Insgesamt werden drei Arten beschrieben, die in Abbildung 7.1 dargestellt sind. In der Literatur werden für die einzelnen Erscheinungsformen verschiedene Namen eingeführt. In [147] sind folgende drei Begriffen definiert: Passive Verwaltungsschale im Dateiformat, Passive Verwaltungsschale mit IP/API-basiertem Zugang und Aktive Verwaltungsschale. Die Begriffe „passiv“ und „aktiv“ beziehen sich nicht auf die Kommunikationsfähigkeit, sondern auf die Rolle, die die jeweilige Verwaltungsschalen-Erscheinungsform in der Wertschöpfungskette spielt. Aus diesem Grund wurden die Begriffe weiterentwickelt und sind heute in der neueren Version [148] wie folgt festgelegt: Passive Verwaltungsschale, Reaktive Verwaltungsschale und Proaktive Verwaltungsschale. Aufgrund der derzeitigen Diskussionen zur Benennung der Typen in diversen Gremien, werden für diese Arbeit die Begriffe Typ 1, Typ 2 und Typ 3 verwendet. Ein Einblick in diese drei Typen erfolgt in den folgenden Unterkapiteln.

7.1.1 Typ 1

Die „Verwaltungsschale Typ 1“, als erste Erscheinungsform, bietet die Möglichkeit die Verwaltungsschale in Form einer Datei darzustellen. In [4] wurden verschiedene Serialisierungen (z. B. XML oder JSON) sowie ein eigenes Datei-Format AASX spezifiziert. Das AASX-Format bietet die Möglichkeiten, alle zu einem Asset gehörenden Informationen in einer standardisierten Form von einem Wertschöpfungspartner an einen anderen Wertschöpfungspartner zu übertragen. Dies kann auf verschiedene Weisen funktionieren,

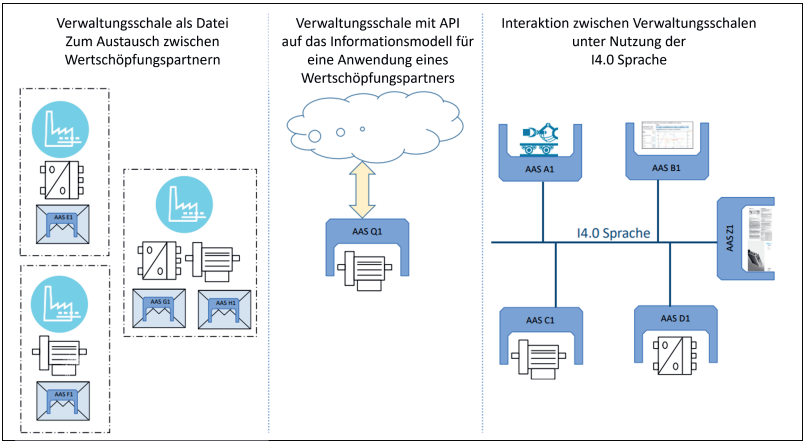

Abbildung 7.1: Erscheinungsformen von Verwaltungsschalen nach [147]

z.B. mittels eines File-Downloads, dem Versand per Mail oder dem Ablegen auf dem konkreten physischen Asset. Die zu übertragenen Informationen können zuvor angepasst werden, sodass ausschließlich freigegebene Daten in der entsprechenden Datei enthalten sind. [147, 148] prognostizieren, dass „dieses Konzept [...] eine neue Qualität dar [stellt], da damit lebensphasenübergreifender, standardisierter Informationsaustausch möglich wird“. Diese Art des Informationsaustauschs setzt voraus, dass eine Software existiert, welche die Datei erstellt, versendet und bei einem anderen Wertschöpfungspartner einliest und weiterverarbeitet, z. B. der AASX-Package-Explorer[1] oder ein Software Development Kit[2].

7.1.2 Typ 2

Die „Verwaltungsschale Typ 2“, als zweite Erscheinungsform, ermöglicht den Zugriff auf die gleichen Informationsinhalte des ersten Typs, jedoch über eine standardisierte Schnittstelle. Die technologie-unabhängige Schnittstellendefinition ist in [5] spezifiziert. Zusätzlich wird in einer nächsten Version dieser Veröffentlichung ein konkretes Technologie-Mapping nach `HTTP` definiert. Die Vorteile dieser Erscheinungsform sind der individuelle Zugriff auf einzelne Verwaltungsschalenelemente sowie die Möglichkeit des individuellen Zugriffsschutzes für verschiedene Anfragende. Außerdem wird die Möglichkeit geboten, Teile der Verwaltungsschale separat bereitzustellen, da für mehrere Modellelemente jeweils Schnittstellen entwickelt wurden. Dies ist z. B. für Verwaltungsschalen und Verwaltungsschalen-Teilmodelle der Fall. Aus Sicht des Informationsaustauschs dienen die Schnittstellen zum Auslesen, Ändern und Erstellen von Informationen. Die Form und der Ort zur Ablage der

[1] https://github.com/admin-shell/aasx-package-explorer
[2] Zum Beispiel: Python-SDK: https://git.rwth-aachen.de/acplt/pyi40aas C#-, Java-SDK: https://www.eclipse.org/basyx/

Informationen im Hintergrund wird nicht spezifiziert und ist ein Implementierungsdetail. Möglichkeiten sind:

- das Ablegen in AASX-Dateien (also Verwaltungsschale Typ 1), die bei jedem Aufruf ausgelesen und wieder beschrieben wird,
- die Speicherung der Informationen in einer Datenbank basierend auf dem Informationsmodell der Verwaltungsschale (vgl. Kapitel 6.3.2), oder
- die Speicherung der Informationen in einem anderen Datenformat, die bei einem Aufruf in das Informationsmodell der Verwaltungsschale transformiert werden.

7.1.3 Typ 3

Die „Verwaltungssschale Typ 3“, als dritte Erscheinungsform, unterscheidet sich stark von Typ 1 und Typ 2. Sie bietet zusätzlich zum reinen Datenzugriff Entscheidungs- und Optimierungsalgorithmen an. Damit wird das Konzept des Asset-bezogenen Informationszugriffs mit dem Konzept von Agenten verknüpft. Mit Hilfe dieser Algorithmen besteht die Möglichkeit, dass Verwaltungsschalen gegenseitig proaktiv in Interaktion treten. Dadurch wird ein erster Schritt in Richtung Autonomie getätigt. Dieser Bereich der Forschung wird zunehmend größer, da viele Ansätze aus der Agententheorie Wiederverwendung finden. In [149] und [150] wurden entsprechende Konzepte vorgestellt, die jeweils zwei Ebenen einführen. Die erste Ebene enthält dabei die Daten in Form von Verwaltungsschalen-Teilmodellen (Typ 1 oder Typ 2). Die zweite Ebenen verwaltet die Algorithmen und Automaten für die Entscheidungen und Optimierungen.

In [147] wird festgelegt, dass die Interaktion mit Hilfe der I4.0-Sprache erfolgen soll, die in der VDI/VDE 2193-Richtlinie [151, 152] beschrieben ist. Diese Sprache definiert das Vokabular, mit dem eine Protokoll-basierte Interaktion ermöglicht wird. In [153] wird ein auf dem Vokabular basierendes Protokoll für ein Ausschreibungsverfahren beschrieben. Für die Informationsmodellierung der Daten werden auch in diesen Protokollen die Verwaltungsschalen-Teilmodelle verwendet.

7.1.4 Vergleich

Bei einem Vergleich der drei Erscheinungsformen ist zu erkennen, dass die ersten beiden auf die Darstellung bzw. den reinen Zugriff von Asset-Informationen beschränkt sind. Es wird die Möglichkeit gegeben, das Informationsmodell der Verwaltungsschale in Form einer Datei oder einer Schnittstelle zur Verfügung zu stellen. Die Semantik des Informationsaustausches steckt in den eigentlichen Daten, hier den Verwaltungsschalen-Teilmodellen. Die dritte Erscheinungsform ermöglicht zusätzlich die Festlegung von eigenen Interaktionsprotokollen. Dadurch verlagert sich die Semantik zunehmend in die Protokolle, die vorgeben, in welcher Reihenfolge welche Nachrichten auszutauschen sind. Mit Hilfe dieser Protokolle wird ein erster Schritt in Richtung der Behavioural Interoperability gegangen. Anschließend werden die Inhalte der Nachrichten wiederum mit den Daten aus den ersten beiden Typen in Form von Verwaltungsschalen-Teilmodellen angereichert.

Aus Kommunikationssicht kann der Austausch in vertikale und horizontale Interaktionen unterteilt werden. Vertikale Interaktion bedeutet, dass eine Hierarchie vorliegt. Das heißt, dass ein Interaktionspartner als Server (Slave) und ein weiterer als Client (Master) fungieren [149]. Der Client ruft die Informationen beim Server ab, sodass die Interaktion immer vom Client initiiert wird. Dies ist z. B. bei der zweiten Erscheinungsform der Fall, bei der die Daten mittels eines Servers bereitgestellt werden. Bei der horizontalen Interaktion sind beide Interaktionspartner gleich berechtigt und werden auch als Peers bezeichnet. Jeder Peer kann Anfragen starten und gleichzeitig auf Anfragen reagieren. Der Unterschied zur Client-Server-Interaktion ist, dass ein eigener Entscheidungsalgorithmus sowie entsprechende Automaten entscheiden, wie mit eingehenden Anfragen umgegangen wird. Die Peers unterliegen dabei immer einem selbst verfolgten Ziel. Eine Verwaltungsschale Typ 3 unterstützt daher sowohl vertikale als auch horizontale Interaktion.

7.2 Nutzung von Verwaltungsschalen-Teilmodellen für semantische Interoperabilität: Offene Fragestellungen und mögliche Lösungsoptionen

Unabhängig von der Erscheinungsform werden die Asset-Informationen mit Hilfe von Verwaltungsschalen-Teilmodellen für andere Kommunikationspartner zur Verfügung gestellt.

In diesem Abschnitt wird beschrieben, wie Verwaltungsschalen-Teilmodelle für die semantische Interoperabilität zu nutzen sind. Wie bereits in Abschnitt 4.2 aufgezeigt, gehen damit einige Probleme einher. Mit Hilfe von Verwaltungsschalen-Teilmodellen werden domänenspezifische Aspekte eines Assets dargestellt. In Kapitel 6.3.2 sind die Modellelemente für die Darstellung eines Verwaltungsschalen-Teilmodells bereits vorgestellt. Dadurch ist die syntaktische Interoperabilität gewährleistet. Im Nachfolgenden wird näher auf die Semantik und die Bedeutung für den Informationsaustausch eingegangen.

Ein Verwaltungsschalen-Teilmodell ist die Darstellung genau eines Asset-Aspekts und wird für einen konkreten Anwendungsfall entwickelt. Folgendes Beispiel soll dies illustrieren: Als Asset soll ein Roboter dienen. Je nach Nutzer des Roboters werden andere Informationen des Roboters benötigt. Für die Raumplanung werden z. B. die geometrischen Maße benötigt. Ein Elektriker benötigt jedoch die Informationen für den elektrischen Anschluss (z. B. Anschlussleistung, benötigte Spannung). Ein Software-Entwickler hingegen benötigt die Informationen über die Steuerung, die Protokolle oder die Byte-Belegung. Um für jeden Nutzer die passenden Informationen bereitzustellen, könnte ein Verwaltungsschalen-Teilmodell mit einer flachen Liste aller Eigenschaften erstellt werden. Dies ist aber nicht zielführend, da viele nicht benötigte Informationen für einen Nutzer vorliegen. Das Ziel ist daher, die Informationen Anwendungsfall-orientiert in unterschiedliche Verwaltungsschalen-Teilmodelle zu strukturieren. Durch die Definition von unabhängigen Verwaltungsschalen-Teilmodellen kann weitere Semantik zu den Informationen hinzugefügt werden, beispielsweise wie sich Höhe und Ausdehnung des Roboters zueinander verhält. Viele dieser Informationen gelten auch für andere Roboter, sodass das Ziel sein sollte,

möglichst viele Verwaltungsschalen-Teilmodelle zu standardisieren. Da verschiedene Stakeholder jedoch ggf. andere Informationen bei ihrem Informationsaustausch benötigen, können die Verwaltungsschalen-Teilmodelle lediglich in Grundzügen standardisiert werden [147, 148]. Anschließend werden einzelnen Interaktionspartner weitere Elemente spezifizieren, die diese zusätzlich für den konkreten Anwendungsfall austauschen.

Applikationen, die Informationen über so einen Aspekt des Assets erhalten, müssen das entsprechende Verwaltungsschalen-Teilmodell verstehen. Hierfür werden semantische Verweise in Form der SemanticID zur Verfügung gestellt, die auf Konzept-Beschreibungen zeigen. Applikation können in einer Art Type-Checking prüfen, ob das Element der semantischen Beschreibung folgt, die die Applikation an dieser Stelle erwartet oder benötigt. Jedoch ist dies vielfach nicht ausreichend, da das Element in seinem Kontext verstanden werden muss. Das Verwaltungsschalen-Teilmodell kann diesen Kontext definieren und gibt dem Element eine entsprechende Bedeutung innerhalb dessen. Dieser Kontext ist für die Applikation lediglich ermittelbar, wenn die Verwaltungsschalen-Teilmodelle bekannt sind.

Aus diesem Grund wird versucht, die Verwaltungsschalen-Teilmodell-Templates zu standardisieren. Da jedoch für verschiedene Aspekte unterschiedliche Verwaltungsschalen-Teilmodell-Templates entstehen werden, tritt das Problem der semantischen Interoperabilität auf. Die Frage lautet, wie bei Verwaltungsschalen auf Verwaltungsschalen-Teilmodell-Ebene eine semantische Interoperabilität erreicht wird.

Eine Lösung ist, dass ein gemeinsames und standardisiertes Verwaltungsschalen-Teilmodell-Template pro Aspekt existiert. Die Verwaltungsschalen-Teilmodell-Templates zu verschiedenen Aspekten sollten dabei möglichst disjunkt sein, damit nicht zu ähnliche Verwaltungsschalen-Teilmodell-Templates entstehen und verschiedene Applikationen verschiedene Verwaltungsschalen-Teilmodell-Templates nutzen. Die Möglichkeit von firmeninternen Verwaltungsschalen-Teilmodell-Templates oder firmeninternen Erweiterungen ist nicht mehr gegeben. Was passiert jedoch, wenn eine neue Version veröffentlicht wird? Werden die alten Versionen nicht mehr nutzbar, damit nicht zu ähnliche Verwaltungsschalen-Teilmodell-Templates existieren? Nach Ansicht des Autors ist dies keine zukunftsorientierte Lösung.

Eine andere Lösung ist, dass jeder Stakeholder beliebige Verwaltungsschalen-Teilmodell-Templates erstellen, diese zentral verwaltet werden und jede Applikation diese kennt und nutzt. Dies ist bei einer geringen Anzahl möglich, skaliert aber leider nicht. Gerade vor dem Hintergrund, dass die Verwaltungsschale Domänen- und Branchen-übergreifend ist, werden viele Verwaltungsschalen-Teilmodell-Templates entstehen. Die Vergangenheit hat gezeigt, dass der Markt dies alleine nicht regeln kann. Wie in Abschnitt 4.2 aufgezeigt, werden sich verschiedene Interessengruppen bilden, die unabhängig voneinander Verwaltungsschalen-Teilmodell-Templates erstellen.

Eine weitere und vielversprechende Lösung ist die automatische Abbildung der Verwaltungsschalen-Teilmodell-Templates untereinander. Das bedeutet, dass ein Mapping der Elemente mehrerer Verwaltungsschalen-Teilmodell-Templates durchgeführt wird. Dadurch wird ein Nutzer bei der Erstellung bzw. Befüllung von neuen Verwaltungsschalen-Teilmodellen unterstützt, sodass dies automatisiert erfolgt. Hierfür kann das in Kapitel 5 vorgestellte Konzept der Modelltransformation genutzt werden. In dieser Arbeit

wird die Modelltransformation als eine mögliche Lösung vorgestellt, um formal angeforderte Verwaltungsschalen-Teilmodell-Instanzen aus basierenden Verwaltungsschalen-Teilmodell-Instanzen zu erstellen. Welche Arten der Modelltransformation zwischen Verwaltungsschalen-Teilmodellen (Informationsmodellen) auftreten und welche Anforderungen an eine entsprechende Modelltransformationssprache vorliegen, wird im nächsten Kapitel näher beleuchtet.

8 Modelltransformationen für die semantische Interoperabilität zwischen verschiedenen Informationsmodellen

Um die semantische Interoperabilität zwischen Informationsmodellen zu ermöglichen, wird das Konzept der Modelltransformation (vgl. Kapitel 5) genutzt. Die Modelltransformation kann in zwei Arten unterteilt werden: syntaktische und semantische Transformation. Worin der Unterschied liegt und wann welche Transformation auftritt, wird in Abschnitt 8.1 anhand der Informationsmodelle der Verwaltungsschale (Verwaltungsschalen-Teilmodell) genauer beschrieben.

Da in dieser Arbeit eine Modelltransformationssprache entwickelt wird, werden in den darauffolgenden Abschnitten die einzelnen Schritte des Leitfadens zur Erstellung einer Transformationssprache aus Abschnitt 5.4.2 durchgeführt: Klassifikation der Modelltransformation zwischen Informationsmodellen in Abschnitt 8.2, Anforderungsdefinition und benötigte Sprachelemente in Abschnitt 8.3 und Evaluation bestehender Transformationssprachen in Abschnitt 8.4. Für ein besseres Verständnis werden an entsprechenden Stellen Beispiele aus dem Kontext der Verwaltungsschale vorgestellt.

8.1 Syntaktische und semantische Transformationen

In Abschnitt 3.1 der Vorveröffentlichung [19] wird die Unterscheidung in syntaktische und semantische Transformationen [154] bei Verwaltungsschalen-Teilmodellen beschrieben und anhand von konkreten Beispielen erläutert. Nachfolgend ist dieser vorveröffentlichte Abschnitt abgedruckt:

„Die Modell-zu-Modell-Transformationen können in syntaktische und semantische Transformationen unterschieden werden. Bei einer syntaktischen Transformation wird lediglich die abstrakte Syntax der Modelle umgewandelt, d. h. die modellierten Informationen bleiben gleich, werden jedoch durch andere Modellelemente dargestellt. Bei einer semantischen Transformation werden die Informationen der Modellelemente hingegen genutzt, um daraus neue Strukturen im Zielmodell zu erzeugen und somit explizit neue Semantik hinzuzufügen.

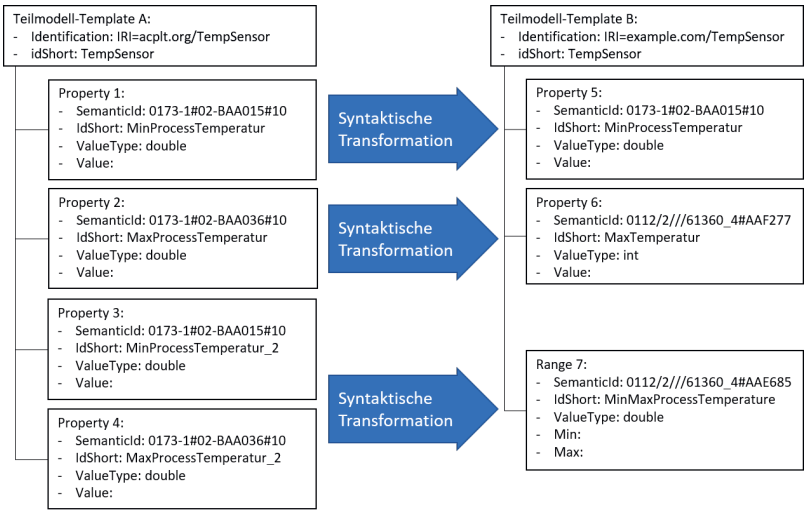

Abbildung 8.1: Syntaktische Transformationen bei Teilmodellen [19]

In Abbildung 8.1 sind drei mögliche syntaktische Transformationen gezeigt, d.h. solche, bei denen sich ausschließlich die syntaktische Repräsentation (z. B. Modellelementtyp, Datentyp, SemanticId) wandelt, nicht die Werte. Die erste Transformation zeigt eine reine Wertübertragung von zwei gleich modellierten Eigenschaften. Da Teilmodelle von unterschiedlichen Organisationen und Firmen erstellt werden, können dieselben Eigenschaften in verschiedenen Teilmodell-Templates jedoch auch unterschiedlich modelliert sein, wie dies bei den Properties 2 bis 4 bzw. 6 und 7 der Fall ist. Im zweiten Fall wurde für die Eigenschaft „Maximale Prozesstemperatur“ zwei verschiedene Eigenschaftsbibliotheken genutzt (ECLASS und IEC61360-CDD). Dementsprechend sind die Attribute der Properties (Property 2 und Property 6) unterschiedlich, die Semantik bleibt jedoch gleich. Als letztes Beispiel wurden für die Eigenschaften „Maximale Prozesstemperatur“ und „Minimale Prozesstemperatur“ auf der linken Seite zwei einzelnen Property-Elemente verwendet. Auf der rechten Seite wurden diese hingegen gemeinsam in einem Range-Element modelliert. Bei der Transformation müssen daher die Werte (Values) aus den Property-Elementen 3 und 4 extrahiert und in die entsprechenden „Min“- und „Max“-Attribute des Range-Elements übertragen werden.

In Abbildung 8.2 sind demgegenüber semantische Transformationen dargestellt. Hierbei werden die Werte (Values) aus den Property-Elementen 8 und 9 benutzt, um auf Basis ihrer Semantik mit Hilfe eines physikalischen Zusammenhangs einen neuen Wert zu berechnen. Dieser wird in Property-Element 11 gespeichert. Bei der unteren Transformation wird basierend auf der Anzahl der digitalen Eingänge (Value von Property 10) eine entsprechende Anzahl von Repräsentationen digitaler Eingänge (SubmodelElementCollection 13) angelegt.“ [1]

[1] Abschnitt 3.1 der Vorveröffentlichung [19].

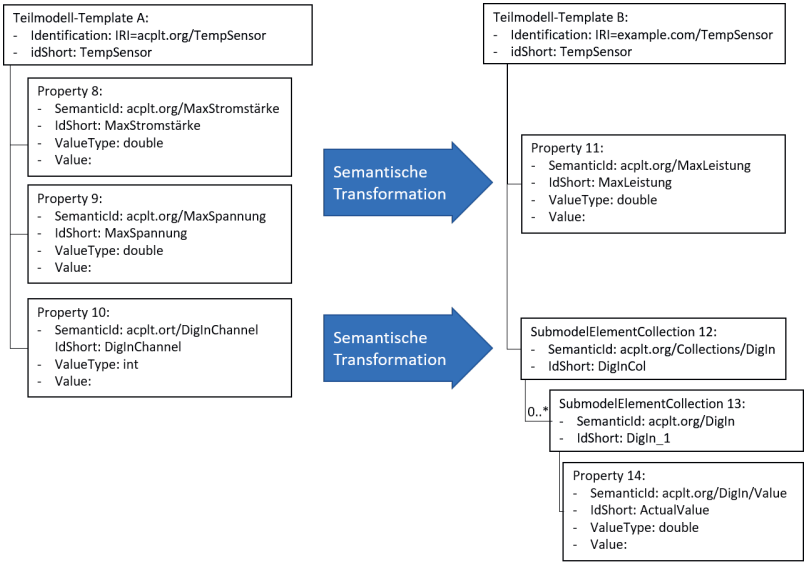

Abbildung 8.2: Semantische Transformationen bei Teilmodellen [19]

8.2 Klassifikation der Transformationen

Im ersten Schritt wird eine Klassifikation zur Auswahl einer Transformationssprache durchgeführt. Dafür werden die Merkmale aus Abschnitt 5.2 genutzt und deren Ausprägungen festgelegt.

Bei der vorliegenden Modelltransformation sollen (semi)-automatisch neue Informationsmodell-Instanzen erstellt und mit Werten aus bereits existierenden Informationsmodell-Instanzen befüllt werden. Die bereits existierenden Informationsmodell-Instanzen sollen dabei nicht verändert werden. Eine Rückkopplung ist somit ausgeschlossen und die Transformation erfolgt rückwirkungsfrei. Die Transformationsrichtung ist somit unidirektional. Im Rahmen dieser Arbeit werden die Transformationen als nicht-inkrementell angesehen, da das Aktualisieren von bestehenden Informationsmodell-Instanzen zunächst nicht weiter betrachtet wird. Als Ergebnis der Transformation wird ein Modellartefakt und explizit kein Textartefakt erzeugt. Aus diesem Grund liegt eine Modell-zu-Modell-Transformation vor. Da genau eine neue Informationsmodell-Instanz erzeugt wird, aber Informationen aus beliebig vielen existierenden Informationsmodell-Instanzen genutzt werden, liegt eine M:1 Transformation vor. Die Transformation wird als horizontal bezeichnet, da die Quellmodelle und das Zielmodell auf gleicher Abstraktionsebene liegen. Zudem sind die Quell- und Zielmodelle unterschiedlich, sodass eine Out-Place Transformation gegeben ist. In dieser Arbeit werden die Informationsmodell-Templates nicht als eigene Metamodelle aufgefasst, sondern als eine konkrete Ausgestaltung der Modellelemente des zugehörigen Metamodells. Da die beiden

Metamodelle von Quell- und Zielmodell gleich sind, liegt eine endogene Transformation vor.

Zusammengefasst kann die Transformation zwischen Informationsmodellen zur Assetbeschreibung gemäß Kapitel 6 wie folgt klassifiziert werden: unidirektional, nicht inkrementell, Modell-zu-Modell, M:1, horizontal, out-place und endogen.

8.3 Anforderungen an die zu entwickelnde Transformationssprache

Neben der allgemeinen Klassifikation der Modelltransformation in Abschnitt 8.2 werden Anforderungen an die Gestaltung der Sprachsyntax formuliert. Hierzu zählen Anforderungen an die Art und Weise, wie die Sprache nachher in der Praxis verwendet werden soll. Zusätzlich existieren zur Definition von Regeln Anforderungen bezüglich der benötigten Sprachelemente.

8.3.1 Allgemeine Anforderungen

In [19] wird bereits der allgemeine Anwendungsfall der Transformationssprache beschrieben. Es soll eine (semi-)automatische Generierung von neuen Informationsmodell-Instanzen basierend auf Informationen aus anderen bereits existierenden bzw. vorliegenden Informationsmodell-Instanzen erfolgen. Die Transformationsdefinitionen sollen in erster Linie von Domänenexperten, z. B. Datenmodellierern in Unternehmen, erstellt werden, die bereits das Konzept und die Grundelemente des übergeordneten Konzepts (hier Verwaltungsschale) kennen. Aufgrund dessen soll die Syntax der Transformationssprache für Domänenexperten einfach zu verstehen sein und die individuellen Arbeitsabläufe unterstützen. Um dies zu erreichen, soll die Syntax möglichst nahe am Metamodell der Informationsmodelle (hier Metamodell der Verwaltungsschale) sein und ausschließlich mit notwendigen Sprachelementen erweitert werden. Dies ermöglicht anschließend eine einfachere und breitere Verwendung.

Anforderung 8.1 *Die Syntax der Sprache soll einfach zu verstehen und nahe am Metamodell der Informationsmodelle sein.*

Bei klassischen Transformationsproblemen werden Regeln für bestimmte Objekttypen definiert, die dann auf jedes Objekt im Quellmodell angewendet werden. In diesem Fall soll jedoch die Hauptstruktur der zu erzeugenden Informationsmodell-Instanz in der Transformationsdefinition konform zum gewünschten Informationsmodell-Template definiert werden[2]. Das Ziel einer Transformationsdefinition soll sein, eine einzige wohlgeformte Informationsmodell-Instanz aus den gesammelten Informationen bestehender Informationsmodell-Instanzen zu erzeugen.

[2]ForEach-Strukturen, die über die Objekte der Quellmodelle iterieren, werden nur vereinzelt auftreten.

Anforderung 8.2 *Die Sprache soll genau eine wohlgeformte Informationsmodell-Instanz erzeugen.*

Um dem Nutzer die Erstellung einer Informationsmodell-Instanz zu vereinfachen, soll die Struktur des Zielmodells möglichst in einem verständlichen, vorzugsweise *deklarativen Template-ähnlichen Stil* [155] formulierbar sein. Dies bedeutet auch, dass eine explizite Definition von einzelnen Regeln nicht existieren muss. In der Terminologie von Modelltransformation bedeutet das: Jede Transformation besteht aus genau einer einzigen Regel, die ein vollständiges Zielmodell erzeugt. Durch diese Forderung werden keine Sprachelemente für die Definition von expliziten Ausführungsbedingungen, Steuerung von Regelanwendungen, Regelauswahl, Regelwiederholung und Unterstützung von Phasen erforderlich. Zudem ist eine syntaktische Separation nicht notwendig.

Anforderung 8.3 *Die Sprache soll die Definition der Struktur des Zielmodells in einem deklarativen Template-ähnlichen Stil unterstützen.*

Die vorgestellten Metamodelle für die Asset-Information sind bereits spezifiziert. Daher liegen in der IT und Automatisierungstechnik viele verschiedene Implementierungen vor, die in unterschiedlichen Programmiersprachen entwickelt wurden. Damit die Sprache eine möglichst breite Verbreitung und Umsetzung in bestehenden Systemen erreicht, sollen so wenig Sprachelementen wie möglich definiert werden. Dies erleichtert die Implementierung in den erforderlichen Software-Tools. Gleichzeitig soll die Sprache auf Konzepten bzw. Modellen aus dem aktuellen Stand der Wissenschaft basieren.

Anforderung 8.4 *Die Sprache soll einfach implementierbar sein und auf bestehenden Konzepten und Modellen aus dem Stand der Wissenschaft und Industrie basieren.*

Zwingend erforderlich ist eine leistungsfähige Ausdruckssyntax, mit deren Hilfe das Erstellen, Manipulieren und die Kombination von Informationen aus den Quellmodellen ermöglicht wird. Um dem Nutzer die Wiederverwendung von sich wiederholenden Ausdrücken zu vereinfachen, sollen Sprachmittel für die Modularisierung und Wiederverwendung mit Parametrisierung existieren. Die konkreten benötigten Sprachelemente werden in Abschnitt 8.3.2 genauer behandelt.

8.3.2 Benötigte Transformationssprachelemente

Um sowohl die syntaktischen als auch semantischen Transformationen (vgl. Abschnitt 8.1) zwischen Informationsmodellen zu ermöglichen, werden entsprechende Transformationssprachelemente benötigt. Diese sollen den allgemeinen Anforderungen aus Abschnitt 8.3.1 entsprechen. Eine Transformationssprache muss daher nachfolgende Sprachelemente anbieten:

Das Ziel der Transformation ist die Erstellung einer neuen Informationsmodell-Instanz. Dafür sollen die einzelnen Informationsmodellelemente mit Hilfe von entsprechenden Sprachelementen angelegt werden. Eine Sprache muss also die Erstellung von allen Informationsmodellelementtypen des zugehörigen Metamodells (hier die des Verwaltungsschalen-Metamodells nach [4]) unterstützen. Dabei ist zu beachten, dass die Objekte inklusive ihrer Attribute angelegt werden, auch wenn diese zunächst nicht initialisiert werden.

Anforderung 8.5 *Die Sprache muss Sprachelemente für die Erstellung von Informationsmodellelementen nach dem zugehörigen Metamodell bereitstellen.*

Das Erstellen allein ermöglicht jedoch nicht das Setzen der Attributwerte dieser Informationsmodellelemente. Die Sprachelemente sind so zu gestalten, dass diese die Datentypen der einzelnen Attribute unterstützen. Zudem sollen je nach Datentyp unterschiedliche Ausdrücke (Expressions) ermöglicht werden. Für eine Zeichenkette soll beispielsweise ermöglicht werden, zwei einzelne Zeichenketten anzugeben, die dann konkateniert werden (z. B. Zeichenkette1 + Zeichenkette2). Aus diesem Grund werden Sprachelemente zur Definition von Ausdrücken und zum Setzen von Attributen benötigt.

Anforderung 8.6 *Die Sprache muss Sprachelemente für das Definieren von Ausdrücken und das Setzen von Attributen in den Informationsmodellelementen beinhalten.*

Sowohl bei der syntaktischen als auch bei der semantischen Transformation werden Quell-Informationsmodellelemente in Ziel-Informationsmodellelemente transformiert. Dafür müssen zunächst die benötigten Quell-Informationsmodellelemente in den Quell-Informationsmodell-Instanzen gefunden werden. Es müssen Sprachelemente definiert werden, die das Finden von Informationsmodellelemente im Quellmodell anhand ihrer Attributwerte ermöglichen. Bei Verwaltungsschalen kann z. B. eine Suche über die *idShort*[3] bzw. eine *Liste von idShorts* oder über die *semanticID* erfolgen.

Anforderung 8.7 *Die Sprache muss Sprachelemente für das Finden von Informationsmodellelementen in den Quell-Informationsmodellen basierend auf Attributwerten vorsehen.*

Nach dem Finden des Informationsmodellelements in der zugehörigen Quell-Informationsmodell-Instanz müssen die Werte der einzelnen Attribute ausgelesen werden können. Der Zugriff auf die Attribute soll für den Nutzer der Sprache einfach und intuitiv sein.

Anforderung 8.8 *Die Sprache muss Sprachelemente für das Auslesen von Attributwerten aus Objekten in den Quell-Informationsmodell-Instanzen bereitstellen.*

[3]Objekt-Identifier, die in ihrem Namensraum eindeutig sind, wie z. B. Verwaltungsschalen-Teilmodell oder SubmodelElementCollection.

In einigen Fällen können bedingte Fallunterscheidungen (z. B. If-Then-Else) notwendig sein. Gerade bei optionalen Informationsmodellelementen kann mit Hilfe einer bedingten Fallunterscheidung entschieden werden, ob im Zielmodell ein Informationsmodellelement angelegt werden muss oder nicht. Zusätzlich kann basierend auf Attributwerten in einem Informationsmodellelement der Quell-Informationsmodell-Instanz entschieden werden, ob oder wie weitere Informationsmodellelemente in der Ziel-Informationsmodell-Instanz angelegt oder deren Attribute gesetzt werden sollen. Aus diesem Grund werden Sprachelemente benötigt, die die Definition von bedingten Fallunterscheidungen ermöglichen.

Anforderung 8.9 *Die Sprache muss Sprachelemente zur Unterstützung bedingter Fallunterscheidungen definieren.*

Bei einigen Transformationen müssen mehrere gleiche Modellelemente in der Ziel-Informationsmodell-Instanz angelegt werden, deren Bezeichner sich unterscheiden. Um das Anlegen einer abhängigen Anzahl zu vereinfachen, sollen Sprachelemente bereitgestellt werden, die die Erstellung und Nutzung von Schleifen ermöglichen.

Anforderung 8.10 *Die Sprache muss Sprachelemente zur Unterstützung von Schleifen beinhalten.*

Die Zwischenspeicherung und der spätere Zugriff auf Werte in Variablen kann notwendig sein. Dies kann ein unnötiges wiederholtes Einlesen von Attributwerten eines Informationsmodellelements in den Quell-Informationsmodell-Instanzen einsparen oder die Möglichkeit der Speicherung von Zwischenwerten bei Formeln oder sonstigen Operationen ermöglichen. Zusätzlich können Variablen bei Schleifen für das Zählen der Schleifendurchgänge genutzt werden. Aus diesem Grund sollen Sprachelemente zur Speicherung und zum Auslesen von Variablen während der Auswertung der Transformationsdefinition existieren.

Anforderung 8.11 *Die Sprache muss Sprachelemente zur Speicherung und zum Lesen von Variablen bereitstellen.*

Es kann davon ausgegangen werden, dass eine Vielzahl von Transformationsdefinitionen existieren und dass Ausdrücke[4] ähnlich oder in gleicher Form in verschiedenen Transformationsdefinitionen vorkommen werden. Um eine effiziente Nutzung und die schnelle Erstellung von Transformationsdefinitionen zu ermöglichen, sollten einmal erstellte Ausdrücke wiederverwendet werden können. Diese können auch zunächst abstrakter und ohne konkrete Werte bzw. mit Variablen spezifiziert und dann bei der Nutzung mit entsprechenden Werten belegt werden. Dies wird in Programmiersprachen häufig als Makro bezeichnet. Zur Ermöglichung werden Sprachelemente für die Erstellung, die Speicherung und die Nutzung dieser Makros benötigt.

Anforderung 8.12 *Die Sprache muss Sprachelemente zur Erstellung, Speicherung und Nutzung von Makros definieren.*

[4]Hier liegt der Fokus auf komplexen Ausdrücken.

8.4 Evaluation bestehender Transformationssprachen

Basierend auf der Klassifikation und der Anforderungsdefinition wurden im dritten Schritt, analog zur Vorgehensweise in Abschnitt 5.4.2, bestehende Transformationssprachen hinsichtlich ihrer Eignung evaluiert. Das Ziel ist, diese direkt zu verwenden oder anzupassen (vgl. Vorveröffentlichung [21]).

Zunächst wurden typische generische Transformationssprachen, wie QVT [156], ATL [99], ETL [102] oder VIATRA [101] betrachtet. Andere Ansätze nutzen bestehende GPTL und erweitern diese um domänenspezifische Sprachelemente. Zum Beispiel wird in [113] eine abgeleitete Transformationssprache für Modelle von Benutzerschnittstellen entwickelt. Es kann gezeigt werden, dass die Syntax nicht neu entwickelt werden muss, sondern auf bestehende Konstrukte aufsetzen kann. Jedoch muss aufgrund der Erweiterung auch ein neues Transformationssystem entwickelt werden, welches sowohl die domänenspezifischen als auch alle Funktionalitäten der GPTL unterstützt. Dies ist sinnvoll, wenn die GPTL um Funktionalität erweitert werden soll. Bezogen auf die Anforderungen aus Abschnitt 8.3.1 konnte dieses Verfahren oder die direkte Nutzung jedoch nicht verwendet werden, da die generischen Transformationssprachen zu viele nicht benötigte Funktionen und Einschränkungen, wie die Definition von Regeln oder keine einfache Unterstützung von Schleifen (ATL), besitzen. Wenn nur wenige Funktionen benötigt werden, sollte eine neue Sprache entwickelt werden, da die Dokumentation dieser kürzer wird und Entwicklern somit die Erstellung einer Implementierung vereinfacht wird.

Aus diesem Grund wurden im nächsten Schritt Frameworks zur Generierung von domänenspezifischen Transformationssprachen hinsichtlich ihrer Nutzbarkeit analysiert. Baar und Whittle entwickelten in [103] ein Verfahren zur Generierung der abstrakten Syntax einer domänenspezifischen Transformationssprache in Form eines Metamodells. Dazu wird das Metamodell der domänenspezifischen Modellierungssprache verwendet. Es wird konzeptionell beschrieben, wie die Syntax automatisiert erstellt werden kann. Eine Beschreibung für die Ausführung der Regeln und wie eine mögliche Implementierung aussehen könnte wird nicht bereitgestellt. Auch existiert keine vollständige Implementierung. Bei weiteren Recherchen konnten keine anderen Arbeiten bzw. Veröffentlichungen zu diesem Framework gefunden werden. Daher wird von einer Nutzung dieses Frameworks in dieser Arbeit abgesehen.

In [108] wurde das Generator-Framework *Marius* entwickelt, welches Modelle, die mit der EBNF Syntax entwickelt wurden, transformieren kann. Auch in [106] wurde ein ähnliches Verfahren entwickelt, wie aus einer domänenspezifischen, textuellen Modellierungssprache die Transformationssprache sowie das Transformationssystem automatisiert generiert werden kann. Allerdings wird gefordert, dass die Sprachen mit dem *MontiCore* Framework [157] unter Nutzung der MontiCore-Grammatik (ähnlich zu Erweiterte Backus-Naur-Form (EBNF)) entwickelt werden. Hölldobler hat in ihrer Arbeit [110] eine Erweiterung entwickelt, welche die n-zu-m Transformation ermöglicht. Allerdings müssen die Modellsprachen immer noch in der MontiCore-Grammatik beschrieben sein.

Die drei genannten Verfahren schränken die Nutzbarkeit dahingehend ein, dass diese lediglich bestimmte Sprachen unterstützen. Die Sprachen müssen formal textuell beschrieben sein und entweder der MontiCore-Grammatik oder der EBNF Syntax folgen. Zusätzlich

wird die konkrete Syntax bereits von dem Framework vorgegeben, sodass die Definition von eigenen Syntaxelementen aufwändiger und nur bedingt möglich ist. Aus diesem Grund wird die Nutzung dieses Frameworks in dieser Arbeit nicht weiter betrachtet.

In [112] wurde ein Metamodell für die Entwicklung von DSTL entwickelt, welches alle notwendigen Klassen, die bei einer Transformation benötigt werden, beschreibt. Basierend auf diesem Modell wurde ein Verfahren entwickelt, wie automatisiert die abstrakte und konkrete Syntax sowie das Transformationssystem erstellt werden. Der Nutzer muss seine Transformationsregeln mit diesem Metamodell beschreiben und kann dann alles weitere generieren lassen. Dies vereinfacht die Erstellung der Syntax sowie des Transformationssystems. Jedoch ist der Nutzer an diesen Workflow und das erstellte Tool gebunden. Zudem muss der Nutzer alle möglichen Regeln formulieren und diese konsistent halten. Der benötigte Overhead und die Bindung an das erstellte Tool schränken die Verbreitung und Nutzung in den Domänen signifikant ein.

Zusammenfassend lässt sich sagen, dass alle Frameworks zur Generierung benutzerdefinierter DSTLs nicht verwendbar sind. Entweder sind die Frameworks nicht vollständig entwickelt, decken nur wenige Schritte ab, treffen Annahmen, die nicht zum gegebenen Anwendungsfall passen - z.B. die Notwendigkeit einer konkreten Syntax der Modellierungssprache - oder es werden Anforderungen an die vorhandenen Werkzeuge in der Systemlandschaft gestellt. Zudem sind vielfach die Sprachelemente der generierten Sprachen nicht so leicht zu verstehen, da sie generisch formuliert sind, um verschiedene Anwendungsfälle abzudecken. Dies reduziert die Verbreitung in hohem Maße, da die Nutzung für Domänenexperten erschwert wird.

8.5 Fazit

In Abschnitt 8.4 wurde aufgezeigt, dass die geforderten Anforderungen aus Abschnitt 8.3 nicht mit einer am Markt verfügbaren Transformationssprache gelöst werden kann und auch kein Framework zur Generierung einer passenden Sprache genutzt werden kann. Aus diesem Grund muss eine neue Transformationssprache entwickelt werden. Dies bedeutet nach Abschnitt 5.4.2, dass zunächst eine abstrakte Syntax und die Definition der statischen Semantik (Metamodell der Sprache) erfolgen muss. Dabei soll sich an bestehenden Transformationssprachen orientiert und nutzbare Konzepte wiederverwendet werden. Aufbauend muss mindestens eine zugehörige konkrete Syntax entworfen werden. Den Abschluss bildet die Implementierung eines vollständigen Tool-Sets bestehend aus Parser, Checker und Interpreter, auch Transformationssystem genannt. In den nächsten Kapiteln wird das Metamodell der neu entwickelten Modelltransformationssprache vorgestellt, eine Abbildung für Verwaltungsschalen gegeben und eine Implementierung eines zugehörigen Transformationssystems als Proof-of-Concept beschrieben.

9 Metamodell der Modelltransformationssprache

Im vorherigen Kapitel wurden die Anforderungen an eine Modelltransformationssprache beschrieben und aufgezeigt, dass derzeit keine passende Sprache existiert. Aufgrund dessen wird in diesem Kapitel das Metamodell einer neuen Modelltransformationssprache definiert, welche allen Anforderungen gerecht wird. Für die Definition des Metamodells werden die benötigten Sprachelemente und deren Semantik beschrieben, welches in Abschnitt 9.1 erfolgt. Zusätzlich sind für die Sprachelemente zugehörige Syntaxregeln notwendig. Außerdem wird eine konkrete Syntax für eine Anwendung der Sprache benötigt. Die Regeln und die Syntaxdarstellungen werden in Abschnitt 9.2 vorgestellt. Abschließend wird in Abschnitt 9.3 gezeigt, dass die Sprache den Anforderungen aus Abschnitt 8.3 gerecht wird.

9.1 Benötigte Sprachelemente und deren Semantik

Viele der notwendigen Sprachelemente aus Abschnitt 8.3 erfordern die Möglichkeit, Ausdrücke (Expressions) zu formulieren. Zum Beispiel müssen Definitionen von Literalen (z. B. String oder Boolean), Fallunterscheidungen, Typüberprüfungen, Schleifen sowie die Verwendung von Variablen ermöglicht werden. Die Ausdruckssprache OCL ist ein Bestandteil der etablierten UML (vgl. Kapitel 3) und unterstützt bereits viele dieser Sprachelemente. Die Sprache baut dabei auf der Prädikatenlogik auf und erweitert diese. Die Beschreibung der Sprachelemente erfolgt durch natürliche Sprache, wodurch auch Nicht-Mathematiker oder Nicht-Informatiker die Sprachelemente verstehen und anwenden können. Zudem ist OCL eine deklarative Sprache. Da heutige Metamodelle meistens in UML spezifiziert werden (hier z. B. das Metamodell der Verwaltungsschale), OCL eine anerkannte Sprache sowie die Basis für viele Transformationssprachen ist und bereits viele der benötigten Sprachelemente unterstützt, wird diese als Grundlage für die neue Modelltransformationssprache verwendet.

BasicOCL erfüllt bereits fast alle Anforderungen aus Abschnitt 8.3. Lediglich das Anlegen von neuen Objekten sowie die Definition von Makros ist mit OCL nicht möglich (Anforderung 8.5). Aufgrund dessen werden neue Sprachelemente für diese Aufgaben benötigt. Zudem wird ein Sprachelement für die Definition der Elemente einer Transformationsdefinition gebraucht.

In Abbildung 9.1 ist das Metamodell der neuen Modelltransformationssprache dargestellt, welches auf dem veröffentlichten Metamodell in [21] basiert und weiterentwickelt wurde. Alle Ausdrücke von BasicOCL werden für die neue Modelltransformationssprache wiederverwendet. Diese sind in der Abbildung weiß dargestellt. Zusätzlich wurden die Klassen

Package, *TransformationDefiniton*, *Macro*, *MacroCallExp*, *ObjectLiteralExp* und *AttributeBinding* hinzugefügt (in der Abbildung grau dargestellt).

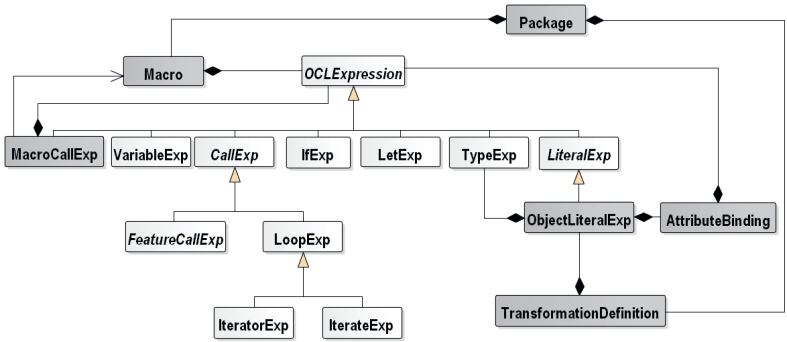

Abbildung 9.1: Metamodell der Modelltransformationssprache nach [21]

Eine Semantikbeschreibung der OCL-Ausdrücke findet sich in Kapitel 3. Um die Semantik der neuen Klassen zu verdeutlichen, ist eine detaillierte Sicht auf die hinzugefügten Elemente und deren Assoziationen in Abbildung 9.2 dargestellt.

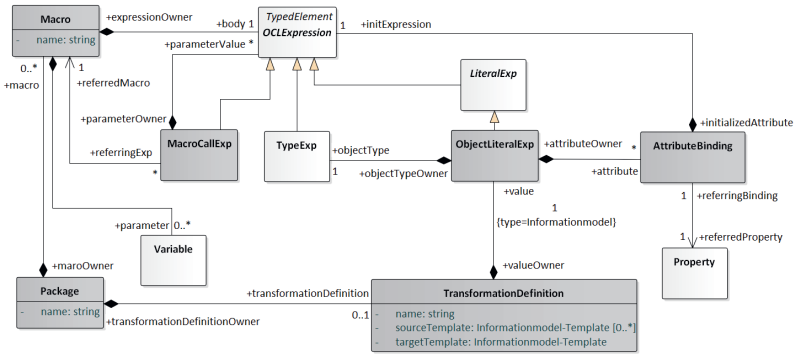

Abbildung 9.2: Detaillierte Ansicht des Metamodells der Modelltransformationssprache nach [21] inkl. der Assoziationen

Das Element *TransformationDefinition* stellt den Eingangspunkt für jede Transformationsdefinition dar. Um festzulegen, für welche Informationsmodell-Templates die Transformationsdefinition gilt, müssen sowohl die Templates der benötigten Quell-Informationsmodelle als auch das des Ziel-Informationsmodells angegeben werden. Dies erfolgt in den Attributen *sourceTemplate* und *targetTemplate*. Über das Attribut *value* wird die eigentliche Transformation durch eine *ObjectLiteralExp* formuliert. Diese wird ausgewertet und erstellt als Rückgabewert das zu erzeugende Informationsmodell.

Um Elemente innerhalb der Informationsmodell-Instanz zu erstellen, reichen die OCLExpressions, im Besonderen die *LiteralExp*, nicht aus. Mit diesen können lediglich Literale von einfachen Datentypen, wie Strings oder Booleans, erstellt werden. Aufgrund dessen wird die Klasse *ObjectLiteralExp* definiert. Diese ermöglicht die Erstellung von neuen Instanzen einer konkreten Modellklasse des Informationsmodell-Metamodells. Die ObjectLiteralExp ist eine Unterklasse der LiteralExp und besteht aus einer *TypeExp* über die *objectType*-Assoziation und einer beliebigen Anzahl von *AttributeBinding*s. Mit der TypeExp wird die Definition der Modellklasse des zu erstellenden Objekts vorgenommen. Die Klasse AttributeBinding ermöglicht das Setzen von Attributen der neu erstellten Instanz. Jeder ObjectLiteralExp können beliebig viele AttributeBinding-Elemente hinzugefügt werden, die jeweils die Definition eines Attributwerts ermöglichen. Die AttributeBinding-Elemente entsprechen dabei den Attributen der Objekt-Klasse. Bei der Ausführung einer ObjectLiteralExp wird genau eine neue Objektinstanz erstellt und zurückgegeben. An jeder Stelle, an der ein mit einer Objekt-Klasse typisierter Ausdruck erwartet wird, können Objekte der neu eingeführten Klasse verwendet werden.

Jedes AttributeBinding-Element ist Teil einer ObjectLiteralExp und erlaubt die Definition eines Attributs des zu instanziierenden Objekts. Es enthält einen Verweis auf die Klasseneigenschaft, die das Attribut spezifiziert. Zusätzlich muss ein Ausdruck als *initExpression* angegeben werden, um den Wert des Attributs zu initialisieren. Der Typ der *initExpression* muss mit dem Typ der Klasseneigenschaft übereinstimmen.

Um die Wiederverwendung von komplexen Ausdrücken zu ermöglichen und diese durch die Übergabe von Parameterwerten zu parametrisieren, wird die Klasse *Macro* eingefügt. Viele Modelltransformationssprachen nutzen ähnliche Konzepte, z. B. Helpers in ATL [99]. Beispiele für Makros sind das vollständige Kopieren eines Objekts oder das vereinfachte Setzen eines Attributs[1]. Die Klasse *Macro* formuliert eine beliebige OCLExpression als *body* und kann beliebig viele Variablen als Parameter definieren, die im Ausdruck Verwendung finden können. Das Macro-Element kann entweder zusammen mit der Transformationsdefinition in einem Paket oder ohne eine solche (z. B. zur allgemeinen Verwendung in einer Bibliothek) enthalten sein. Die Klasse für die Bibliothek ist in Abbildung 9.2 nicht dargestellt.

Um Makros aufzurufen, wird die Klasse *MacroCallExp* eingeführt. Diese ist eine Unterklasse der OCLExpression, die sich bei einem Aufruf zum Body-Ausdruck des referenzierten Makros auflöst. Sie kann mehrere OCLExpressions als *parameterValue* enthalten, die bei Auswertung den Parametervariablen des referenzierten Makros zugewiesen werden. Hiermit kann die Auswertung des Ausdrucks parametrisiert werden.

Für eine effiziente Sicherung der Transformationsdefinitionen und Makros wird zusätzlich die Klasse *Package* eingeführt. Jedes Objekt dieser Klasse besteht aus maximal einer Transformationsdefinition und kann beliebig viele Makros enthalten. Es ist auch möglich, dass ausschließlich Makros enthalten sind.

[1] In Kapitel 10 sind einige Makros für das Metamodell der Verwaltungsschale beschrieben.

9.2 Syntaxregeln und konkrete Syntax

Für eine vollständige Sprachspezifikation der Modelltransformationssprache müssen zusätzlich Regeln für die Wohlgeformtheit der in Abschnitt 9.1 beschriebenen Sprachelemente spezifiziert werden. Dadurch wird eine Überprüfung der Gültigkeit und Typkonsistenz von Transformationsdefinitionen ermöglicht. Diese können durch Invarianten mit OCL-Sprachelementen formuliert werden. Um die Sprache zu nutzen, wird eine konkrete Syntax benötigt. Dies erfolgt in Form von Produktionsregeln.

Für die wiederverwendeten Sprachelemente aus OCL werden die Syntaxregeln und die Produktionsregeln der konkreten Syntax aus der OCL-Spezifikation [53] genutzt. Für die zusätzlichen Elemente werden folgende Invarianten und Produktionsregeln definiert:

Informationmodel: Da die Sprache unabhängig eines konkreten Metamodells (generisch) spezifiziert ist, muss die Klasse Informationsmodel bei der Anwendung der Sprache auf ein Metamodell konkretisiert werden. Beim Verwaltungsschalen-Metamodell ist dies beispielsweise die Klasse Submodel.

Macro: Ein Makro ermöglicht, wiederholt auftretende komplexe OCL-Ausdrücke wiederzuverwenden. Es enthält genau einen OCL-Ausdruck im Attribut *body*. Dieser Ausdruck kann weitere OCL-Ausdrücke beinhalten. Ein Makro ist typisiert und der Typ entspricht dem Typen des OCL-Ausdrucks im Attribut *body*. Dabei stellt ein Makro einen Namensraum zur Verfügung und ermöglicht die Definition von Parametern (Variablendeklarationen), die innerhalb des OCL-Ausdrucks verwendet werden. Ein Makro kann beliebig viele Parameter haben, jedoch müssen die Namen der Parameter eindeutig sein. Um diese bei einem Aufruf durch eine MacroCallExp mit Werten zu füllen, muss die Reihenfolge festgelegt sein, da auch Default-Werte unterstützt werden sollen. Der Name eines Makros muss wiederum in dem Namensraum, in dem dieser definiert wird, eindeutig sein. Für eine formale Überprüfung werden diese Syntaxregeln in Form von Invarianten festgelegt:

```
context Macro
  inv: self.name.type.oclIsKindOf(PrimitiveType)
  inv: self.name.type.name = 'String'
  inv: self.parameter.type.oclIsKindOf(SequenceType)
  inv: self.parameter->forAll(oclIsKindOf(Variable))
  inv: self.parameter->isUnique(name)
  inv: self.type = self.body.type
  inv: self.body.type.oclIsKindOf(OCLExpression)
```

Für die konkrete Syntax wird die Produktionsregel eines Makros wie folgt definiert: Die Produktionsregel besteht aus dem Terminalsymbol *Macro* gefolgt von einem frei definierbaren Namen, der durch das in OCL definierte Nichtterminalsymbol *simpleNameCS* [53] formuliert wird. Danach folgen in Klammern die Parameterdefinitionen. Diese bestehen jeweils aus dem Namen und dem Typen des Parameters (siehe Definition von parametersCS in [53]). Da Makros typisiert sind, kann durch einen Doppelpunkt getrennt, der Typ[2] des

[2]Als Typ wird der Rückgabewert verstanden, der bei der Auflösung der enthaltenen Expression erzeugt wird.

Makros durch das OCL Nichtterminalsymbol *typeCS* explizit angegeben werden. Zuletzt wird der komplexe OCL-Ausdruck innerhalb geschweifter Klammern festgelegt.

```
MacroDeclarationCS ::= 'Macro' simpleNameCS
                       '(' parametersCS? ')' (':' typeCS)?
                       '{' OCLExpressionCS '}'
```

MacroCallExp: Für das Aufrufen eines Makros wird die Klasse *MacroCallExp* eingeführt. Diese referenziert im Attribut *referredMacro* ein vorher definiertes Makro und wertet dieses aus. Der Ausdruck *MacroCallExp* kann, sofern das referenzierte Makro Parameter spezifiziert, eine Liste von Parameterausdrücken enthalten. Diese werden im Attribut *parameterValue* als eine Sequenz gespeichert. In diesem Fall müssen die Anzahl und die Typen der Parameterausdrücke mit denen der Parameter des Makros übereinstimmen. Die *MacroCallExp* ist eine Unterklasse der *OCLExpression* und kann folglich an einer beliebigen Stelle genutzt werden, an der ein OCL-Ausdruck benötigt wird, der dem Typen des referenzierten Makros entspricht. Bei der Auswertung wird das referenzierte Makro mit den übergebenen Parametern ausgewertet.

```
context MacroCallExp
  inv: self.type = self.referredMacro.type
  inv: self.parameterValue.type.oclIsKindOf(SequenceType)
  inv: self.parameterValue->forAll(
         oclIsKindOf(OCLExpression))
  -- Anzahl der ParameterValue und der Parameter des
     referenzierten Makros müssen gleich sein
  inv: self.parameterValue->size() =
         self.referredMacro.parameter->size()
  -- Typen der ParameterValue und der Parameter des
     referenzierten Makros müssen gleich sein
  inv: self.paramterValue->forAll(p | p.type.conformsTo
         (self.referredMacro.parameter->at
             (self.parameterValue->indexOf(p)).type))
```

Die Produktionsregel für die Nutzung einer MacroCallExp besteht aus dem Namen des Makros sowie dem zugehörigen Paket, sofern das Makro nicht im gleichen Paket wie die Transformationsdefinition spezifiziert wurde. Die Zuordnung zum Paket geschieht optional durch das OCL Nichtterminalsymbol *simpleNameCS* gefolgt von zwei Doppelpunkten. Anschließend wird der Name des Makros ebenfalls durch das OCL Nichtterminalsymbol *simpleNameCS* formuliert. Für diesen Fall existiert in OCL bereits das Nichtterminalsymbol *PathNameCS*. Im Anschluss folgen die Variablen in Klammern, die den Parametern des Makros zugewiesen werden. Diese werden durch das OCL Nichtterminalsymbol *argumentsCS* beschrieben und entsprechen der Syntax des Aufrufs einer Operation innerhalb eines Kontexts, wie in [53] spezifiziert.

```
MacroCallExpCS ::= PathNameCS '(' argumentsCS? ')'
```

TransformationDefinition: Jedes Objekt der Klasse TransformationDefinition hat einen Namen, der frei gewählt wird. Das Attribut *sourceTemplate* ist ein Set von Informationsmodell-Template-Definitionen, bestehend aus einem Variablennamen zur

weiteren Nutzung und einem möglichen Attribut-Wert zur Findung der zugehörigen Quellmodell-Instanzen. Jeder Variablenname sowie mögliche Attribut-Werte dürfen ausschließlich einmal verwendet werden, um Mehrfachzuweisungen zu verhindern. Das Attribut *targetTemplate* legt fest, für welches Informationsmodell-Template eine Instanz erstellt wird. Dafür wird ebenfalls eine Informationsmodell-Template-Definition angegeben. Als Ergebnis wird eine neue Instanz eines Informationsmodells erstellt, welche im Attribut *value* über eine ObjectLiteralExp formuliert wird. Diese Instanz muss dem Typ des *Informationmodel* entsprechen und die geforderten Constraints des Informationsmodell-Templates erfüllen, welches als *targetTemplate* angegeben wurde.

```
context TransformationDefinition
  inv: self.name.type.oclIsKindOf(PrimitiveType)
  inv: self.name.type.name = 'String'
  inv: self.value.type.oclIsKindOf(Informationmodel)
  inv: self.sourceTemplate.type.oclIsKindOf(Set)
  inv: self.sourceTemplate->forAll(
         oclIsKindOf(InformationmodelTemplate))
```

Für die Definition einer Transformationsdefinition wird die Produktionsregel wie folgt definiert: Sie wird durch das Terminalsymbol *TransformationDefinition* eingeleitet und besitzt einen eindeutigen Namen, der über das OCL Nichtterminalsymbol *simpleNameCS* angegeben wird. Danach folgen optional die Definitionen der Quell-Informationsmodell-Templates. Diese werden durch das Terminalsymbol *sourceTemplate*, gefolgt von einem Doppelpunkt sowie dem Nichtterminalsymbol *InformationmodelTemplateListCS* formuliert. Zwingend anzugeben ist das Ziel-Informationsmodell-Template, welches durch das Terminalsymbol *targetTemplate*, gefolgt von einem Doppelpunkt und der Nutzung des Nichtterminalsymbols *InformationmodelTemplateCS* erfolgt. Weiterhin ist die Angabe des Terminalsymbols *value* gefolgt von einem Doppelpunkt und einer ObjectLiteralExp durch die zugehörige Produktionsregel *ObjectLiteralExpCS* verpflichtend.

```
TransformationDefinitionCS ::=
    'TransformationDefinition' simpleNameCS
    'sourceTemplates' ':' InformationmodelTemplateListCS?
    'targetTemplate' ':' InformationmodelTemplateCS
    'value' ':' ObjectLiteralExp
```

InformationmodelTemplate: Die Definition eines Informationsmodell-Templates dient der Ermittlung der zugehörigen Quellmodelle für die weitere Nutzung in der Transformationsdefinition. Die Definition besitzt eine Variable, über die auf Instanzen dieses Templates in der Transformationsdefinition zugegriffen wird. Der Name muss innerhalb der Transformationsdefinition eindeutig sein. Zusätzlich muss der Typ der Quellmodelle angegeben werden. Dieser Typ kann entweder der konkrete Informationsmodell-Template-Typ oder eine Sammlung eines Informationsmodell-Template-Typs sein, sofern mehrere Informationsmodell-Instanzen des gleichen Typs verwendet werden. Der Initialwert darf nicht gesetzt werden, da er nicht benötigt wird. Danach kann in der Produktionsregel das Terminalsymbol -> gefolgt von einer *LiteralExpression* genutzt werden. Diese LiteralExpression stellt den Verweis auf ein Informationsmodell-Template dar, so-

fern die Informationsmodell-Templates alle vom gleichen Typ sind[3]. In diesem Fall ist das Ergebnis der Literal Expression der Attribut-Wert, der für die Unterscheidung der Informationsmodell-Templates im spezifischen Metamodell spezifiziert ist. Für die Definition einer Liste von Informationsmodell-Templates wird eine zusätzliche Produktionsregel definiert:

```
InformationmodelTemplateCS ::=
    VariableDeclarationCS ('->' LiteralExpCS)?

InformationmodelTemplateListCS[1] ::=
    InformationmodelTemplateCS
    (',' InformationmodelTemplateListCS[2])?
```

ObjectLiteralExp: Die Object Literal Expression dient dem Erstellen eines neuen Objekts aus dem Metamodell. Der Typ dieses Objekts ist der referenzierte Typ der OCL-Expression im objectType. Dieser muss folglich ein Subtyp der Klasse *Class* sein. Die Klasse Class ist der Obertyp aller Klassen des zugehörigen UML Metamodells. Zusätzlich kann eine ObjectLiteralExp eine beliebige Anzahl von AttributBinding-Objekten enthalten. Jedes dieser AttributeBinding-Objekte referenziert dabei genau ein Attribut des zu erstellenden Objekttyps. Zwei AttributeBinding-Objekte dürfen nicht das gleiche Attribut referenzieren.

```
context ObjectLiteralExp
  inv: self.objectType.referredType.oclIsKindOf(Class)
  inv: self.type = self.objectType.referredType
  -- AttributBindings müssen genau ein Attribut des
     referenzierten Objekttypes referenzieren
  inv: self.attribute->forAll(a |
         self.objectType.property->exists(
           a.referredProperty))
  -- AttributBindings dürfen nicht das gleiche Attribut des
     referenzierten Objekttypes referenzieren
  inv: self.attribute->isUnique(referredProperty)
```

Die Definition einer Object Literal Expression beginnt mit dem OCL Nichtterminalsymbol *typeCS*, welches den Typen des zu erstellenden Objekts spezifiziert, gefolgt von optionalen AttributeBinding-Objekten in geschweiften Klammern. Dabei können kein, ein oder mehrere AttributeBinding-Objekte durch das zugehörige Syntaxelement *AttributeBindingListCS* formuliert werden.

```
ObjectLiteralExpCS ::=
    typeCS '{' AttributeBindingListCS? '}'
```

Da die Object Literal Expression eine Spezialisierung der OCL *LiteralExp* ist, muss diese der Produktionsregel für LiteralExp hinzugefügt werden:

```
[F] LiteralExpCS ::= ObjectLiteralExpCS
```

[3]Dies ist z. B. bei den Submodel-Templates im Metamodell der Verwaltungsschale der Fall, die alle vom Typ Submodel sind. Hier wird die Unterscheidung über das Attribut *SemanticId* getätigt.

AttributeBinding: Jedes AttributeBinding-Objekt ist genau einer Object Literal Expression zugewiesen und bezieht sich auf ein Attribut des zu erstellenden Objekttyps. Auf dieses Attribut verweist das AttributeBinding-Objekt über die Assoziation *referredProperty*. Zusätzlich wird über die Assoziation *initExpression* der Wert des Attributs festgelegt. Der Typ der *initExpression* muss dem Typen des referenzierten Attributs entsprechen.

```
context AttributeBinding
  inv: self.initExpression.type = self.referredProperty.type
```

Die Produktion eines AttributeBinding-Elements beginnt mit der Namensdefinition des referenzierten Attributs durch das OCL Nichtterminalsymbol *simpleNameCS*. Danach folgt, durch einen Doppelpunkt getrennt, die Initialisierung des Wertes. Dafür kann ein beliebiger OCL Ausdruck genutzt werden. Die vollständige Produktionsregel ist daher wie folgt definiert:

```
AttributeBindingCS ::= simpleNameCS ':' OCLExpressionCS
```

Um AttributeBinding-Elemente in einer Liste zusammenzufassen, wird eine weitere Produktionsregel benötigt. Diese besteht aus der Nutzung des AttributeBindingCS für die Definition eines AttributeBinding-Elements, gefolgt von einer optionalen Gruppe bestehend aus einem Komma und dem rekursiven Aufruf dieser Produktionsregel. Damit können beliebig viele AttributeBindingCS durch ein Komma getrennt angehängt werden.

```
AttributeBindingListCS[1] ::=
    AttributeBindingCS (',' AttributeBindingListCS[2])?
```

Package: Ein Package dient der Strukturierung von maximal einer Transformationsdefinition und beliebig vielen Makros. Jedes Package hat einen Namen, der bei der Nutzung von mehreren Packages eindeutig sein muss. Ein Package kann maximal ein Objekt der Klasse *TransformationDefinition* beinhalten sowie eine beliebige Anzahl von Objekten der Klasse *Macro*. Die Namen der enthaltenen Makros müssen dabei unterschiedlich sein.

```
context Package
  inv: self.name.type.oclIsKindOf(PrimitiveType)
  inv: self.name.type.name = 'String'
  inv: self.transformationDefiniton->size() <= 1
  inv: self.transformationDefinition.type.oclIsKindOf(
         TransformationDefinition)
  inv: self.macro->forAll(oclIsKindOf(Macro))
  inv: self.macro->isUnique(name)
```

Die Definition eines Packages beginnt durch das Terminalsymbol *package* gefolgt von einem eindeutigen Namen, der über das Nichtterminalsymbol *simpleNameCS* formuliert wird. Danach folgt optional eine Transformationsdefinition mittels des Nichtterminalsymbols *TransformationDefinitionCS* sowie eine Liste des Nichtterminalsymbols *MacroDeclarationCS* für die Definition der enthaltenen Makros.

```
PackageCS ::=
    'package' simpleNameCS
    TransformationDefinitionCS?
    MacroDeclarationCS*
```

9.3 Evaluation der Sprache

Mit Hilfe der neu definierten Sprache kann die Transformation zwischen mehreren Eingangs-Informationsmodellen hin zu einem Ziel-Informationsmodell formuliert werden. Als Basis wird BasicOCL verwendet und um zwingend notwendige Sprachelemente erweitert, die zum Teil bereits in anderen Transformationssprachen integriert sind. Dazu wurden Konzepte anderer Transformationssprachen analysiert und für diesen Anwendungsfall sinnvoll angepasst und übernommen, wie z. B. Makros. Die neue Sprache basiert somit auf dem aktuellen Stand der Wissenschaft und Industrie, und ist durch die Definition von lediglich notwendigen Sprachelementen leicht zu implementieren (Anforderung 8.4). Mit Hilfe der OCL-Ausdrücke können Elemente in den Quell-Informationsmodellen gefunden (Anforderung 8.7) und deren Attribute ausgelesen werden (Anforderung 8.8). Zusätzlich definiert OCL bereits Ausdrücke, mit denen Fallunterscheidungen definiert (Anforderung 8.9), Schleifen bzw. Iterationen durchgeführt (Anforderung 8.10) sowie Variablen angelegt werden können (Anforderung 8.11). Ziel der neuen Sprache ist, ein wohlgeformtes Informationsmodell zu erzeugen, welches gegen konkrete Invarianten von Informationsmodell-Templates geprüft werden kann. Die Sprachelemente von OCL ermöglichen keine Objekt-Erstellung. Aufgrund dessen wurden die beiden Klassen *ObjectLiteralExp* und *AttributeBinding* eingeführt, die die Erstellung von Informationsmodellelemente des Metamodells ermöglichen (Anforderung 8.5). Bei der Erstellung wird die Sprachsyntax des Metamodells genutzt, sodass der Anwender die Transformationsdefinitionen leichter erstellen kann. Durch die Klasse *AttributeBinding* wird sichergestellt, dass alle Attribute eines Objekts setzbar sind. Dies ist mit den Sprachelementen von OCL allein nicht möglich (Anforderung 8.6). Für die vereinfachte Wertdefinition beim Setzen können wiederum die in OCL definierten Operationen (z. B. String-Konkatenation, Addition von Zahlenwerten) genutzt werden. Abschließend ermöglichen die neu eingeführten Klassen *Macro* und *MacroCallExp* das Erstellen von Makros sowie deren Nutzung (Anforderung 8.12). Es wurden bewusst wenige Sprachelemente spezifiziert, sodass die Sprache einfach zu verstehen ist und dadurch eine bessere Verbreitung findet (Anforderung 8.1). Zudem wurde, im Gegensatz zu vielen bestehenden Transformationssprachen, nicht das Mapping zwischen Objekttypen betrachtet, sondern die Erstellung genau eines Ziel-Informationsmodells fokussiert (Anforderung 8.2). Alle Sprachelemente sind so formuliert, dass das neue Ziel-Informationsmodell deklarativ beschrieben wird (Anforderung 8.3). Die neu definierte Sprache erfüllt somit alle geforderten Anforderungen aus Kapitel 8.

10 Abbildung der Modelltransformationssprache für Verwaltungsschalen

Die in Kapitel 9 vorgestellte Modelltransformationssprache wird für die Nutzung von Verwaltungsschalen angepasst. Dafür werden das Informationsmodell und die Referenzierung zu Informationsmodell-Templates festgelegt sowie Makros für die einfachere Nutzbarkeit definiert.

10.1 Anpassungen des Informationsmodells

Im Metamodell der Verwaltungsschale werden Informationsmodelle durch die Klasse *Submodel* beschrieben. Das angepasste Metamodell der Sprache ist in Abbildung 10.1 dargestellt.

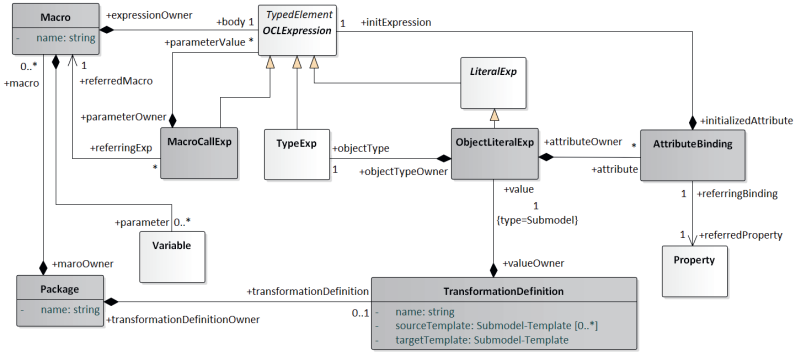

Abbildung 10.1: Detaillierte Ansicht des Metamodells der Modelltransformationssprache inkl. der Assoziationen nach [21] angepasst für Verwaltungsschalen

Die Referenzierung einer Submodel-Instanz auf das zugehörige Submodel-Template erfolgt über das Attribut *semanticId*. Die Klasse Referenz (*Reference*) besteht aus einer geordneten Liste von Instanzen der Klasse *Key*, die drei Attribute besitzt:

- *type:* Typ der referenzierten Entität, z. B. globale Referenz oder Referenz auf eine Instanz einer Metamodell-Klasse

- *value:* Wert der Referenz, z. B. eine IRDI
- *idType:* Weitere Unterscheidung zwischen IRI, IRDI, Custom, IdShort oder FragementId

Eine Referenz auf ein Submodel-Template beginnt mit der Definition des Variablennamens, gefolgt vom Typ *Submodel* oder *Set(Submodel)*. Anschließend wird das Submodel-Template durch eine ObjectLiteralExp vom Typ *Reference* referenziert:

```
a: Submodel -> Reference{key: Sequence{Key{
                  idType: KeyType::IRI,
                  value: "http://acplt.org/ExampleSMT",
                  type: KeyElements::GlobalReference}}}
```

Ein Beispiel für eine einfache Transformationsdefinition mit den definierten Sprachelementen sieht wie folgt aus:

```
transformationDefinition td1
  sourceTemplate:
    a: Submodel -> Reference{key: Sequence{Key{
                    idType: KeyType::IRI,
                    value: "http://acplt.org/ExampleSMT",
                    type: KeyElements::GlobalReference}}},

    b: Submodel -> Reference{key: Sequence{Key{
                    idType: KeyType::IRI,
                    value: "http://acplt.org/ExampleSMT_2",
                    type: KeyElements::GlobalReference}}}
  targetTemplate:
    c: Submodel -> Reference{key: Sequence{Key{
                    idType: KeyType::IRI,
                    value: "http://acplt.org/ExampleSMT_3",
                    type: KeyElements::GlobalReference}}}
  value: Submodel {
    identification: Identifier{
        id: "https://acplt.org/Test_Submodel",
        idType: IdentifierType::IRI},
    submodelElement: copySubmodelElementSet(
      a.submodelElement)->union(
        copySubmodelElementSet(b.submodelElement)),
    semanticId: Reference{key: Sequence{Key{
                  idType: KeyType::IRI,
                  value: "http://acplt.org/ExampleSMT_3",
                  type: KeyElements::GlobalReference}}}
  }
```

Die Transformationsdefinition hat den Namen *td1*. Sie formuliert die Transformation zweier existierender Teilmodell-Instanzen in eine neue Teilmodell-Instanz. Die Quell-Templates sind vom Typ *Submodel* und werden durch die Reference-Deklarationen *a* und *b*, die nach

dem Terminalsymbol *sourceTemplate* folgen, festgelegt. Analog erfolgt der Verweis auf das Ziel-Template. Die eigentliche Transformationsregel wird nach dem Terminalsymbol *value* spezifiziert. Durch eine *ObjectLiteralExp* wird eine neue Teilmodell-Instanz vom Typ Submodel erstellt und die entsprechenden Attribute gesetzt. Die neue Teilmodell-Instanz erhält die Kind-Elemente von beiden Quell-Teilmodell-Instanzen (Attribut *submodelElement*) sowie eine Referenz auf das zugehörige Teilmodell-Template aus der Variable *c* (Attribut *semanticId*). Für die Zuweisung des Attributs *submodelElement* werden *MacroCallExp* genutzt.

Im Kontext von Verwaltungsschalen werden einige Funktionalitäten häufiger benötigt. Diese können in Makros gekapselt und dem Anwender zur Verfügung gestellt werden. Aus diesem Grund werden Makros für folgende Funktionalitäten definiert:

1. Das vollständige Kopieren von einzelnen *SubmodelElement*-Objekten
2. Das vollständige Kopieren eines Sets aus *SubmodelElement*-Objekten
3. Der Zugriff auf ein *SubmodelElement*-Objekt basierend auf dem Attribut *IdShort*
4. Der Zugriff auf ein *SubmodelElement*-Objekt basierend auf einem Pfad aus *IdShort*s
5. Der Zugriff auf ein oder mehrere *SubmodelElement*-Objekte basierend auf dem Attribut *SemanticId*
6. Das vollständige Kopieren von *SubmodelElement*-Objekten basierend auf dem Attribut *IdShort*, welches eine Kombination des zweiten und dritten Makros ist
7. Das vollständige Kopieren von *SubmodelElement*-Objekten basierend auf einem Pfad aus *IdShort*s, welches eine Kombination des zweiten und vierten Makros ist
8. Das vollständige Kopieren von *SubmodelElement*-Objekten basierend auf dem Attribut *SemanticId*, welches eine Kombination des zweiten und fünften Makros ist

Die Makros werden mit den Sprachelementen der Transformationssprache aus Kapitel 9 formuliert, sodass keine weitere Auswertelogik von Seiten des Transformations-Tools benötigt wird. Die vollständige Auflistung der definierten Makros ist in Anhang A gegeben. In den beiden folgenden Unterkapiteln werden die wesentlichen Inhalte der Makros für das vollständige Kopieren und dem Zugriff auf *SubmodelElement*-Objekte vorgestellt.

10.2 Makros für das vollständige Kopieren von SubmodelElement-Objekten

Sofern die Informationen und die Modellierung der Elemente im Quell- und Ziel-Teilmodell-Template identisch sind, können Objekte vollständig kopiert werden. Aus diesem Grund wird für jede SubmodelElement-Klasse ein entsprechendes Makro spezifiziert. Der Aufbau dieser Makros ist immer identisch. Als Übergabeparameter wird das zu kopierende Objekt übergeben und in der Variable *element* gespeichert. Innerhalb des Makros wird eine *ObjectLiteralExp* formuliert, die ein neues Objekt der entsprechenden Klasse erstellt. Innerhalb der *ObjectLiteralExp* werden mit Hilfe von *AttributeBinding*-Elementen die Attribute des

neuen Objekts mit den Attribut-Werten des übergebenen Objekts gesetzt. Nachfolgend ist das Makro für das Kopieren eines *Property*-Objekts als Beispiel dargestellt:

```
macro copyProperty(element: Property) : Property{
  Property {
    idShort: element.idShort,
    valueType: element.valueType,
    value: element.value,
    valueId: element.valueId,
    displayName: element.displayName,
    category: element.category,
    description: element.description,
    semanticId: element.semanticId,
    kind: element.kind
  }
}
```

Falls die Modellklasse nicht bekannt ist, kann das Makro *copySubmodelElement* genutzt werden, welches die Klasse des übergebenen Objekts analysiert und das entsprechende Makro für die jeweilige Klasse aufruft.

Einen Sonderfall stellen die SubmodelElement-Klassen *SubmodelElementCollection*, *AnnotatedRelationshipElement* und *Entity* dar, da in ihren Attributen *value*, *annotation* oder *statement* weitere SubmodelElement-Objekte als Set enthalten sein können. Aus diesem Grund wird das Makro *copySubmodelElementSet* definiert, welches ein Set von SubmodelElement-Objekten als Übergabeparameter erhält. Über dieses wird iteriert, für jedes enthaltene Objekt das Makro *copySubmodelElement* aufgerufen, ein neues Set mit diesen Elementen erstellt und dieses Set zurückgegeben.

In den Anwendungsfällen hat sich gezeigt, dass *SubmodelElementCollection*-Objekte häufig nicht vollständig kopiert werden sollen, sondern die enthaltenen *SubmodelElement*-Objekte im Attribut *value* variieren können, und z. B. lediglich eine gewisse Auswahl der Elemente übernommen wird. Aus diesem Grund wird das Makro *copySubmodelElementCollectionWithValue* eingeführt, welches ein *SubmodelElementCollection*-Objekt sowie ein Set von *SubmodelElement*-Objekten übergeben bekommt. Im Makro wird ein neues *SubmodelElementCollection*-Objekt erzeugt, welches alle Attributwerte aus dem übergebenen *SubmodelElementCollection*-Objekt kopiert, jedoch dem Wert des Attributs *value* eine Kopie des übergebenen Sets zuweist.

10.3 Makros für den Zugriff auf ein SubmodelElement-Objekt

Für den Zugriff auf Objekte des Quell-Teilmodells können drei Arten unterschieden werden: Zugriff mittels des Attributs *IdShort*, eines Pfads aus *idShorts* und des Attributs *SemanticId*.

Durch das Verwaltungsschalen-Metamodell ist vorgegeben, dass die eindeutige Identifikation über das Attribut *IdShort* erfolgt, da die IdShort eines Objekts im Namensraum des Vater-Objekts eindeutig sein muss. Daher wird das Makro *getSubmodelElementByIdShort* spezifiziert. Dieses erhält als Übergabeparameter das Vater-Objekt sowie die IdShort des gesuchten Kind-Objekts, welche vom Typ String ist. Das Metamodell definiert keine Funktion zur Auflösung einer IdShort. Aus diesem Grund wird zunächst die Klasse des Vater-Objekts analysiert und je nach Klasse wird anschließend auf das entsprechende Klassen-Attribut zugegriffen, in dem weitere Elemente enthalten sein können, z. B. *annotation* oder *statement*. Auf dieses Attribut wird dann die OCL-Operation *select* angewendet, die über die enthaltenen Objekte iteriert und jedes Element zurückgibt, welches die übergebene IdShort besitzt. Als Ergebnis wird eine OCL-Collection erzeugt, die in diesem Fall kein oder ein Element enthalten kann. Über die Funktion *first* wird das erste Element dieser Collection zurückgegeben. Konnte kein Element mit dieser IdShort gefunden werden — enthält die Liste also kein Element — wird das OCL-Objekt *Invalid* zurückgegeben. Ein Ausschnitt des Makros ist nachfolgend abgebildet:

```
macro getSubmodelElementByIdShort(
  parent: SubmodelElement,
  idShortVar: String) : SubmodelElement{
    if parent.oclIsKindOf(Submodel)
    then parent.submodelElement->select(x |
      x.idShort = idShortVar)->first()
    else
      ...
    endif
}
```

Da Kind-Objekte wiederum Kind-Objekte enthalten können, wird durch die IdShorts eine Baumstruktur aufgebaut. Mit Hilfe des Makros *getSubmodelElementByIdShortPath* kann auf ein beliebiges Objekt in dieser Struktur zugegriffen werden. Dafür wird das Vater-Objekt, bei dem gestartet werden soll und der Pfad zum Kind-Objekt mittels einer Auflistung von IdShorts an das Makro übergeben. Dabei muss die Reihenfolge der IdShorts korrekt sein, weswegen der OCL-Datentyp *Sequence* genutzt wird. Das Makro iteriert anschließend mit der OCL-Funktion *iterate* über die IdShort-Auflistung und sucht für die aktuelle IdShort im aktuellen Vater-Objekt das zugehörige Kind-Objekt. Danach wird das Vater-Objekt mit dem Kind-Objekt überschrieben, sodass das Kind-Objekt zum neuen Vater-Objekt wird, innerhalb dessen wieder das neue Kind-Objekt gesucht wird. Dies wird so lange wiederholt, bis die Auflistung vollständig abgearbeitet wurde. Schließlich wird das letzte Kind-Objekt zurückgegeben. Nachfolgend ist die Definition des Makros abgebildet:

```
macro getSubmodelElementByIdShortPath(
        parent: SubmodelElement,
        idShortPath: Sequence(String)) : SubmodelElement{
  idShortPath->iterate(
    x: String; sme: SubmodelElement = parent |
      getSubmodelElementByIdShort(sme, x))
}
```

Die dritte Möglichkeit ist der Zugriff über das Attribut *SemanticId* mit Hilfe des Makros *getSubmodelElementsBySemanticId*. Anders als bei der IdShort sind Elemente über die SemanticId nicht zwingend in ihrem Vater-Objekt eindeutig. Aufgrund dessen kann nicht genau ein Element, sondern lediglich eine Sammlung von Objekten zurückgegeben werden. Analog zum Makro *getSubmodelElementByIdShort* wird zunächst der Objekttyp des Vaters geprüft und anschließend auf dem entsprechenden Attribut die OCL-Funktion *select* ausgeführt. Die erstellte Sammlung der *select*-Funktion wird anschließend zurückgegeben. Entsprechend sieht das Makro wie folgt aus:

```
macro getSubmodelElementsBySemanticId(parent:
   SubmodelElement, semanticId: Reference) : Set(
   SubmodelElement){
  if parent.oclIsKindOf(Submodel) then parent.
     submodelElement->select(x | x.semanticId = semanticId)
  else
    if parent.oclIsKindOf(Entity) then parent.statement->
       select(x | x.semanticId = semanticId)
    else
      if parent.oclIsKindOf(SubmodelElementCollection) then
         parent.value->select(x | x.semanticId = semanticId)
      else
        if parent.oclIsKindOf(AnnotatedRelationshipElement)
           then parent.annotation->select(x | x.semanticId =
           semanticId)
        else invalid
        endif
      endif
    endif
  endif
}
```

In vielen Fällen besteht die SemanticId lediglich aus einem *Key*-Objekt und die Werte für die Attribute *idType* und *type* sind *KeyType::IRI* und *KeyElements::GlobalReference*. Aufgrund dessen wird ein weiteres Makro *getSubmodelElementsBySemanticIdValue* eingeführt, welches den Wert für das Attribut *value* dieses *Key*-Objekts übergeben bekommt, anstelle eines vollständigen *Reference*-Objekts. Im Makro wird zunächst das benötigte *Reference*-Objekt erstellt und anschließend das Makro *getSubmodelBySemanticId* aufgerufen:

```
macro getSubmodelElementsBySemanticIdValue(
  parent: SubmodelElement,
  semanticIdValue: String) : Set(SubmodelElement){
    getSubmodelElementsBySemanticId(
      parent,
      Reference{key: Sequence{Key{
        idType: KeyType::IRI,
        value: semanticIdValue,
        type: KeyElements::GlobalReference}}})
}
```

11 Transformationssystem

In den Kapiteln 9 und 10 wurde die Transformationssprache vorgestellt, mit der Transformationsdefinitionen zwischen Quell- und Ziel-Informationsmodell-Templates formuliert werden können. Für die Ausführung einer Transformationsdefinition mit konkreten Informationsmodell-Instanzen wird ein Transformationssystem benötigt (vgl. Abbildung 5.1 in Kapitel 5). In den folgenden Abschnitten erfolgt zunächst eine grobe Vorstellung der Hauptfunktionalitäten des Transformationssystems und basierend darauf eine Beschreibung einer konkreten Umsetzung in Python.

11.1 Allgemeiner Aufbau eines Transformationssystems

Das in dieser Arbeit benötigte Transformationssystem muss drei Funktionen zur Verfügung stellen:

1. Erstellung des abstrakten Syntaxbaums einer Package-Definition
2. Erstellung eines ausführbaren abstrakten Syntaxbaums[1]
3. Ausführung des abstrakten Syntaxbaums auf konkrete Quellmodelle

In der Regel wird für die **Erstellung des abstrakten Syntaxbaums** ein Parser verwendet. Dieser erhält als Eingabe die zu analysierenden Dateien, in denen Transformationsdefinition und Makros sowie die Grammatik-Definition der Transformationssprache (konkrete Syntax in Form von Produktionsregeln) enthalten sind. Für die Analyse der Dateien nutzt der Parser einen Lexer, der den gegebenen Text bzw. Quellcode in Token zerlegt. Ein Token ist eine Zeichenkette, die einem der in der Grammatik definierten Terminalen zugeordnet werden kann. Zum Beispiel kann die Zeichenfolge „abcd“ dem Terminal *SimpleNameCS* zugewiesen werden. Mit diesem Ergebnis erstellt der Parser einen Parserbaum. Dazu können verschiedene Methoden genutzt werden, wie z. B. Top Down oder Bottom Up [158]. Beim Top-Down-Parsing startet der Parser beim Startsymbol[2] und versucht eine Produktionsregel für die gegebene Zeichenkette zu finden. Anschließend wird das nächste nicht zugeordnete Symbol betrachtet und die nächste Produktionsregel gesucht. Im Gegensatz dazu startet das Bottom-Up-Parsing bei einem Token auf der untersten Ebene[3] und versucht die Zusammenhänge zu den anderen Token zu ermitteln. Der Parser erstellt aus diesen Zusammenhängen schließlich den Parserbaum. Der finale Parserbaum besteht zunächst aus den Terminalnamen. Um daraus einen abstrakten Syntaxbaum zu erstellen,

[1] Instanziierung von ausführbaren Klassen-Objekten.
[2] Wurzelelement
[3] Blattelement

erfolgt anschließend eine Übersetzung der Terminalnamen in die Klassenbezeichnungen der abstrakten Syntax.

Für die **Erstellung des ausführbaren abstrakten Syntaxbaums** erfolgt ein weiteres Übersetzen der abstrakten Syntaxklassenbezeichnungen zu einem ausführbaren abstrakten Syntaxbaums. Dies kann beispielsweise durch die Instanziierung von zugehörigen Klassenobjekten in einer konkreten Implementierung erfolgen, die eine Möglichkeit der Ausführbarkeit haben. Das bedeutet, dass eine aufzurufende Funktion zur Evaluation der Klasseninstanz existieren muss.

Bei der **Ausführung des abstrakten Syntaxbaums** werden die Quellmodell-Instanzen eingelesen und der abstrakte Syntaxbaum mit diesen als Eingabedaten ausgeführt. Als Ergebnis wird die Zielmodell-Instanz erstellt.

Der komplette Ablauf inkl. der benötigten Artefakte ist in Abbildung 11.1 dargestellt.

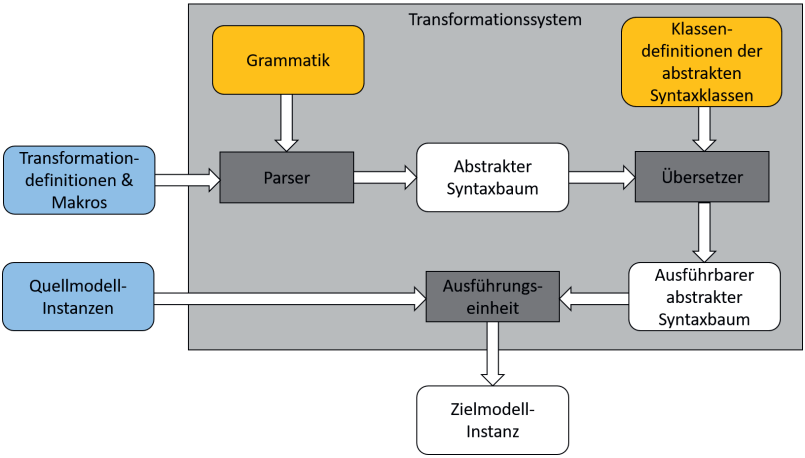

Abbildung 11.1: Aufbau und Ablauf eines Transformationssystem

Die Trennung dieser drei Funktionalitäten bietet den Vorteil, dass die Daten nicht vor jeder Durchführung einer Modelltransformation analysiert werden. Dadurch können bereits im Vorfeld die abstrakten Syntaxbäume erstellt und (zwischen-)gespeichert werden, sodass ein Transformationssystem diese direkt ausführen kann. Einige der Parser-Implementierungen bieten bereits die Option, die Erstellung eines ausführbaren abstrakten Syntaxbaum direkt durchzuführen, sodass ein weiteres Übersetzen nicht notwendig ist. Dies ermöglicht Performance-Vorteile, da direkt ein Baum aus Instanzen der abstrakten Syntaxklassen erstellt wird, anstelle eines Baums aus Token, der in einem zweiten Schritt vollständig eingelesen und transformiert werden muss.

11.2 Umsetzung in Python

Im Rahmen dieser Arbeit wurde eine Umsetzung für ein Transformationssystem in der Programmiersprache Python entwickelt. Die Realisierung wurde generisch gehalten, sodass diese für verschiedene Metamodelle genutzt werden kann. Dafür wurde eine zusätzliche Schnittstelle bei der Ausführungseinheit geschaffen, über die die Metamodell-spezifischen Klassen erzeugt werden.

Als Parser wurde **lark**[4] ausgewählt, da dieser die Möglichkeit der Definition einer kontextfreien Grammatik in Erweiterter Backus-Naur-Form erlaubt, direkt die Übersetzung zu den Klasseninstanzen der abstrakten Syntaxklassen ermöglicht sowie verschiedene Optionen der Performance-Optimierung beinhaltet. Um den Code einfach und einzeln wartbar zu halten, wurden für die einzelnen Teile des Transformationssystems jeweils Python-Skripte erstellt.

Grammatik: Für die Grammatikdefinition wurden zwei Dateien `grammar_ocl.lark` und `grammar_mtl.lark` erstellt (Anhang B). Da aktuell keine Python-Umsetzung von OCL existiert, wurden die Definitionen der Produktionsregeln von OCL Basic [53] in `grammar_ocl.lark` festgelegt. In `grammar_mtl.lark` wurden die benötigten Regeln inkludiert, nach Vorgabe erweitert sowie die neuen Produktionsregeln der Transformationssprache (Kapitel 9.2) hinzugefügt. Die Produktionsregeln in beiden Dateien wurden mit den Sprachelementen von lark umgesetzt. Die Sprachelemente von lark ermöglichen die Definition von Terminalen sowie Nicht-Terminalen (Produktionsregeln). Zusätzlich können einzelne für die weitere Auswertung nicht benötigte Terminale sowie optionale Terminale herausgefiltert werden. Des Weiteren wird die Definition der Multiplizität von Terminalen ermöglicht.

Klassendefinition der abstrakten Syntaxklassen: Für die Python-Klassendefinitionen der abstrakten Syntaxklassen wurden zwei weitere Python-Skripte `ast_ocl.py` und `ast_mtl.py` erstellt (Anhang C). Analog zur Grammatikdefinition wurden in `ast_ocl.py` Python-Klassen für die OCL-Klassen definiert. Jede Klassendefinition erbt dabei von der abstrakten Klasse *Token*, die den Konstruktor spezifiziert. Dieser erhält die Kind-Token und speichert diese in einer Variablen ab. Um einen ausführbaren abstrakten Syntaxbaum zu erhalten, definiert jede Klasse eine Funktion, die bei Aufruf die gespeicherten Token auswertet. Beispielsweise werden beim Aufruf dieser Auswertefunktion der Klasse *SimpleName* die enthaltenen Kind-Token, die jeweils einzelne Zeichen sind, zu einem String zusammengefügt und an die aufrufende Funktion zurückgeben. Die enthaltenen Kind-Token können wiederum Instanzen von Klassen mit Auswertefunktion sein. Zum Beispiel wird bei der *IfExp*-Klasse zunächst die Funktion des ersten Kind-Token, welche einen booleschen Wert zurückgibt, aufgerufen. Basierend auf dem Ergebnis wird entweder die Auswertefunktion des zweiten oder des dritten Kind-Token ausgeführt. Dadurch kann ein kompletter Baum beginnend beim Wurzel-Token ausgeführt werden. Um Variablen in verschiedenen Funktionen zu nutzen und die Zugriffe einzuschränken, wurden zwei Klassen für eine lokale und eine globale Umgebung eingeführt. Beide Umgebungen dienen dem Speichern von Variablen und Typen sowie dem Zugriff auf deren Werte. Während die Variablen und Typen der globalen Umgebung einmal festgelegt

[4] `https://github.com/lark-parser/lark`

werden, können bei der lokalen Umgebung bei jeder Auswertung eines Klassenobjekts weitere Variablen und Typen hinzugefügt werden.

Für die Klassendefinition der Transformationssprachelemente wurden in `ast_mtl.py` die benötigten Python-OCL-Klassen aus `ast_ocl.py` inkludiert und die benötigten Python-Klassen für die Transformationssprachelemente erstellt. Zusätzlich wurde eine neue globale Umgebung abgeleitet. Diese ermöglicht die Speicherung und den Zugriff auf Makros. Um in den Auswertefunktionen Instanzen der Metamodell-spezifischen Klassen zu ermöglichen, kann in der globalen Umgebung die Übersetzung von Python-Klassen einer Metamodell-Umsetzung auf die eigentlichen Metamodell-Bezeichner gespeichert werden. Für die Anwendungsbeispiele aus Kapitel 10 wurden z. B. die Klassendefinitionen des Verwaltungsschalen-Python-SDK PyI40AAS[5] gespeichert, um Submodel-Instanzen, die konform zur Metamodell-Definition [4] sind, in Python zu erstellen. Bei jedem Aufruf einer Auswertefunktion muss je eine Instanz der beiden Umgebungen übergeben werden. Dies ermöglicht, dass in jeder Auswertefunktion sowohl auf die global definierten Variablen, Typen und Makros zugegriffen als auch neue innerhalb dieser Funktion benötigten Variablen erstellt und auf deren Werte zugegriffen werden kann.

Für die Ausführung eines abstrakten Syntaxbaums muss mindestens eine Klasse als Wurzel-Klasse festgelegt werden. In dieser Sprachdefinition ist das die Klasse *PackageDeclaration*, die als Kind-Token maximal eine Transformationsdefinition enthält sowie eine beliebige Anzahl an Makrodefinitionen. Die Klasse bietet die Möglichkeit, die Transformationsdefinition sowie die Liste der Makrodefinitionen zu erhalten. Zusätzlich stellt diese Klasse eine Funktion zur Verfügung, die eine Liste von benötigten Paketen zurückgibt, aus denen Makrodefinitionen verwendet werden.

Für die Ausführung einer Modelltransformation wird die Auswertefunktion der Klasse *TransformationDefinition* genutzt. Beim Aufruf dieser Funktion müssen die Quellmodell-Instanzen, ggf. der Namen bzw. der Namenspfad auf das Attribut zur eindeutigen Definition der zugehörigen Quellmodell-Template-Definitionen, die benötigen Klassendefinitionen und vorgegebenen Variablen des Metamodells sowie die benötigten Makro-Token übergeben werden. Die Funktion erstellt zunächst die Instanzen für die lokale und globale Umgebung und fügt der globalen Umgebung die übergebenen Klassendefinitionen und Variablen des Metamodells sowie die Makros hinzu. Im nächsten Schritt werden die übergebenen Quellmodell-Instanzen hinsichtlich der geforderten Quellmodell-Templates überprüft. Entspricht eine Instanz dem Template, wird der Wert der entsprechenden Variable in der Transformationsdefinition durch dieses Objekt ersetzt. Am Ende dieses Schritts wird überprüft, ob Instanzen für alle geforderten Quellmodell-Templates vorliegen. Nachdem die Umgebungen vollständig initialisiert wurden, wird die eigentliche Transformation ausgeführt, indem die Auswertefunktion der enthaltenen *ObjectLiteralExp* aufgerufen wird.

[5] `https://git.rwth-aachen.de/acplt/pyi40aas`

Parser, Übersetzer und Ausführungseinheit: Die Funktionalitäten des Parsers, des Übersetzers und der Ausführungseinheit wurden in einem Python-Skript `parser.py` umgesetzt. Dabei wurden drei einzelne Funktionen erstellt:

1. Erstellung des Parsers
2. Erstellung des ausführbaren abstrakten Syntaxbaums
3. Erstellung und Ausführung des abstrakten Syntaxbaums

Die erste Funktion `get_parser` erzeugt eine Instanz des Lark-Parsers basierend auf den Grammatikdefinitionen in `grammar_mtl.lark` und `grammar_ocl.lark`. Dem Parser wird eine Transformer-Klasse übergeben, die das Mapping zwischen den Terminal- und Nichtterminalnamen und den Python-Klassendefinitionen beschreibt. Als Start-Token wird die Package-Deklaration vordefiniert sowie als Parsing-Methode *LALR (Look-Ahead-Left-To-Right)* genutzt. LALR wurde 1969 von Frank DeRemer erfunden und ermöglicht das Parsen eines Textes gemäß einer Menge von Produktionsregeln, die durch eine formale Grammatik spezifiziert sind [159]. Die erstellte Instanz kann im Anschluss von den anderen beiden Funktionen genutzt werden.

Das eigentliche Parsen und Übersetzen wird in der zweiten Funktion `parse_transformation_definition_file` durchgeführt. Diese erhält als Parameter eine Hauptdatei, weitere Dateien sowie optional einen Parser. In der Hauptdatei muss genau eine Transformationsdefinition enthalten sein. In den weiteren Dateien werden Makros spezifiziert. Falls kein Parser übergeben wurde, wird in der Funktion als erstes die Funktion `get_parser` aufgerufen. Anschließend wird das Hauptdokument mit diesem Parser eingelesen und aus dem Package die enthaltene Transformationsdefinition, die enthaltenen Makros sowie die benötigten Packages herausgefiltert. Im nächsten Schritt werden (sofern weitere Packages benötigt werden) die entsprechenden Dateien ebenfalls eingelesen und die Makros extrahiert. Die Rückgabe besteht aus dem ausführbaren abstrakten Syntaxbaum der Transformationsdefinition und den benötigten Makros. Mit diesen sowie den weiteren benötigten Informationen (siehe Auswertefunktion der Klasse *TransformationDefinition*) kann die Ausführung der Transformation erfolgen.

Für die Benutzerfreundlichkeit wird eine weitere Funktion `execute_transformation_definition` eingeführt, die zunächst die Funktion `parse_transformation_definition_file` aufruft und danach die Auswertefunktion der entsprechenden Transformationsdefinition ausführt. Als Ergebnis wird das Zielmodell erzeugt.

Es existieren verschiedene Anwendungsfälle. Je nach Anwendungsfall kann es sinnvoll sein, direkt die Funktion `execute_transformation_definition` aufzurufen oder die vorhergenannten Funktionen jeweils einzeln zu nutzen. Dies hat unterschiedliche Auswirkungen auf die Performance. Die genauen Details dazu sind in Abschnitt 12.5 beschrieben.

12 Evaluation

Die in Kapitel 9 vorgestellte und in Kapitel 10 auf Verwaltungsschalen angepasste Sprache kann für unterschiedliche Anwendungsfälle genutzt werden. In den nachfolgenden Abschnitten werden drei mögliche Anwendungsfälle beschrieben. Der erste Anwendungsfall behandelt die Nutzung von abgeleiteten firmenspezifischen Teilmodell-Templates. Im zweiten Anwendungsfall wird die Verwendung von unterschiedlichen Versionen eines Teilmodell-Templates beschrieben. Zuletzt wird gezeigt, wie mit Hilfe der vorgestellten Sprache die Integration von Informationen aus unterlagerten Komponenten in einer zusammengesetzten Anlage oder Maschine erfolgen kann. Danach folgt eine Diskussion über die benötigte Zeit für die Erstellung einer Transformationsdefinition. Den Abschluss dieses Kapitels bildet eine Evaluation der Umsetzung des Transformationssystems aus Kapitel 11. Außerdem werden verschiedene Optimierungsempfehlungen hinsichtlich der Funktionsaufrufe des entwickelten Transformationssystems zur Verbesserung der Performance in der Anwendung vorgestellt.

12.1 Anwendungsfall 1: Firmenspezifische Informationsmodelle

Häufig werden für die firmeninterne Nutzung von Verwaltungsschalen weitere spezifische Informationen benötigt, die in den standardisierten Teilmodell-Templates nicht abgebildet sind. Um dennoch die Informationen Firmen-intern einheitlich abzulegen und zu nutzen, werden Firmen die standardisierten Teilmodell-Templates erweitern und eigene firmenspezifische Teilmodell-Templates erstellen. Dadurch erhalten diese eine eigene SemanticId. Da externe Kommunikationspartner jedoch die standardisierten Teilmodell-Templates anfragen, müssen die Informationen aus den vorgehaltenen Teilmodell-Instanzen in die Form der standardisierten Teilmodell-Templates transformiert werden. Hierfür kann die vorgestellte Sprache genutzt werden. Vor allem bei einer Reduzierung der Informationen kann die Transformationsdefinitionen einfach formuliert oder automatisiert generiert werden.

Anhand eines abgeleiteten firmenspezifischen Teilmodell-Templates der Firma WITTENSTEIN galaxie GmbH wird dieser Anwendungsfall beschrieben. Im Anhang D.1 ist das UML des vom ZVEI standardisierten Teilmodell-Templates „ZVEI Digital Nameplate for industrial equipment (Version 1.0)“ [14] (im Folgenden ZVEI-Template genannt) dargestellt. Basierend darauf wurde von der Firma WITTENSTEIN galaxie GmbH eine Erweiterung des Templates für „Galaxie“-Aktuatoren[1] [160] erstellt, welches dadurch eine neue SemanticId erhalten hat. Ein Ausschnitt des zugehörigen UML ist in

[1]Galaxie-Aktuatoren sind von der WITTENSTEIN galaxie entwickelte und produzierte radikal innovative Getriebe und Antriebssysteme `https://galaxie.wittenstein.de/de-de/produkte/galaxie/`

Anhang D.2 gegeben (im Folgenden WITTENSTEIN-Template genannt). Dabei wurden alle verpflichtenden Elemente übernommen, einige der optionalen Elemente entfernt sowie neue Elemente hinzugefügt. Die neuen Elemente sind in der SubmodelElementCollection *AssetSpecificProperties* dargestellt. Sobald das standardisierte ZVEI-Template angefragt wird, muss eine entsprechende Instanz dieses Templates erzeugt und mit den Werten des WITTENSTEIN-Templates befüllt werden. Die zugehörige Transformationsdefinition ist in Anhang D.3 abgedruckt.

Die Transformationsdefinition beginnt mit der Referenz auf das WITTENSTEIN-Template als Quell-Informationsmodell-Template, gefolgt von der Referenz auf das ZVEI-Template als Ziel-Informationsmodell-Template. Im Anschluss folgt die Definition der neu zu erstellenden Teilmodell-Instanz. Viele der Attribute werden eins zu eins aus dem WITTENSTEIN-Template übernommen. Lediglich die SemanticId wird auf die des ZVEI-Template angepasst sowie die enthaltenen SubmodelElements explizit kopiert. Da eine Reduktion der Informationen vorliegt, werden zunächst alle gleichen Elemente mit dem Makro *copySubmodelElementByIdShort* (s. Kapitel 10) übernommen. Dies beinhaltet auch die Kindelemente der Elemente vom Typ *SubmodelElementCollection*, z. B. bei *Address* oder *Markings*. Die firmenspezifisch hinzugefügten Elemente *Weight*, *FeedbackSystem*, *MountingPosition*, *Lubrication*, *CommutationOffset* und *TempSensorType* werden jedoch nicht kopiert. Als Ergebnis wird eine zum ZVEI-Template konforme Teilmodell-Instanz erstellt, die dem Anfragenden zurückgegeben wird.

12.2 Anwendungsfall 2: Verschiedene Versionen standardisierter Informationsmodelle

In Kapitel 1.1 wurde bereits das Problem von verschiedenen Informationsmodell-Versionen beschrieben. Im Zuge der Nutzung von Teilmodell-Templates wird immer wieder eine Anpassung der Informationen notwendig, da dies für eine bessere Verarbeitung oder aufgrund einer anderen Darstellung gefordert wird. Infolgedessen werden neue Versionen dieser standardisierten Templates entstehen. Die Informationen können identisch, jedoch strukturell anders modelliert sein. Am Beispiel des unveröffentlichten Teilmodell-Templates „Minimum requirements for the Handover documentation from the manufacturer to the operator based on the VDI 2770 specification“ wird dies nachfolgend beschrieben. Dazu werden drei verschiedene nicht veröffentlichte Versionen betrachtet, die bei der Entwicklung des finalen Teilmodell-Templates entstanden sind. Diese dienen stellvertretend für spätere Versionen veröffentlichter Teilmodell-Templates. Die UML-Diagramme der einzelnen Versionen sind in den Anhängen E.1, E.2 und E.3 abgebildet.

Änderungen von Version 1 zu Version 2

In der ersten Version wurden die Informationen durch zwei SubmodelElementCollections *Document* und *DocumentVersion* modelliert. Bei der zweiten Version wurden die Informationen zur Dokument-Klassifikation in eine weitere SubmodelElementCollection *DocumentClassification* ausgelagert. Da lediglich eine strukturelle Änderung erfolgte, bleiben die Informationen gleich. Die enthaltenen Elemente wurden strukturell und von ihren Attributwerten wiederverwendet. Zudem wurde ein neues optionales File-Objekt mit der

IdShort *PreviewFile* hinzugefügt. Eine Darstellung der Änderungen ist in Abbildung 12.1 gegeben.

Version 1		idShort	semanticId	Type	Kardinalität
Submodel		ManufacturerDocumentation	[IRI]http://admin-shell.io/vdi/2770/1/0/Documentation		
SMC		Document{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/Document		0..*
Property		DocumentId	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentId/Id	string	1
Property		IsPrimaryDocument	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentId/isPrimary	boolean	1
Property		DocumentClassId	[IRI]http://adminshell.io/vdi/2770/1/0/DocumentClassification/ClassId	string	1
MLP		DocumentClassName	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentClassification/ClassName		1
Property		DocumentClassificationSystem	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentClassification/ClassificationSystem	string	1
Ref		ReferencedObject{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/Document/ReferencedObject		0..*
SMC		DocumentVersion{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentVersion		0..*
	Property	Language{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentVersion/Language	string	1..*
	Property	DocumentVersionId	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentVersion/DocumentVersionId	string	1
	MLP	Title	[IRI]http://admin-shell.io/vdi/2770/1/0/Description/Title		1
	MLP	Summary	[IRI]http://adminshell.io/vdi/2770/1/0/DocumentDescription/Summary		1
	MLP	KeyWords	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentDescription/KeyWords		0..1
	Property	SetDate	[IRI]http://admin-shell.io/vdi/2770/1/0/LifeCycleStatus/SetDate	date	1
	Property	StatusValue	[IRI]http://admin-shell.io/vdi/2770/1/0/LifeCycleStatus/StatusValue	string	1
	Property	Role	[IRI]http://admin-shell.io/vdi/2770/1/0/Party/Role	string	1
	Property	OrganizationName	[IRI]http://adminshell.io/vdi/2770/1/0/Organization/OrganizationName	string	1
	Property	OrganizationOfficialName	[IRI]http://admin-shell.io/vdi/2770/1/0/Organization/OrganizationOfficialName	string	1
	File	DigitalFile{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/StoredDocumentRepresentation/DigitalFile	application/pdf	1..*

Version 2		idShort	semanticId	Type	Kardinalität
Submodel		ManufacturerDocumentation	[IRI]http://admin-shell.io/vdi/2770/1/1/Documentation		
SMC		Document{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/Document		0..*
Property		DocumentId	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentId/Id	string	1
Property		IsPrimaryDocument	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentId/isPrimary	boolean	1
SMC		DocumentClassification{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/ DocumentClassification/ DocumentClassification		1..*
	Property	DocumentClassId	[IRI]http://adminshell.io/vdi/2770/1/0/DocumentClassification/ClassId	string	1
	MLP	DocumentClassName	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentClassification/ClassName		1
	Property	DocumentClassificationSystem	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentClassification/ClassificationSystem	string	1
Ref		ReferencedObject{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/Document/ReferencedObject		0..*
SMC		DocumentVersion{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentVersion		0..*
	Property	Language{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentVersion/Language	string	1..*
	Property	DocumentVersionId	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentVersion/DocumentVersionId	string	1
	MLP	Title	[IRI]http://admin-shell.io/vdi/2770/1/0/Description/Title		1
	MLP	Summary	[IRI]http://adminshell.io/vdi/2770/1/0/DocumentDescription/Summary		1
	MLP	KeyWords	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentDescription/KeyWords		0..1
	Property	SetDate	[IRI]http://admin-shell.io/vdi/2770/1/0/LifeCycleStatus/SetDate	date	1
	Property	StatusValue	[IRI]http://admin-shell.io/vdi/2770/1/0/LifeCycleStatus/StatusValue	string	1
	Property	Role	[IRI]http://admin-shell.io/vdi/2770/1/0/Party/Role	string	1
	Property	OrganizationName	[IRI]http://adminshell.io/vdi/2770/1/0/Organization/OrganizationName	string	1
	Property	OrganizationOfficialName	[IRI]http://admin-shell.io/vdi/2770/1/0/Organization/OrganizationOfficialName	string	1
	File	DigitalFile{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/StoredDocumentRepresentation/DigitalFile	application/pdf	1..*
	File	PreviewFile	[IRI] https://admin-shell.io/vdi/2770/1/0/ StoredDocumentRepresentation/PreviewFile	image/jpg	0..1

Abbildung 12.1: Detaillierte farbliche Auflistung der semantischen Gleichheiten zwischen Version 1 und Version 2

Änderung von Version 2 zu Version 3

In Version 3 wurden, neben der Auslagerung von Informationen in eine neue Submodel-ElementCollection, auch die IdShorts und SemanticIds geändert. Des Weiteren wurde eine Aufsplittung von Werten sowie eine Anpassung der Kardinalitäten vorgenommen. Der erste Unterschied ist die neu eingeführte SubmodelElementCollection *DocumentId*, die die Informationen zur Identifikation des Dokuments enthält. Während in Version 2 ausschließlich eine Id enthalten sein konnte, sind in der Version mehrere Ids möglich. Zusätzlich wurde die DocumentId genauer spezifiziert, sodass die *DocumentDomainId* explizit angegeben werden muss, die in Version 2 lediglich implizit enthalten ist. Außerdem wurde die IdShort *IsPrimaryDocument* auf *IsPrimary* gekürzt. Die SubmodelElement-Collection *DocumentClassification* wurde im Inhaltlichen beibehalten, jedoch wurden die IdShorts sowie die SemanticId der Collection selbst gekürzt. Semantisch sind die Informationen weiterhin identisch. Die Referenz mit der IdShort *ReferencedObject* wurde ebenfalls semantisch beibehalten, jedoch wurde die IdShort und die SemanticId verändert. Abschließend wurden die Elemente der SubmodelElementCollection *DocumentVersion* fast vollständig beibehalten. Das Property-Element *Role* wurde entfernt sowie neue optionale Elemente hinzugefügt (*SubTitle*, *RefersTo*, *BasedOn*, *TranslationOf*). Die verschiedenen

Änderungen sind in Abbildung 12.2 farblich dargestellt.

Version 2		idShort	semanticId	Type	Kardinalität
Submodel		ManufacturerDocumentation	[IRI]http://admin-shell.io/vdi/2770/1/0/Documentation		
SMC		Document{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/Document		0..*
	Property	DocumentId	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentId/Id	string	1
	Property	IsPrimaryDocument	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentId/isPrimary	boolean	1
	SMC	DocumentClassification{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/ DocumentClassification/ DocumentClassification		1..*
		Property DocumentClassId	[IRI]http://adminshell.io/vdi/2770/1/0/DocumentClassification/ClassId	string	1
		MLP DocumentClassName	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentClassification/ClassName		1
		Property DocumentClassificationSystem	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentClassification/ClassificationSystem	string	1
	Ref	ReferencedObject	[IRI]http://admin-shell.io/vdi/2770/1/0/Document/ReferencedObject		0..1
	SMC	DocumentVersion{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentVersion		0..*
		Property Language{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentVersion/Language	string	1..*
		Property DocumentVersionId	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentVersion/DocumentVersionId	string	1
		MLP Title	[IRI]http://admin-shell.io/vdi/2770/1/0/Description/Title		1
		MLP Summary	[IRI]http://adminshell.io/vdi/2770/1/0/DocumentDescription/Summary		1
		MLP KeyWords	[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentDescription/KeyWords		0..1
		Property SetDate	[IRI]http://admin-shell.io/vdi/2770/1/0/LifeCycleStatus/SetDate	date	1
		Property StatusValue	[IRI]http://admin-shell.io/vdi/2770/1/0/LifeCycleStatus/StatusValue	string	1
		Property Role	[IRI]http://admin-shell.io/vdi/2770/1/0/Party/Role	string	1
		Property OrganizationName	[IRI]http://adminshell.io/vdi/2770/1/0/Organization/OrganizationName	string	1
		Property OrganizationOfficialName	[IRI]http://admin-shell.io/vdi/2770/1/0/Organization/OrganizationOfficialName	string	1
		File DigitalFile{00}	[IRI]http://admin-shell.io/vdi/2770/1/0/StoredDocumentRepresentation/DigitalFile	application/pdf	1..*
		File PreviewFile{00}	[IRI] https://admin-shell.io/vdi/2770/1/0/ StoredDocumentRepresentation/PreviewFile	image/jpg	0..1

Version 3		idShort	semanticId	Type	Kardinalität
Submodel		ManufacturerDocumentation{00}	[IRI] https://admin-shell.io/vdi/2770/1/1/Documentation		
Entity		Entity{00}	[IRI] http://adminshell.io/vdi/2770/1/0/EntityForDocumentation		0..*
SMC		Document{00}	[IRI] http://admin-shell.io/vdi/2770/1/0/Document		0..*
	Ref	DocumentedEntity{00}	[IRI] https://admin-shell.io/vdi/2770/1/0/Document/DocumentedEntity		0..*
	SMC	DocumentId{00}	[IRI] https://admin-shell.io/vdi/2770/1/0/DocumentId		1..*
		Property DocumentDomainId	[IRI]https://admin-shell.io/vdi/2770/1/0/DocumentId/DocumentDomainId	string	1
		Property ValueId	[IRI] https://admin-shell.io/vdi/2770/1/0/DocumentId/ValueId	string	1
		Property IsPrimary	[IRI]https://admin-shell.io/vdi/2770/1/0/DocumentId/IsPrimary	boolean	0..1
	SMC	DocumentClassification{00}	[IRI] https://admin-shell.io/vdi/2770/1/0/DocumentClassification		1..*
		Property ClassId	[IRI] https://adminshell.io/vdi/2770/1/0/DocumentClassification/ClassId	string	1
		MLP ClassName	[IRI] http://admin-shell.io/vdi/2770/1/0/DocumentClassification/ClassName		1
		Property ClassificationSystem	[IRI] http://admin-shell.io/vdi/2770/1/0/DocumentClassification/ClassificationSystem	string	1
	SMC	DocumentVersion{00}	[IRI] https://admin-shell.io/vdi/2770/1/0/DocumentVersion		0..*
		Property Language{00}	[IRI]https://admin-shell.io/vdi/2770/1/0/DocumentDescription/Language	string	1..*
		Property DocumentVersionId	[IRI] https://admin-shell.io/vdi/2770/1/0/DocumentVersion/DocumentVersionId	string	1
		MLP Title	[IRI] https://admin-shell.io/vdi/2770/1/0/DocumentDescription/Title		1
		MLP SubTitle	[IRI] https://admin-shell.io/vdi/2770/1/0/DocumentDescription/SubTitle		0..1
		MLP Summary	[IRI] https://adminshell.io/vdi/2770/1/0/DocumentDescription/Summary		1
		MLP KeyWords	[IRI] https://admin-shell.io/vdi/2770/1/0/ DocumentDescription/KeyWords		1
		Property SetDate	[IRI]https://admin-shell.io/vdi/2770/1/0/LifeCycleStatus/SetDate	date	1
		Property StatusValue	[IRI]https://admin-shell.io/vdi/2770/1/0/LifeCycleStatus/StatusValue	string	1
		Property OrganizationName	[IRI] https://adminshell.io/vdi/2770/1/0/Organization/OrganizationName	string	1
		Property OrganizationOfficialName	[IRI] https://admin-shell.io/vdi/2770/1/0/ Organization/OrganizationOfficialName	string	1
		File DigitalFile{00}	[IRI] https://admin-shell.io/vdi/2770/1/0/ StoredDocumentRepresentation/DigitalFile	application/pdf	1..*
		File PreviewFile{00}	[IRI] https://admin-shell.io/vdi/2770/1/0/ StoredDocumentRepresentation/PreviewFile	image/jpg	0..1
		Ref RefersTo{00}	[IRI] https://admin-shell.io/vdi/2770/1/0/ DocumentVersion/RefersTo		0..*
		Ref BasedOn{00}	[IRI] https://admin-shell.io/vdi/2770/1/0/ DocumentVersion/BasedOn		0..*
		Ref TranslationOf{00}	[IRI] https://admin-shell.io/vdi/2770/1/0/ DocumentVersion/TranslationOf		0..*

Abbildung 12.2: Detaillierte farbliche Auflistung der semantischen Gleichheiten zwischen Version 2 und Version 3

Beispielhafte Transformationsdefinition

Eine beispielhafte Transformationsdefinition für die Transformation eines Teilmodells der Version 1 in ein Teilmodell der Version 2 ist in Anhang E.4 dargestellt. Zunächst werden die entsprechenden Referenzen auf die beiden Versionen des Teilmodell-Templates als Quell- und Ziel-Informationsmodell-Template angegeben. Die Definition der neuen Teilmodell-Instanz übernimmt alle Attribute bis auf die SemanticId und die enthaltenen SubmodelElemente. Die SemanticId wird auf die zu erstellende Version angepasst. Für die Erstellung der enthaltenen und zu iterierenden Elemente werden alle SubmodelElementCollections mit der SemanticId *[IRI]http://admin-shell.io/vdi/2770/1/0/Document* ermittelt. In jeder Iteration wird eine neue SubmodelElementCollection mit denselben Attributwerten, jedoch mit anderen Kindelementen, erstellt. Bei den Kindelementen werden zunächst die Elemente mit der IdShort *DocumentId* und *IsPrimary* sowie die Elemente mit der SemanticId *[IRI]http://admin-shell.io/vdi/2770/1/0/Document/ReferencedObject* kopiert. Anschließend folgt die Erstellung der neu eingeführten SubmodelElementCollec-

tion *DocumentClassification* mit den definierten Attributen aus der Spezifikation sowie den SubmodelElementen mit den IdShorts *DocumentClassId*, *DocumentClassName* und *DocumentClassificationSystem*. Diese stammen aus der vorherigen Version der SubmodelElementCollection *Document*. Als letztes werden alle SubmodelElementCollection mit der SemanticId *[IRI]http://admin-shell.io/vdi/2770/1/0/DocumentVersion* hinzugefügt.

Fazit

Es wurde gezeigt, dass alle Informationen der Version 2 aus den Informationen der Version 1 und andersherum erzeugt werden können. Dasselbe trifft auf die Versionen 2 und 3 zu, wobei die *DocumentId* bzw. *DocumentDomainId* und *ValueId* je nach Richtung nicht direkt aus den vorliegenden Informationen erstellt werden kann. Durch String-Konkatenation oder String-Splitten ist dies dennoch mit der Sprache möglich. Dazu wird in der Transformationsdefinition eine entsprechende manuelle Entscheidung über den neuen Wert vorgegeben. Somit wird mit der vorgestellten Sprache die Erstellung von Transformationsdefinitionen ermöglicht, die (teils-)automatisch Teilmodell-Instanzen einer anderen Version instanziieren können. Ziel soll sein, bei der Definition einer neuen Version eine entsprechende Transformationsdefinition zu erstellen und zur Verfügung zu stellen, sodass weiterhin eine semantische Interoperabilität zwischen Komponenten, die mit verschiedenen Versionen arbeiten, gewährleistet ist.

12.3 Anwendungsfall 3: Integration von Komponenten und zugehörigen Informationsmodellen

Bei der Integration von Komponenten in (Teil-)Anlagen können vielfach Informationen aus den einzelnen Komponenten für die Beschreibung der Gesamtanlage wiederverwendet werden. Dazu müssen die Informationen in dem Informationsmodell der Anlage zusammengeführt werden. Dieser Vorgang kann mit Hilfe einer Modelltransformation und der in dieser Arbeit vorgestellten Sprache erfolgen. Anhand von Leistungskennzahlen wird dies nachfolgend exemplarisch gezeigt.

In Abbildung 12.3 sind beispielhaft zwei Teilmodell-Templates für diesen Anwendungsfall definiert. Das erste Template repräsentiert die Leistungsüberwachung einer verbauten Komponente einer Anlage und enthält ein Property-Element für die maximale Leistungsaufnahme, ein Property-Element für die Stromart (Gleichstrom/Wechselstrom), mit der die jeweilige Komponente betrieben werden darf, sowie ein Property-Element für die Netz-Nennspannung. Das zweite Teilmodell stellt die Daten der Gesamtanlage dar. Ein Property-Element enthält dabei die maximale Gesamtleistungsaufnahme der Anlage und ein weiteres Property-Element die maximal benötigte Netzspannung, sofern alle Komponenten mit maximaler Netzspannung betrieben werden. Das dritte Property-Element beschreibt alle benötigten Kombinationen aus Netz-Nennspannung und zugehöriger Stromart.

Dabei wird angenommen, dass jede verbaute elektrische Komponente der Anlage eine Instanz des ersten Teilmodell-Templates besitzt. Für einen Servomotor könnte z. B. 36 Watt sowie 24V DC oder 1100 Watt sowie 400V AC in der Teilmodell-Instanz eingetragen sein. In

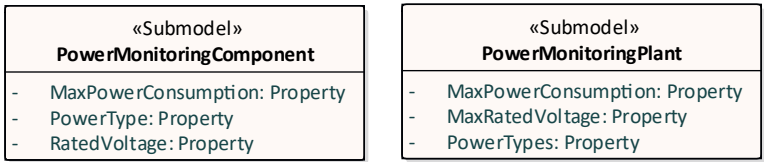

Abbildung 12.3: Beispiel für ein Teilmodell-Template zur Erfassung von Leistungskennzahlen

der Teilmodell-Instanz der Gesamtanlage können die Informationen der einzelnen Komponenten zusammengeführt werden. Dafür werden für die maximale Leistungsaufnahme der Anlage die Werte der einzelnen Komponenten addiert. Die Netzspannung der Anlage kann über die maximale Netzspannung alle Komponenten ermittelt werden (MAX-Funktion). Zur Bestimmung der benötigten Netz-Nennspannung- und Stromart-Kombinationen werden alle verschiedenen Kombinationen der einzelnen Komponenten als Sammlung zusammengetragen. Dadurch können diese an einer Stelle ausgelesen werden und die benötigten Arten von Netzteilen bzw. AC/DC-Wandlern ermittelt werden.

Eine zugehörige Transformationsdefinition ist in Anhang F gegeben. Als Eingangsinstanzen werden alle Teilmodelle genutzt, die eine Referenz auf das Teilmodell-Template *PowerMonitoringComponent* besitzen. Das Ziel-Teilmodell ist eine Instanz des Teilmodell-Templates *PowerMonitoringPlant*. Es werden die drei im Teilmodell-Template definierten Property-Elemente angelegt und mit den entsprechenden Werten belegt. Für das Property-Element *MaxPowerConsumption* wird über alle übergebenen Teilmodelle iteriert und jeweils der Wert aus dem Property-Element *MaxPowerConsumption* addiert. Das Ergebnis entspricht der maximalen Gesamtleistungsaufnahme der Anlage. Im zweiten Property-Element *RatedVoltage* wird ebenfalls über die einzelnen Teilmodelle iteriert, jedoch wird lediglich der Maximalwert der Property-Elemente *RatedVoltage* ermittelt. Für das Property-Element *PowerTypes* wird wiederum über die Teilmodelle iteriert. Pro Iteration wird die Kombination aus Netz-Nennspannung und Stromart in Form eines Strings erstellt. Anschließend wird in der bestehenden Liste überprüft, ob diese Kombination bereits enthalten ist. Ist dies der Fall, bleibt die Liste unverändert und die nächste Iteration wird durchgeführt. Andernfalls wird die neue Kombination der Liste hinzugefügt.

In der Tabelle 12.2 ist das Ergebnis beispielhaft auf Basis der Komponentenwerte aus Tabelle 12.1 einer solchen Transformation dargestellt. Es wird von drei Komponenten ausgegangen, die in einer Gesamtanlage verbaut sind.

Tabelle 12.1: Beispielwerte der Komponenten

Property	Komponente 1	Komponente 2	Komponente 3
MaxPowerConsumption	36	1100	40
RatedVoltage	24	400	24
PowerType	DC	AC	DC

Tabelle 12.2: Beispielwerte der Gesamtanlage

Property	Gesamtanlage
MaxPowerConsumption	1176
MaxRatedVoltage	400
PowerTypes	{24DC, 400AC}

12.4 Benötigte Zeit für die Erstellung einer Transformationsdefinition

Eine genaue Angabe der benötigten Zeit für die Erstellung einer Transformationsdefinition ist nicht möglich. Dies liegt daran, dass zu viele verschiedene Einflussfaktoren vorliegen. Nachfolgend sind einige Faktoren beschrieben.

Ein entscheidender Faktor ist die Komplexität der Transformation. Je umfangreicher und komplexer die einzelnen Regeln sind, desto aufwändiger ist die Erstellung. Dies tritt insbesondere dann auf, wenn aus verschiedenen Quellmodellen Informationen zusammengeführt werden und diese zusätzlich noch in einer anderen Darstellung im Zielmodell vorliegen sollen. Zusätzlich muss die Erfahrung des Erstellers im Umgang mit der Sprache betrachtet werden. Hat der Ersteller bereits Erfahrung im Umgang mit OCL ist die Nutzung der Sprache einfacher, da nur wenige neue Sprachelemente gelernt werden müssen. Andernfalls muss zunächst die Sprache erlernt werden, welches zur langsameren Erstellung führt. Ein weiterer wichtiger Faktor ist die Tool-Unterstützung. Zum einen kann eine Tool-Unterstützung helfen Syntaxfehler zu vermeiden. Dadurch können Transformationsdefinitionen deutlich schneller entwickelt werden. Zum Anderen muss der Ersteller im ersten Schritt die semantische Gleichheit von verschiedenen Informationen ermitteln. Hierzu könnte ein Assistenzsystem Unterstützung bieten, indem Vorschläge über mögliche Gleichheiten gegeben werden. Erste Arbeiten in diesem Bereich existieren [16]. Alternativ benötigt der Ersteller Erfahrung im Bereich der semantischen Gleichheit. Je besser ein Domänenexperte die verschiedenen Informationsmodelle kennt und dadurch schneller die semantischen Zusammenhänge herstellen kann, desto schneller kann er auch die dazugehörigen Regeln definieren. Abschließend kann die Nutzung von Makros eine Reduzierung der Erstellungszeit erreichen. Hierbei ist sowohl die Anzahl und die Art der verfügbaren Makros zu betrachten als auch die Kenntnis über diese. Je mehr Makros für oft benötigte Regeldefinitionen vorliegen, desto schneller können Transformationsdefinitionen erstellt werden.

12.5 Optimierung der Funktionsaufrufe im Lebenszyklus einer Komponente bei Nutzung des entwickelten Transformationssystems

Für die Evaluation der Implementierung wurden verschiedene Testreihen durchgeführt. Jede Testreihe entsprach 100 Durchläufen einer vollständigen Transformation. Dazu wurde

jeweils ein Dokument mit der durchzuführenden Transformationsdefinition, weitere Dokumente mit Makro-Definitionen, sowie entsprechende Test-Instanzen vorgegeben. Es wurden insgesamt 16 Testreihen mit acht unterschiedlichen Transformationen auf zwei Systemen durchgeführt. In Tabelle 12.3 sind die Hardware-Spezifikationen der beiden verwendeten Systeme aufgelistet.

Tabelle 12.3: Hardware-Spezifikationen der Systeme

Nr.	CPU	Arbeitsspeicher	Betriebssystem
1	Intel Core i5-6440HQ	8GB 1067MHz DDR4-2133	Windows 10 Enterprise LTSC (Version 1809)
2	Intel Core i7-6700K	32GB 2133MHz DDR4-2133	Arch Linux (Version 5.12.2)

Die Testreihen unterscheiden sich im Umfang der Transformationsdefinition sowie in der Anzahl der verwendeten Makros. Es wurden verschiedene Zeitdifferenzen für die einzelnen Schritte der Transformation gemessen und wie folgt kategorisiert:

1. Erstellung des Parsers
2. Parsen der Datei mit der entsprechenden Transformationsdefinition
3. Ermittlung der benötigten zusätzlichen Dateien aufgrund der Verwendung von Makros aus diesen Dateien
4. Parsen der zusätzlich benötigten Makro-Dateien
5. Anwendung des ausführbaren abstrakten Syntaxbaums

Für die Analyse wurde der in Kapitel 11 vorgestellte Parser verwendet, der beim Parsen direkt ausführbare abstrakte Syntaxbäume erstellt.

Die gemessenen Daten sind in den Abbildungen G.1 bis G.10 dargestellt. In den Abbildungen ist die gleiche Skalierung der Achsen gewählt, um die Testreihen besser vergleichen zu können. Die unterschiedlichen Zeiten zwischen den Testreihen sind durch die unterschiedlich verwendeten Transformationsdefinitionen und deren benötigte Makros zu erklären. Beispielsweise waren in der Testreihe *Test2* die Makros bereits in dem Dokument mit der durchzuführenden Transformationsdefinition enthalten. Aus diesem Grund hat das Parsing der entsprechenden Datei länger gedauert als bei den anderen Testreihen. Dafür musste jedoch keine weitere Datei geparst werden, weswegen die Zeitdifferenz für den vierten Schritt gleich null ist.

Aufgrund der geringen Streuung der Werte kann repräsentativ pro Testreihe ein Mittelwert berechnet und für die Interpretation genutzt werden. Für einen Vergleich zwischen den Systemen sind die Mittelwerte der jeweiligen Zeitdifferenzen und Systeme in Tabelle 12.4 aufgelistet. Zusätzlich wurden die prozentualen Anteile der Zeitdifferenzen auf die Gesamtzeit der jeweiligen Testreihen berechnet. Erwartungsgemäß konnten die Mittelwerte der Zeitdifferenzen beim gleichen Testsetup durch eine bessere Hardware-Spezifikation gesenkt werden. Eine weitere Erkenntnis ist, dass eine bessere Hardware-Spezifikation keinen Einfluss auf die prozentualen Anteile der einzelnen Zeitdifferenzen hat. Das bedeutet, dass

durch Auslagerung der aufwändigsten Schritte - sofern möglich - ein Performance-Vorteil in den jeweiligen Applikationen erreicht werden kann.

Der größte Teil der Gesamtzeit wird für die Erstellung des Parsers benötigt (vgl. erste Zeitdifferenz in Tabelle 12.4). Das mehrmalige Erstellen des Parsers ist jedoch nicht notwendig, da ein einmalig erstellter Parser in beliebig vielen Transformationen wiederverwendet werden kann. Aufgrund dessen sollte dieser beim Starten einer Applikation einmal erstellt und im Arbeitsspeicher vorgehalten werden. Die nächsten beiden größeren Zeitanteile sind das Parsen der Transformationsdefinitions-Datei und das Parsen der Makro-Dateien. Dabei besteht die Option, die Standard-Makros (z. B. die Makrodefinitionen aus Kapitel 10) ebenfalls beim Starten der Applikation zu parsen und die erhaltenen ausführbaren abstrakten Syntaxbäume der Makros im Arbeitsspeicher abzulegen. Dadurch kann die vierte Zeitdifferenz, sofern keine weiteren Makros benötigt werden, eliminiert werden. Für die zweite Zeitdifferenz (Parsen der Transformationsdefinitions-Datei) existieren mehrere Optionen. Eine Option wäre, bereits beim Start häufig benötigte Transformationsdefinitions-Dateien zu parsen und die entsprechenden ausführbaren abstrakten Syntaxbäume im Arbeitsspeicher vorzuhalten. Alternativ könnte nachdem eine Transformationsdefinitions-Datei geparst wurde, der zugehörige ausführbare abstrakte Syntaxbaum für eine gewisse Zeit im Arbeitsspeicher vorgehalten werden.

Wie gezeigt existieren einige Optimierungsvarianten, die eine Reduzierung der benötigten Zeit für eine Transformation um bis zu 95% ermöglichen. Welche Variante für die jeweilige Applikation sinnvoll ist, kann pauschal nicht beantwortet werden, da die Anforderungen pro Applikation unterschiedlich sein können. Anforderungen könnten beispielsweise ein möglichst geringer Arbeitsspeicherverbrauch, immer die aktuelle Transformationsdefinitions-Datei und Makro-Dateien zu nutzen, oder ein Wechsel des Parsers je nach auszuführender Transformation sein.

Tabelle 12.4: Mittelwerte und prozentualer Anteil der Zeitdifferenzen an der Gesamtzeit

Testreihe	System	MW ZD1	% ZD1	MW ZD2	% ZD2	MW ZD3	% ZD3	MW ZD4	% ZD4	MW ZD5	% ZD5
1	1	0.5528	81.16	0.0599	8.80	0.0003	0.05	0.0669	9.82	0.0012	0.18
1	2	0.3626	81.37	0.0402	9.02	0.0002	0.05	0.0419	9.40	0.0007	0.16
2	1	0.5489	76.35	0.1670	23.14	0.0007	0.10	0.0000	0.00	0.0030	0.41
2	2	0.3619	77.44	0.1031	22.06	0.0004	0.09	0.0000	0.00	0.0019	0.41
3	1	0.5469	72.82	0.0466	6.20	0.0003	0.04	0.1536	20.44	0.0037	0.50
3	2	0.3619	73.81	0.0302	6.17	0.0002	0.05	0.0955	19.47	0.0025	0.50
4	1	0.5419	71.30	0.0627	8.25	0.0009	0.11	0.1494	19.65	0.0053	0.70
4	2	0.3628	72.51	0.0404	8.08	0.0005	0.10	0.0931	18.60	0.0036	0.71
5	1	0.5421	71.25	0.0603	7.92	0.0008	0.10	0.1487	19.54	0.0091	1.19
5	2	0.3617	72.56	0.0386	7.75	0.0005	0.09	0.0915	18.35	0.0062	1.24
6	1	0.5468	70.37	0.0791	10.18	0.0011	0.14	0.1439	18.51	0.0062	0.80
6	2	0.3633	72.31	0.0506	10.07	0.0006	0.12	0.0839	16.70	0.0040	0.79
7	1	0.5568	67.80	0.1177	14.28	0.0029	0.35	0.1364	16.59	0.0081	0.99
7	2	0.3634	68.79	0.0741	14.02	0.0017	0.32	0.0838	15.86	0.0054	1.02
8	1	0.5507	69.63	0.0947	11.97	0.0017	0.22	0.1349	17.05	0.0089	1.12
8	2	0.3622	70.94	0.0597	11.68	0.0010	0.19	0.0818	16.01	0.0060	1.17

Abkürzungen:

MW Mittelwert
% Prozentualer Anteil an der Gesamtzeit
ZD1 Zeitdifferenz 1: Erstellung Parser
ZD2 Zeitdifferenz 2: Parsen Transformationsdefinition-Datei
ZD3 Zeitdifferenz 3: Ermittlung benötigter Dateien
ZD4 Zeitdifferenz 4: Parsen Makro-Dateien
ZD5 Zeitdifferenz 5: Ausführung

13 Zusammenfassung

Zunächst folgt eine kurze Zusammenfassung der Hauptergebnisse der Arbeit, bevor im Anschluss mögliche Anknüpfungspunkte für weitere Forschungsaktivitäten aufgeführt werden.

In der vorliegenden Arbeit wurde eine neue Modelltransformationssprache bestehend aus abstrakter und konkreter Syntax vorgestellt, um Modelltransformationen zwischen Informationsmodellen für Asset-Beschreibungen zu formulieren und maschinell auszuwerten. Die Sprache ist ein Baustein für die Erreichung der semantischen Interoperabilität zwischen Applikationen. Hauptziele bei der Sprachentwicklung waren eine einfach zu verstehende Syntax, die Nutzung des aktuellen Stand der Technik sowie eine einfache Implementierbarkeit, um die Integration in bestehenden Systemen zu fördern. Die Notwendigkeit einer neuen Sprache wurde durch eine ausführliche Evaluation bestehender Transformationssprachen begründet.

In dieser Arbeit wird davon ausgegangen, dass ein Mensch die semantische Gleichheit und die Definition von Regeln zwischen Informationen manuell festlegen muss. Zusätzlich wird angenommen, dass Software-Entwickler gegen vordefinierte Informationsmodelle implementieren. Dies bedeutet, dass sie die Daten in genau diesen Informationsmodell-Strukturen erwarten. Die Nutzung der in dieser Arbeit vorgestellten Modelltransformationssprache ermöglicht die Definition dieser Transformationen unter den gegebenen Randbedingungen. Sie ist damit eine Grundlage für die anwendungsfallbezogene Aufbereitung der Informationen. Die entwickelte Sprache ist praxistauglich, da sie auf der weitverbreiteten Ausdruckssprache BasicOCL aufbaut und nur wenige notwendige Sprachelemente definiert, dabei aber jegliche Art von Regeldefinition unterstützt. Zugehörige Transformationssysteme sind dadurch leicht zu implementieren und können einfach in bestehende Systeme integriert werden.

Für die Anwendung im Kontext von Industrie 4.0 wurde die Abbildung auf Verwaltungsschalen inklusive einiger dabei unterstützender Makros beschrieben. Die Erprobung der Sprache wurde anhand einer prototypischen Realisierung eines Transformationssystems durchgeführt und anhand von drei Anwendungsfällen evaluiert. Es konnte gezeigt werden, dass die gestellten Anforderungen erfüllt werden. Zusätzlich wurden Einflussfaktoren für die Erstellungszeit und Optimierungsempfehlungen hinsichtlich der Funktionsaufrufe des entwickelten Transformationssystems auf Basis von Testreihen beschrieben. Dies ermöglicht bei der Durchführung von Modelltransformationen eine Verbesserung der Performance in der Anwendung.

13.1 Ausblick

Die Themen möglicher zukünftiger Forschungsarbeiten werden in drei Kategorien unterteilt. Die erste Kategorie beschreibt weitere zu untersuchende Anwendungsmöglichkeiten der Sprache sowie Empfehlungen für benötigte Systeme zur einfacheren Nutzung und Erstellung von Transformationsdefinitionen. Verbesserungen für die prototypische Realisierung des Transformationssystems werden in der zweiten Kategorie aufgezeigt. Den Abschluss bilden offene Fragestellung zur Verbreitung und Nutzung der Sprache sowie zur Anwendung bei Verwaltungsschalen.

Anwendungsfälle und benötigte Systeme

In dieser Arbeit wurde die Sprache für Verwaltungsschalen konkretisiert und anhand von drei Anwendungsfällen evaluiert. Eine Evaluation weiterer Modellierungs- und Kommunikationsprotokolle ist nicht erfolgt. Die Erprobung an weiteren Metamodellen, wie z. B. OPC UA oder AutomationML, wäre für den Nachweis einer breiten Anwendung sinnvoll. Dafür könnten weitere spezifische Makros für die konkreten Metamodelle entwickelt werden.

Um eine bessere Nutzung und Akzeptanz der Sprache zu erreichen, wäre eine Unterstützung des Anwenders zielführend. Als Beispiel könnte ein Assistenzsystem entwickelt werden, welches Anwender bei der Erstellung einer Transformationsdefinition unterstützt und sicherstellt, dass die zu erzeugenden Informationsmodell-Instanzen den zugehörigen Templates folgen. Dieses Assistenzsystem könnte ein geeignetes User Interface für die einfachere Erstellung anbieten. Zudem könnten in dem Assistenzsystem KI-basierte Ansätze integriert werden, die Vorschläge für eine mögliche semantische Gleichheit zwischen Modellelementen geben. Erste Ansätze aus dem Natural Language Processing existieren bereits [16, 161] und könnten in einem Assistenzsystem genutzt werden. Zusätzlich könnten Ontologien, die im Kontext des Semantic Web genutzt werden, in Verbindung mit einem zugehörigen Reasoner Vorschläge über semantische Gleichheit geben oder die Abhängigkeit zwischen Informationen aufzuzeigen [162].

Verbesserungen der prototypischen Realisierung des Transformationssystems

Die Realisierung des Transformationssystems wurde im Rahmen der Arbeit prototypisch entwickelt. Einige Funktionen von BasicOCL wurden aus diesem Grund zunächst nicht implementiert und sollten hinzugefügt werden, um die volle Ausdrucksfähigkeit von BasicOCL in Transformationsdefinitionen nutzen zu können. Zusätzlich könnte das Fehlermanagement verbessert werden, sodass dem Anwender hilfreiche Hinweise über aufgetretene Fehler und zugehörige Lösungen angezeigt werden. Für die Überprüfung der spezifizierten Anforderungen und um bei der Weiterentwicklung eine weitgehend fehlerfreie Nutzung beizubehalten, sollten automatische Software-Tests erstellt werden.

Verbreitung und Nutzung für Verwaltungsschalen

Die aktuelle Realisierung des Transformationssystems ist in Python implementiert und nutzt das vom Lehrstuhl für Prozessleittechnik entwickelte Python-SDK PyI40AAS zur Erstellung von Verwaltungsschalen-Teilmodellen. Nächste Schritte wären zum einen die Nutzung anderer Python-SDKs und zum anderen die Umsetzung in anderen Programmiersprachen, z. B. in C# oder Java. Durch zweitgenanntes kann die Sprache auch in

anderen SDKs genutzt und das vorhandene Potential voll ausgeschöpft werden. Dadurch können Online-Transformationen in den jeweiligen SDKs direkt durchgeführt werden, ohne dass ein Adapter für die in dieser Arbeit entwickelte Umsetzung genutzt werden muss.

Anhang

A Makro-Definitionen für Verwaltungsschalen

```
package aas_macros
  macro copySubmodelElementSet(sme_collection: Set(
     SubmodelElement)) : Set(SubmodelElement){
    sme_collection->collect(sme_item | aas_macros::
       copySubmodelElement(sme_item))
  }

  macro copySubmodelElement(sme: SubmodelElement) :
     SubmodelElement{
    if sme.oclIsKindOf(AnnotatedRelationshipElement) then
      aas_macros::copyAnnotatedRelationshipElement(sme)
    else
      if sme.oclIsKindOf(RelationshipElement) then
        aas_macros::copyRelationshipElement(sme)
      else
        if sme.oclIsKindOf(Operation) then
          aas_macros::copyOperation(sme)
        else
          if sme.oclIsKindOf(Property) then
            aas_macros::copyProperty(sme)
          else
            if sme.oclIsKindOf(Capability) then
              aas_macros::copyCapability(sme)
            else
              if sme.oclIsKindOf(BasicEvent) then
                aas_macros::copyBasicEvent(sme)
              else
                if sme.oclIsKindOf(SubmodelElementCollection
                   ) then
                  aas_macros::copySubmodelElementCollection(
                     sme)
                else
                  if sme.oclIsTypeOf(MultiLanguageProperty)
                     then
                    aas_macros::copyMultiLanguageProperty(
                       sme)
```

```
                else
                  if sme.oclIsTypeOf(Range) then
                    aas_macros::copyRange(sme)
                  else
                    if sme.oclIsTypeOf(Blob) then
                      aas_macros::copyBlob(sme)
                    else
                      if sme.oclIsTypeOf(File) then
                        aas_macros::copyFile(sme)
                      else
                        if sme.oclIsTypeOf(
                           ReferenceElement) then
                          aas_macros::copyReferenceElement
                            (sme)
                        else Error
                        endif
                      endif
                    endif
                  endif
                endif
              endif
            endif
          endif
        endif
      endif
    endif
  }

  macro copyRelationshipElement(element: RelationshipElement
     ) : RelationshipElement {
    RelationshipElement{
      idShort: element.idShort,
      first: element.first,
      second: element.second,
      category: element.category,
      description: element.description,
      semanticId: element.semanticId,
      kind: element.kind
    }
  }

  macro copyAnnotatedRelationshipElement(element:
     AnnotatedRelationshipElement) :
     AnnotatedRelationshipElement {
    AnnotatedRelationshipElement{
      idShort: element.idShort,
```

```
    first: element.first,
    second: element.second,
    annotation: aas_macros::copySubmodelElementSet(element
       .annotation),
    category: element.category,
    description: element.description,
    semanticId: element.semanticId,
    kind: element.kind
  }
}

macro copyOperation(element: Operation) : Operation {
  Operation{
    idShort: element.idShort,
    inputVariable: element.inputVariable,
    outputVariable: element.outputVariable,
    inoutputVariable: element.inoutputVariable,
    category: element.category,
    description: element.description,
    semanticId: element.semanticId,
    kind: element.kind
  }
}

macro copyProperty(element: Property) : Property{
  Property {
    idShort: element.idShort,
    valueType: element.valueType,
    value: element.value,
    valueId: element.valueId,
    displayName: element.displayName,
    category: element.category,
    description: element.description,
    semanticId: element.semanticId,
    kind: element.kind
  }
}
macro copyCapability(element: Capability) : Capability{
  Capability {
    idShort: element.idShort,
    displayName: element.displayName,
    category: element.category,
    description: element.description,
    semanticId: element.semanticId,
    kind: element.kind
  }
}
```

```
macro copyBasicEvent(element: BasicEvent) : BasicEvent{
  BasicEvent {
    idShort: element.idShort,
    observed: element.observed,
    displayName: element.displayName,
    category: element.category,
    description: element.description,
    semanticId: element.semanticId,
    kind: element.kind
  }
}

macro copySubmodelElementCollection(element:
   SubmodelElementCollection) : SubmodelElementCollection{
  SubmodelElementCollection {
    idShort: element.idShort,
    value: aas_macros::copySubmodelElementSet(element.
       value),
    ordered: element.ordered,
    allowDuplicates: element.allowDuplicates,
    displayName: element.displayName,
    category: element.category,
    description: element.description,
    semanticId: element.semanticId,
    kind: element.kind
  }
}

macro copyMultiLanguageProperty(element:
   MultiLanguageProperty) : MultiLanguageProperty{
  MultiLanguageProperty {
    idShort: element.idShort,
    value: element.value,
    valueId: element.valueId,
    displayName: element.displayName,
    category: element.category,
    description: element.description,
    semanticId: element.semanticId,
    kind: element.kind
  }
}

macro copyRange(element: Range) : Range{
  Range {
    idShort: element.idShort,
    valueType: element.valueType,
    min: element.min,
```

```
    max: element.max,
    displayName: element.displayName,
    category: element.category,
    description: element.description,
    semanticId: element.semanticId,
    kind: element.kind
  }
}

macro copyBlob(element: Blob) : Blob{
  Blob {
    idShort: element.idShort,
    mimeType: element.mimeType,
    value: element.value,
    displayName: element.displayName,
    category: element.category,
    description: element.description,
    semanticId: element.semanticId,
    kind: element.kind
  }
}

macro copyFile(element: File) : File{
  File {
    idShort: element.idShort,
    mimeType: element.mimeType,
    value: element.value,
    displayName: element.displayName,
    category: element.category,
    description: element.description,
    semanticId: element.semanticId,
    kind: element.kind
  }
}

macro copyReferenceElement(element: ReferenceElement) :
   ReferenceElement{
  ReferenceElement {
    idShort: element.idShort,
    value: element.value,
    displayName: element.displayName,
    category: element.category,
    description: element.description,
    semanticId: element.semanticId,
    kind: element.kind
  }
}
```

```
  macro getSubmodelElementByIdShort(parent: SubmodelElement,
      idShortVar: String) : SubmodelElement{
    if parent.oclIsKindOf(Submodel) then parent.
       submodelElement->select(x | x.idShort = idShortVar)->
       first()
    else
      if parent.oclIsKindOf(Entity) then parent.statement->
         select(x | x.idShort = idShortVar)->first()
      else
        if parent.oclIsKindOf(SubmodelElementCollection)
           then parent.value->select(x | x.idShort =
           idShortVar)->first()
        else Error
        endif
      endif
    endif
  }

  macro getSubmodelElementByIdShortPath(parent:
     SubmodelElement, idShortPath: Sequence(String)) :
     SubmodelElement{
    idShortPath->iterate(x: String; sme: SubmodelElement =
       parent | aas_macros::getSubmodelElementByIdShort(sme,
       x))
  }

  macro getSubmodelElementsBySemanticId(parent:
     SubmodelElement, semanticIdVar: Reference) :
     SubmodelElement{
    if parent.oclIsKindOf(Submodel) then parent.
       submodelElement->select(x | x.semanticId =
       semanticIdVar)
    else
      if parent.oclIsKindOf(Entity) then parent.statement->
         select(x | x.semanticId = semanticIdVar)
      else
        if parent.oclIsKindOf(SubmodelElementCollection)
           then parent.value->select(x | x.semanticId =
           semanticIdVar)
        else Error
        endif
      endif
    endif
  }
end_package
```

B Grammatikdefinition der Transformationssprache

B.1 Grammar_ocl.lark

Terminals

```
NAME_START_CHAR: /[A-Z]/ | "_" | "$" | /[a-z]/
                | /[\u00C0-\u00D6]/ | /[\u00D8-\u00F6]/
                | /[\u00F8-\u02FF]/ | /[\u0370-\u037D]/
                | /[\u037F-\u1FFF]/ | /[\u200C-\u200D]/
                | /[\u2070-\u218F]/ | /[\u2C00-\u2FEF]/
                | /[\u3001-\uD7FF]/ | /[\uF900-\uFDCF]/
                | /[\uFDF0-\uFFFD]/

NAME_CHAR: NAME_START_CHAR | /[0-9]/

SET: /\bSet\b/
BAG: /\bBag\b/
SEQUENCE: /\bSequence\b/
ORDERED_SET: /\bOrderedSet\b/
collection_type_identifier: SET | BAG | SEQUENCE
                          | ORDERED_SET
_TUPLE: /\bTuple\b/

primitive_type: BOOLEAN | INTEGER | REAL | STRING
BOOLEAN: /\bBoolean\b/
INTEGER: /\bInteger\b/
REAL: /\bReal\b/
STRING: /\bString\b/

ocl_type: OCL_ANY | OCL_INVALID | OCL_VOID
OCL_ANY: /\bOclAny\b/
OCL_INVALID: /\bOclInvalid\b/
OCL_VOID: /\bOclVoid\b/

// Bool type
BOOL: /\btrue\b/ | /\bfalse\b/

// self
SELF: /\bself\b/

// Iterators
predefined_iterators: ANY | CLOSURE | COLLECT
                    | COLLECT_NESTED | EXIST | FOR_ALL
                    | IS_UNIQUE | ONE | REJECT | SELECT
                    | SORTED_BY
```

```
ANY: /\bany\b/
CLOSURE: /\bclosure\b/
COLLECT: /\bcollect\b/
COLLECT_NESTED: /\bcollectNested\b/
EXIST: /\bexist\b/
FOR_ALL: /\bforAll\b/
IS_UNIQUE: /\bisUnique\b/
ONE: /\bone\b/
REJECT: /\breject\b/
SELECT: /\bselect\b/
SORTED_BY: /\bsortedBy\b/

_ITERATE: /\biterate\b/

_predefined_operations_a: NOT_EQUAL | ADD | SUB | MUL | DIV
                        | LOWER_THAN | GREATER_THAN | OR
                        | XOR | AND | EQ

_predefined_operations_b.2:
    LOWER_OR_EQUAL_THAN | GREATER_OR_EQUAL_THAN

_predefined_operation_names_a:
    _predefined_operations_a | _predefined_operations_b

EQ: "="
NOT_EQUAL: "<>"
ADD: "+"
SUB: "-"
MUL: "*"
DIV: "/"
LOWER_THAN: "<"
GREATER_THAN: ">"
LOWER_OR_EQUAL_THAN: "<="
GREATER_OR_EQUAL_THAN: ">="
OR: /\bor\b/
XOR: /\bxor\b/
AND: /\band\b/

---------- Production Rules ----------
_predefined_operation_names_b: OCL_AS_SET | OCL_IS_NEW
                             | OCL_IS_INVALID
                             | OCL_AS_TYPE
                             | OCL_IS_TYPE_OF
                             | OCL_IS_KIND_OF
                             | OCL_IS_IN_STATE
                             | OCL_TYPE_OP | OLC_LOCALE
                             | OCL_IS_UNDEFINED | ABS
```

```
                                | FLOOR | ROUND | MAX | MIN
                                | TO_STRING | DIV_OP | MOD
                                | SIZE | CONCAT
                                | SUB_STRING | TO_INTEGER
                                | TO_REAL | TO_UPPER_CASE
                                | TO_LOWER_CASE | INDEX_OF
                                | EQUALS_IGNORE_CASE | AT
                                | CHARACTERS | TO_BOOLEAN
                                | NOT | IMPLIES | INCLUDES
                                | EXCLUDES | COUNT
                                | INCLUDES_ALL
                                | EXCLUDES_ALL | IS_EMPTY
                                | NOT_EMPTY | SUM | PRODUCT
                                | SELECT_BY_KIND
                                | SELECT_BY_TYPE | AS_SET
                                | AS_ORDERED_SET
                                | AS_SEQUENCE | AS_BAG
                                | FLATTEN | UNION
                                | INTERSECTION | INCLUDING
                                | EXCLUDING
                                | SYMMETRIC_DIFFERENCE
                                | APPEND | PREPREND
                                | INSERT_AT
                                | SUB_ORDERED_SET | FIRST
                                | LAST | REVERSE
                                | SUB_SEQUENCE

predefined_operation_names: _predefined_operation_names_a
                          | _predefined_operation_names_b

predefined_operation_names_property_call:
    _predefined_operation_names_b

OCL_AS_SET: /\boclAsSet\b/
OCL_IS_NEW: /\boclIsNew\b/
OCL_IS_INVALID: /\boclIsInvalid\b/
OCL_AS_TYPE: /\boclAsType\b/
OCL_IS_TYPE_OF: /\boclIsTypeOf\b/
OCL_IS_KIND_OF: /\boclIsKindOf\b/
OCL_IS_IN_STATE: /\boclIsInState\b/
OCL_TYPE_OP: /\boclType\b/
OLC_LOCALE: /\boclLocale\b/
OCL_IS_UNDEFINED: /\boclIsUndefined\b/
ABS: /\babs\b/
FLOOR: /\bfloor\b/
ROUND: /\bround\b/
MAX: /\bmax\b/
```

```
MIN: /\bmin\b/
TO_STRING: /\btoString\b/
DIV_OP: /\bdiv\b/
MOD: /\bmod\b/
SIZE: /\bsize\b/
CONCAT: /\bconcat\b/
SUB_STRING:/\bsubstring\b/
TO_INTEGER: /\btoInteger\b/
TO_REAL: /\btoReal\b/
TO_UPPER_CASE: /\btoUpperCase\b/
TO_LOWER_CASE: /\btoLowerCase\b/
INDEX_OF: /\bindexOf\b/
EQUALS_IGNORE_CASE: /\bequalsIgnoreCase\b/
AT: /\bat\b/
CHARACTERS: /\bcharacters\b/
TO_BOOLEAN: /\btoBoolean\b/
NOT: /\bnot\b/
IMPLIES: /\bimplies\b/
INCLUDES: /\bincludes\b/
EXCLUDES: /\bexcludes\b/
COUNT: /\bcount\b/
INCLUDES_ALL: /\bincludesAll\b/
EXCLUDES_ALL: /\bexcludesAll\b/
IS_EMPTY: /\bisEmpty\b/
NOT_EMPTY: /\bnotEmpty\b/
SUM: /\bsum\b/
PRODUCT: /\bproduct\b/
SELECT_BY_KIND: /\bselectByKind\b/
SELECT_BY_TYPE: /\bselectByType\b/
AS_SET: /\basSet\b/
AS_ORDERED_SET: /\basOrderedSet\b/
AS_SEQUENCE: /\basSequence\b/
AS_BAG: /\basBag\b/
FLATTEN: /\bflatten\b/
UNION: /\bunion\b/
INTERSECTION: /\bintersection\b/
INCLUDING: /\bincluding\b/
EXCLUDING: /\bexcluding\b/
SYMMETRIC_DIFFERENCE: /\bsymmetricDifference\b/
APPEND: /\bappend\b/
PREPREND: /\bprepend\b/
INSERT_AT: /\binsertAt\b/
SUB_ORDERED_SET: /\bsubOrderedSet\b/
FIRST: /\bfirst\b/
LAST: /\blast\b/
REVERSE: /\breverse\b/
SUB_SEQUENCE: /\bsubSequence\b/
```

```
_LET: /\blet\b/
_IN: /\bin\b/

_IF: /\bif\b/
_THEN: /\bthen\b/
_ELSE: /\belse\b/
_ENDIF: /\bendif\b/

NULL: /\bnull\b/
INVALID: /\binvalid\b/

---------- Import of lark production rules ----------

%import common.SIGNED_FLOAT
%import common.ESCAPED_STRING

// no whitespaces
%import common.WS
%ignore WS

// line comments
%import common.SQL_COMMENT
%ignore SQL_COMMENT

// multiline comments
%import common.C_COMMENT
%ignore C_COMMENT

---------- Production Rules ----------
simple_name: (NAME_START_CHAR NAME_CHAR*)

path_name: _path_name_a | _path_name_b
_path_name_a: simple_name
_path_name_b: simple_name "::" simple_name

_ocl_expression: _call_exp | variable_exp | let_exp
                | if_exp | literal_exp
                | "(" _ocl_expression ")"

// VariableExp production rules
variable_exp.2: _variable_exp_a | _variable_exp_b
_variable_exp_a.2: path_name
_variable_exp_b.3: SELF

// Literal production rules
?literal_exp: collection_literal_exp | tuple_literal_exp
```

```
            | primitive_literal_exp | type_literal_exp

collection_literal_exp: collection_type_identifier "{"
                        [_collection_literal_parts] "}"

_collection_literal_parts: _collection_literal_part ("," 
                        _collection_literal_parts)?

_collection_literal_part: _collection_literal_part_a
                        | _collection_literal_part_b
_collection_literal_part_a: collection_range
_collection_literal_part_b: _ocl_expression

collection_range: _ocl_expression ".." _ocl_expression

?primitive_literal_exp: integer_literal_exp
                      | real_literal_exp
                      | string_literal_exp
                      | boolean_literal_exp
                      | null_literal_exp
                      | invalid_literal_exp

tuple_literal_exp: _TUPLE "
{" _variable_declaration_list_a "}"

integer_literal_exp: ["+"|"-"] ("0".."9")+

real_literal_exp: SIGNED_FLOAT //["+"|"-"] ((("0".."9")+
                ("e"|"E") ["+"|"-"] ("0".."9")+) |
                ((("0".."9")+ "." (("0".."9")+)? | "."
                ("0".."9")+) (("e"|"E") ["+"|"-"]
                ("0".."9")+)?))

string_literal_exp: ESCAPED_STRING
                  | ESCAPED_STRING ESCAPED_STRING

boolean_literal_exp.3: BOOL

null_literal_exp: NULL

invalid_literal_exp: INVALID

type_literal_exp: type

// CallExp production rules
_call_exp: _feature_call_exp | _loop_exp
```

```
_loop_exp: _iterator_exp | iterate_exp

_iterator_exp: iterator_exp_a
iterator_exp_a: _ocl_expression "->"
                predefined_iterators "("
                [variable_declaration_b [","
                variable_declaration_b] "|"]
                [_ocl_expression] ")"

iterate_exp: _ocl_expression "->" _ITERATE "("
             [variable_declaration_b ";"]
             variable_declaration_c "|"
             _ocl_expression ")"

variable_declaration_a: simple_name ":" type
variable_declaration_b: simple_name [":" type]
variable_declaration_c: simple_name ":" type "="
                        _ocl_expression
variable_declaration_d: simple_name (":" type)?
                        ("=" _ocl_expression)?

type: path_name | collection_type | tuple_type
      | primitive_type | ocl_type

collection_type: collection_type_identifier "(" type ")"

tuple_type: "Tuple" "(" type* ")"

_variable_declaration_list_a: variable_declaration_c
                        ["," _variable_declaration_list_a]

_feature_call_exp: property_call_exp | _operation_call_exp

_operation_call_exp: operation_call_exp_a
                   | operation_call_exp_b
                   | operation_call_exp_c
                   | operation_call_exp_d

operation_call_exp_a: _ocl_expression
                      predefined_operation_names
                      _ocl_expression
operation_call_exp_b: _ocl_expression "->"
                      predefined_operation_names "("
                      [_arguments] ")"
operation_call_exp_c: _ocl_expression "."
                      (predefined_operation_names |
                      simple_name) "(" [_arguments] ")"
```

```
operation_call_exp_d: predefined_operation_names "("
                      [_arguments] ")"

property_call_exp: _property_call_exp_a
_property_call_exp_a: _ocl_expression "." (simple_name |
                  predefined_operation_names_property_call)

_arguments: _ocl_expression ("," _arguments)?

// LetExp production rules
let_exp: _LET variable_declaration_c _let_exp_sub
_let_exp_sub: _let_exp_sub_a | _let_exp_sub_b
_let_exp_sub_a: "," variable_declaration_c _let_exp_sub
_let_exp_sub_b: _IN _ocl_expression

// IfExp production rules
if_exp: _IF _ocl_expression _THEN _ocl_expression _ELSE
        _ocl_expression _ENDIF

_parameters: variable_declaration_d ("," _parameters)?
```

B.2 Grammar_mtl.lark

```
------------------------- Terminals -------------------------
_TRANSFORMATION_DEFINITION: /\btransformationDefinition\b/
_SOURCE_TEMPLATE:           /\bsourceTemplate\b/
_TARGET_TEMPLATE:           /\btargetTemplate\b/
_VALUE:                     /\bvalue\b/
_MACRO:                     /\bmacro\b/
_PACKAGE:                   /\bpackage\b/
_END_PACKAGE:               /\bend_package\b/

------------------ Import of OCL production rules ------------------
%import .grammar_ocl    (simple_name,
                         _parameters, type,
                         _ocl_expression,
                         _arguments,
                         variable_declaration_a,
                         literal_exp,
                         string_literal_exp)

------------------ Extending of OCL production rules ------------------
%extend literal_exp:      object_literal_exp
%extend _ocl_expression: macro_call_exp
```

```
------------------------ No Whitespaces ------------------------
%import common.WS
%ignore WS

------------------ Line and multiline Comments -----------------
%import common.SQL_COMMENT
%ignore SQL_COMMENT

%import common.C_COMMENT
%ignore C_COMMENT

---------------------- New Production Rules ----------------------
macro_decl:
    _MACRO simple_name "(" (_parameters)? ")" ":" type
    "{" _ocl_expression "}"

macro_call_exp:
    (simple_name "::")? simple_name "(" _arguments? ")"

object_literal_exp:
    type "{" _attribute_binding_list? "}"

attribute_binding:
    simple_name ":" _ocl_expression

_attribute_binding_list:
    attribute_binding ("," _attribute_binding_list)?

_informationmodel_template:
    variable_declaration_a ["->" literal_exp]

_informationmodel_template_list:
    _informationmodel_template ","
    _informationmodel_template_list)?

transformation_definition:
    _TRANSFORMATION_DEFINITION [simple_name]
    _SOURCE_TEMPLATE ":" [_informationmodel_template_list]
    _TARGET_TEMPLATE ":" _informationmodel_template _VALUE
    ":" object_literal_exp

package_decl:
    _PACKAGE simple_name [transformation_definition]
    macro_decl* _END_PACKAGE
```

C Python-Klassendefinition der abstrakten Syntaxklassen

C.1 ast_ocl.py

```
import abc
import math
import operator
import itertools
from typing import Any, Callable, Dict, Iterable, List, Optional,
    Tuple, TypeVar, Union


# -*- encoding: utf-8 -*-


class Environment(abc.ABC):
    def __init__(self):
        self.variables: Dict[str, Any] = {}
        self.types: Dict[str, type] = {}

    def add_var(self, name: str, _type: type = None, value: Any = None
        ) -> None:
        if name in self.variables:
            raise KeyError(f"Variable {name} already exist!")
        self.variables[name] = [_type, value]

    def set_var_value(self, name: str, value: Any) -> None:
        if name not in self.variables:
            raise KeyError(f"Variable {name} hasn't been defined yet!"
                )
        (self.variables[name])[1] = value

    def set_var_type(self, name: str, _type: type) -> None:
        if name not in self.variables:
            raise KeyError(f"Variable {name} hasn't been defined yet!"
                )
        (self.variables[name])[0] = _type

    def del_var(self, name: str) -> None:
        del self.variables[name]

    def get_var_type(self, name: str) -> type:
        if name not in self.variables:
            raise KeyError(f"Variable {name} hasn't been defined yet!"
                )
        return (self.variables[name])[0]
```

```
    def get_var_value(self, name: str) -> Any:
        if name not in self.variables:
            # Test if name is in self.types todo: Hack for parser
                problems, when getting VariableExp instead of TypeExp
            if name not in self.types:
                raise KeyError(f"Variable {name} hasn't been defined
                    yet!")
            return self.types[name]
        return (self.variables[name])[1]


class LocalEnvironment(Environment):
    def __init__(self):
        super().__init__()


class GlobalEnvironment(Environment):
    def __init__(self):
        super().__init__()


class Token(abc.ABC):
    def __init__(self, *tokens):
        self.tokens = None


class OclExpression(Token, abc.ABC):
    @abc.abstractmethod
    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        pass


class PrimitiveType(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[str, ...] = tokens

    def evaluate_value(self) -> str:
        return self.tokens[0]


class OclType(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[str, ...] = tokens

    def evaluate_value(self) -> str:
        return self.tokens[0]
```

```
class Type(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[CollectionType, TupleType,
            PrimitiveType, OclType], ...] = tokens

    def evaluate_value(self) -> str:
        return self.tokens[0].evaluate_value()


class Invalid:
    def __init__(self):
        raise Exception("An error has occurred")


class CollectionTypeIdentifier(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[str, ...] = tokens

    def evaluate_value(self) -> str:
        return self.tokens[0]


class CollectionType(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[CollectionTypeIdentifier, Type], ...]
            = tokens

    def evaluate_value(self) -> str:
        return self.tokens[0].evaluate_value() + "(" + self.tokens[1].
            evaluate_value() + ")"


class TupleType(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Type, ...] = tokens

    def evaluate_value(self) -> str:
        tmp_type: str = "Tuple("
        first = True
        for t in self.tokens:
            if not first:
                tmp_type += ","
            tmp_type += t.evaluate_value()
            first = False
        tmp_type += ")"
```

```
        return tmp_type


class SimpleName(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[str, ...] = tokens

    def evaluate_value(self) -> str:
        return ''.join(self.tokens)


class PathName(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[PathName, SimpleName], ...] = tokens

    def evaluate_value(self) -> str:
        if len(self.tokens) == 2:
            return self.tokens[0].evaluate_value() + "::" + self.
                tokens[1].evaluate_value()
        else:
            return self.tokens[0].evaluate_value()


class StringName(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[str, ...] = tokens

    def evaluate_value(self) -> str:
        return ''.join(self.tokens)


class VariableDeclaration(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[SimpleName, Type, OclExpression],
            ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> str:
        """
        adds the variable to the variable_list and returns the
            variable name
        """
        var_name = ""
        var_type = None
        var_value = None
        for i in self.tokens:
```

```
            if isinstance(i, SimpleName):
                var_name = i.evaluate_value()
            if isinstance(i, Type):
                var_type = global_env.get_type(i.evaluate_value())
            if isinstance(i, OclExpression):
                var_value = i.evaluate_value(global_env, local_env)

        local_env.add_var(var_name, var_type, var_value)
        return var_name


class LiteralExp(OclExpression, abc.ABC):
    def __init__(self):
        super().__init__()


class EnumLiteralExp(LiteralExp):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[Union[SimpleName], ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        try:
            return local_env.get_var_value(self.tokens[0].
                evaluate_value() + "::" + self.tokens[1].
                evaluate_value())
        except KeyError:
            return global_env.get_var_value(self.tokens[0].
                evaluate_value() + "::" + self.tokens[1].
                evaluate_value())


class CollectionRange(Token):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[OclExpression, ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> List[Union[str, int]]:
        lower: Union[str, int] = self.tokens[0].evaluate_value(
            global_env, local_env)
        upper: Union[str, int] = self.tokens[1].evaluate_value(
            global_env, local_env)
        value: List[Union[str, int]] = []
        if isinstance(lower, str) and isinstance(upper, str):
            # Todo: Only useful with StringLiteralExp with single
                letter
            for i in range(ord(lower[1]), ord(upper[1]) + 1):
                value.append(chr(i))
```

```
        elif isinstance(lower, int) and isinstance(upper, int):
            for i in range(lower, upper + 1):
                value.append(i)
        else:
            raise KeyError("Upper and lower boundary do not have the
                same type")
        return value


def list_to_set(tmp_list: Iterable) -> set:
    tmp_set = set()
    for i in tmp_list:
        if isinstance(i, list) or isinstance(i, set):
            tmp_set.update(list_to_set(i))
        else:
            tmp_set.add(i)
    return tmp_set


class CollectionLiteralExp(LiteralExp):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[Union[CollectionTypeIdentifier, List[
            CollectionRange, OclExpression]], ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        collection_type: str = self.tokens[0].evaluate_value()  # type
            : ignore
        collection_item_list = []
        for i in self.tokens:
            if isinstance(i, CollectionRange):
                tmp_list = i.evaluate_value(global_env, local_env)
                for t in tmp_list:
                    collection_item_list.append(t)
            if isinstance(i, OclExpression):
                collection_item_list.append(i.evaluate_value(
                    global_env, local_env))

        if collection_type == "Set":
            return list_to_set(collection_item_list)
        elif collection_type == "Bag":
            return collection_item_list
        elif collection_type == "Sequence":
            return collection_item_list
        elif collection_type == "OrderedSet":
            return list(list_to_set(collection_item_list))  # todo:
                should be an ordered set
        else:
            raise KeyError
```

```
class TupleLiteralExp(LiteralExp):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[VariableDeclaration, ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> tuple:
        tuple_list = []
        for t in self.tokens:
            try:
                tuple_list.append(local_env.get_var_value(t.
                    evaluate_value(global_env, local_env)))
            except KeyError:
                tuple_list.append(global_env.get_var_value(t.
                    evaluate_value(global_env, local_env)))

        return tuple(tuple_list)


class PrimitiveLiteralExp(LiteralExp, abc.ABC):
    def __init__(self):
        super().__init__()


class IntegerLiteralExp(PrimitiveLiteralExp):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[str, ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> int:
        return int(''.join(self.tokens))


class RealLiteralExp(PrimitiveLiteralExp):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[float, ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> float:
        return float(self.tokens[0])


class StringLiteralExp(PrimitiveLiteralExp):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[str, ...] = tokens
```

```
    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> str:
        tmp_string = ''
        for i in self.tokens:
            if isinstance(i, str):
                tmp_string += i.strip('"')
        return tmp_string


class BooleanLiteralExp(PrimitiveLiteralExp):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[str, ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> bool:
        if self.tokens[0] == "true":
            return True
        else:
            return False


class NullLiteralExp(PrimitiveLiteralExp):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[str, ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        return None


class InvalidLiteralExp(PrimitiveLiteralExp):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[Invalid, ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        return self.tokens[0].evaluate_value()


class TypeLiteralExp(LiteralExp):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[Type, ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> type:
```

```
        return global_env.get_type(self.tokens[0].evaluate_value())


class VariableExp(OclExpression):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[Union[SimpleName, str], ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        if isinstance(self.tokens[0], PathName):
            try:
                return local_env.get_var_value(self.tokens[0].
                    evaluate_value())
            except KeyError:
                return global_env.get_var_value(self.tokens[0].
                    evaluate_value())
        else:
            return local_env.get_var_value("self")


class IfExp(OclExpression):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[OclExpression, ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        if self.tokens[0].evaluate_value(global_env, local_env):
            return self.tokens[1].evaluate_value(global_env, local_env
                )
        else:
            return self.tokens[2].evaluate_value(global_env, local_env
                )


class LetExp(OclExpression):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[Union[VariableDeclaration, OclExpression],
            ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        # add variables to env
        var_list = []
        for i in self.tokens:
            if isinstance(i, VariableDeclaration):
                var_list.append(i.evaluate_value(global_env, local_env
                    ))
```

```
        # eval expression
        result = self.tokens[len(self.tokens) - 1].evaluate_value(
            global_env, local_env)
        # remove variables vom env
        for var in var_list:
            local_env.del_var(var)
        return result


class PropertyCallExp(OclExpression):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[OclExpression, SimpleName,
            PredefinedOperationNamesPropertyCall], ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        return getattr(self.tokens[0].evaluate_value(global_env,
            local_env),
                       global_env.
                           get_correct_attribute_name_by_sdk_type(type
                           (
                            self.tokens[0].evaluate_value(global_env,
                                local_env)),
                            self.tokens[1].evaluate_value()))  # type:
                                ignore


class PredefinedIterators(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[str, ...] = tokens

    def evaluate_value(self) -> Callable[[Iterable, Optional[str],
        Optional[str], OclExpression], Any]:
        name_function_map: Dict[str, Callable[[Iterable, Optional[str
            ], Optional[str], OclExpression, GlobalEnvironment,
            LocalEnvironment], Any]] = {
            "any": _any,
            "closure": _closure,
            "collect": _collect,
            "collectNested": _collect_nested,
            "exist": _exist,
            "forAll": _for_all,
            "isUnique": _is_unique,
            "one": _one,
            "reject": _reject,
            "select": _select,
            "sortedBy": _sorted_by
```

```
        }
        return name_function_map[self.tokens[0]]


def _any(parent: Iterable, iterator_1: Optional[str], iterator_2:
    Optional[str], expression: OclExpression, global_env:
    GlobalEnvironment, local_env: LocalEnvironment) -> Any:
    if iterator_2 is not None:
        raise ValueError("->any() only supports one iterator variable,
            two given!")
    for element in parent:
        local_env.set_var_value(iterator_1, element)
        result = expression.evaluate_value(global_env, local_env)
        if isinstance(result, Invalid):
            return result
        if result:
            return element
    return Invalid


def _closure(parent: Iterable, iterator_1: Optional[str], iterator_2:
    Optional[str], expression: OclExpression, global_env:
    GlobalEnvironment, local_env: LocalEnvironment) -> Any:
    pass


def _collect(parent: Iterable, iterator_1: Optional[str], iterator_2:
    Optional[str], expression: OclExpression, global_env:
    GlobalEnvironment, local_env: LocalEnvironment) -> Any:
    if iterator_2 is not None:
        raise ValueError("->collect() only supports one iterator
            variable, two given!")
    res = []
    for element in parent:
        local_env.set_var_value(iterator_1, value=element)
        if isinstance(element, (tuple, list, set)):
            # flatten
            for nested_element in _collect(element, iterator_1, None,
                expression, global_env, local_env):
                res.append(nested_element)
        else:
            res.append(expression.evaluate_value(global_env, local_env
                ))
    return res


def _collect_nested(parent: Iterable, iterator_1: Optional[str],
    iterator_2: Optional[str], expression: OclExpression, global_env:
    GlobalEnvironment, local_env: LocalEnvironment) -> Any:
    if iterator_2 is not None:
```

```
            raise ValueError("->collectNested() only supports one iterator
                variable, two given!")
        res = []
        for element in parent:
            local_env.set_var_value(iterator_1, element)
            res.append(expression.evaluate_value(global_env, local_env))
        return res


def _exist(parent: Iterable, iterator_1: Optional[str], iterator_2:
    Optional[str], expression: OclExpression, global_env:
    GlobalEnvironment, local_env: LocalEnvironment) -> bool:
        if iterator_2 is not None:
            raise ValueError("->exists() only supports one iterator
                variable, two given!")
        for element in parent:
            local_env.set_var_value(iterator_1, element)
            if expression.evaluate_value(global_env, local_env):
                return True
        return False


def _for_all(parent: Iterable, iterator_1: Optional[str], iterator_2:
    Optional[str], expression: OclExpression, global_env:
    GlobalEnvironment, local_env: LocalEnvironment) -> bool:
        if iterator_2 is not None:
            raise ValueError("->forAll() only supports one iterator
                variable, two given!")
        for element in parent:
            local_env.set_var_value(iterator_1, element)
            if not expression.evaluate_value(global_env, local_env):
                return False
        return True


def _is_unique(parent: Iterable, iterator_1: Optional[str], iterator_2
    : Optional[str], expression: OclExpression, global_env:
    GlobalEnvironment, local_env: LocalEnvironment) -> Any:
        if iterator_2 is not None:
            raise ValueError("->isUnique() only supports one iterator
                variable, two given!")
        res = []
        for element in parent:
            local_env.set_var_value(iterator_1, element)
            result = expression.evaluate_value(global_env, local_env)
            if isinstance(result, Invalid):
                return result
            if result is not None:
                if result in res:
                    return False
```

```
            res.append(result)
    return True


def _one(parent: Iterable, iterator_1: Optional[str], iterator_2:
    Optional[str], expression: OclExpression, global_env:
    GlobalEnvironment, local_env: LocalEnvironment) -> Any:
    if iterator_2 is not None:
        raise ValueError("->one() only supports one iterator variable,
            two given!")
    was_true = False
    for element in parent:
        local_env.set_var_value(iterator_1, element)
        result = expression.evaluate_value(global_env, local_env)
        if isinstance(result, Invalid):
            return result
        if result:
            if was_true:
                return False
            was_true = True
    return False


def _reject(parent: Iterable, iterator_1: Optional[str], iterator_2:
    Optional[str], expression: OclExpression, global_env:
    GlobalEnvironment, local_env: LocalEnvironment) -> List[Any]:
    if iterator_2 is not None:
        raise ValueError("->reject() only supports one iterator
            variable, two given!")
    res = []
    for element in parent:
        local_env.set_var_value(iterator_1, element)
        if not expression.evaluate_value(global_env, local_env):
            res.append(element)
    return res


def _select(parent: Iterable, iterator_1: Optional[str], iterator_2:
    Optional[str], expression: OclExpression, global_env:
    GlobalEnvironment, local_env: LocalEnvironment) -> List[Any]:
    if iterator_2 is not None:
        raise ValueError("->select() only supports one iterator
            variable, two given!")
    res = []
    for element in parent:
        local_env.set_var_value(iterator_1, element)
        if expression.evaluate_value(global_env, local_env):
            res.append(element)
    return res
```

```
def _sorted_by(parent: Iterable, iterator_1: Optional[str], iterator_2
    : Optional[str], expression: OclExpression, global_env:
    GlobalEnvironment, local_env: LocalEnvironment) -> Any:
    pass


class IteratorExp(OclExpression):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[OclExpression, PredefinedIterators,
            VariableDeclaration], ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        iterator_function: Callable[[Iterable, Optional[str], Optional
            [str], OclExpression, GlobalEnvironment, LocalEnvironment
            ], Any] = self.tokens[1].evaluate_value()  # type: ignore
        parent: Iterable = self.tokens[0].evaluate_value(global_env,
            local_env)  # type: ignore
        iterator_1: Optional[str] = None
        if self.tokens[2] is not None:
            iterator_1: Optional[str] = self.tokens[2].evaluate_value(
                global_env, local_env)  # type: ignore
        iterator_2: Optional[str] = None
        offset = 5 - len(self.tokens)
        if offset == 0:
            if self.tokens[3] is not None:
                iterator_2: Optional[str] = self.tokens[3].
                    evaluate_value(global_env, local_env)  # type:
                    ignore
        if iterator_1 is None:
            iterator_1 = "_"
            while True:
                try:
                    local_env.get_var_value(iterator_1)
                    iterator_1 += "_"
                except KeyError:
                    local_env.add_var("_")
                    break
        res = iterator_function(parent, iterator_1, iterator_2, self.
            tokens[4 - offset], global_env, local_env)  # type: ignore
        local_env.del_var(iterator_1)
        if iterator_2 is not None:
            local_env.del_var(iterator_2)
        return res


class IterateExp(OclExpression):
    def __init__(self, *tokens):
```

```
        super().__init__(*tokens)
        self.tokens: Tuple[Union[OclExpression, VariableDeclaration],
            ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:  # todo: tbd
        parent = self.tokens[0].evaluate_value(global_env, local_env)
        iterator_1: Optional[str] = None
        if self.tokens[1] is not None:
            iterator_1: Optional[str] = self.tokens[1].evaluate_value(
                global_env, local_env)  # type: ignore
        if iterator_1 is None:
            iterator_1 = "_"
            while True:
                try:
                    local_env.get_var_value(iterator_1)
                    iterator_1 += "_"
                except KeyError:
                    local_env.add_var("_")
                    break
        iterator_2: Optional[str] = self.tokens[2].evaluate_value(
            global_env, local_env)  # type: ignore
        for element in parent:
            local_env.set_var_value(iterator_1, element)
            local_env.set_var_value(iterator_2, self.tokens[3].
                evaluate_value(global_env, local_env))
        result = local_env.get_var_value(iterator_2)
        local_env.del_var(iterator_1)
        local_env.del_var(iterator_2)
        return result


PT = TypeVar("PT", bool, str, float, int)


def _flatten(c):
    ret = []
    for itm in c:
        if not isinstance(itm, list) and not isinstance(itm, set):
            ret.append(itm)
            continue
        for itm_ in _flatten(itm):
            ret.append(itm_)
    return ret


def _all_params(fn: Callable) -> Callable:
    def helper(self_: Any, list_: List):
        if not isinstance(self_, list) and not isinstance(self_, set):
            self_ = [self_]
```

```
        return fn(*{*self_, *list_})
    return helper


class PredefinedOperationNames(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[str, ...] = tokens

    def evaluate_value(self) -> Union[Callable[[Any, Any], PT],
        Callable[[Any, List[Any]], PT]]:
        name_function_map: Dict[str, Optional[Union[Callable[[Any, Any
            ], PT], Tuple[Callable[[Any, List[Any]], PT], int]]]] = {
            "=": operator.eq,
            "<>": operator.ne,
            "+": operator.add,
            "-": operator.sub,
            "*": operator.mul,
            "/": operator.truediv,
            "<": operator.lt,
            ">": operator.gt,
            "<=": operator.le,
            ">=": operator.ge,
            "or": operator.or_,
            "xor": operator.xor,
            "and": operator.and_,
            "oclAsSet": None,
            "oclIsNew": None,  # for pre- and postconditions
            "oclIsInvalid": (lambda x: x is Invalid, 0),
            "oclAsType": None,
            "oclIsTypeOf": (lambda x, y: type(x) is y, 1),
            "oclIsKindOf": (lambda x, y: isinstance(x, y), 1),
            "oclIsInState": None,  # for state machines
            "oclType": None,  # not implemented
            "oclLocale": None,
            "oclIsUndefined": None,  # not implemented
            "abs": (operator.abs, 0),
            "floor": (math.floor, 0),
            "round": (round, 0),
            "max": _all_params(max),
            "min": _all_params(min),
            "toString": (str, 0),
            "div": (operator.floordiv, 1),
            "mod": (operator.mod, 1),
            "size": (len, 0),
            "concat": (operator.concat, 1),
            "substring": (lambda s, start, end: s[start:end], 2),
            "toInteger": (int, 0),
            "toReal": (float, 0),
            "toUpperCase": (lambda s: s.upper(), 0),
```

```
        "toLowerCase": (lambda s: s.lower(), 0),
        "indexOf": lambda s, n: s.find(n[0]),
        "equalsIgnoreCase": lambda s, s2: s.upper() == s2[0].upper
            (),
        "at": (lambda s, pos: s[pos], 1),
        "characters": (lambda s: [x for x in s], 0),
        "toBoolean": lambda a, _: not not a,
        "not": operator.neg,
        "implies": lambda a, b: not a or a and b,
        "includes": (lambda c, e: e in c, 1),
        "excludes": (lambda c, e: e not in c, 1),
        "count": (len, 0),
        "includesAll": (lambda c1, c2: all(e in c1 for e in c2),
            1),
        "excludesAll": (lambda c1, c2: all(e not in c1 for e in c2
            ), 1),
        "isEmpty": (lambda c: len(c) == 0, 0),
        "notEmpty": (lambda c: len(c) != 0, 0),
        "sum": (lambda c: sum(c), 0),
        "product": (lambda c1, c2: itertools.product(c1, c2), 1),
        "selectByKind": (lambda c, k: filter(lambda e: type(e) is
            k, c), 1),
        "selectByType": (lambda c, k: filter(lambda e: isinstance(
            e, k), c), 1),
        "asSet": (set, 0),
        "asOrderedSet": None,  # ordered sets aren't implemented
        "asSequence": (list, 0),
        "asBag": (list, 0),
        "flatten": (_flatten, 0),
        "union": (lambda c1, c2: c1 + c2, 1),
        "intersection": (lambda c1, c2: c1.intersection(c2), 1),
        "including": (lambda c, e: c + [e] if isinstance(c, list)
            else {*c, e}, 1),
        "excluding": (lambda c, l: filter(lambda e: e is not l, c)
            , 1),
        "symmetricDifference": (lambda c1, c2: c1.
            symmetric_difference(c2), 1),
        "append": (lambda c, e: c + (e,), 1),
        "prepend": (lambda c, e: (e,) + c, 1),
        "insertAt": None,  # ordered sets aren't implemented
        "subOrderedSet": None,  # ordered sets aren't implemented
        "first": (lambda c: c[0], 0),
        "last": (lambda c: c[-1], 0),
        "reverse": (lambda c: c.reverse(), 0),
        "subSequence": (lambda c, l, u: c[l:u + 1], 2)
    }
    function_name = self.tokens[0]
    function_maybe: Optional[Union[Callable[[Any, Any], PT], Tuple
        [Callable[[Any, List[Any]], PT], int]]] =
        name_function_map[function_name]
```

```
        if function_maybe is None:
            raise NotImplementedError(f"{function_name}() is not
                implemented!")
        if not isinstance(function_maybe, tuple):
            return name_function_map[function_name]

        def fn_wrap(self_: Any, list_: List) -> PT:
            if len(list_) != function_maybe[1]:
                raise ValueError(f"{function_name}() expects {
                    function_maybe[1]} parameters, but {len(list_)}
                    given: "
                                 + str(list_))
            return function_maybe[0](self_, *list_)

        return fn_wrap


class PredefinedOperationNamesPropertyCall(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[str, ...] = tokens

    def evaluate_value(self) -> str:
        return self.tokens[0]


class OperationCallExpA(OclExpression):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[OclExpression,
            PredefinedOperationNames], ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        if isinstance(self.tokens[1], PredefinedOperationNames):
            operation = self.tokens[1].evaluate_value()
        else:
            raise NotImplementedError()
        var_1 = self.tokens[0].evaluate_value(global_env, local_env)
            # type: ignore
        var_2 = self.tokens[2].evaluate_value(global_env, local_env)
            # type: ignore

        return operation(var_1, var_2)  # type: ignore


class OperationCallExpB(OclExpression):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[OclExpression,
```

```
            PredefinedOperationNames], ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        parent = self.tokens[0].evaluate_value(global_env, local_env)
            # type: ignore
        operation = self.tokens[1].evaluate_value()  # type: ignore
        var_list = []
        for i in range(2, len(self.tokens)):
            var_list.append(self.tokens[i].evaluate_value(global_env,
                local_env))  # type: ignore
        return operation(parent, var_list)


class OperationCallExpC(OclExpression):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[OclExpression,
            PredefinedOperationNames], ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        parent = self.tokens[0].evaluate_value(global_env, local_env)
            # type: ignore
        if isinstance(self.tokens[1], PredefinedOperationNames):
            operation = self.tokens[1].evaluate_value()
        elif isinstance(self.tokens[1], SimpleName):
            operation = getattr(parent, self.tokens[1].evaluate_value
                ())
        else:
            raise NotImplementedError()
        var_list = []
        for i in range(2, len(self.tokens)):
            var_list.append(self.tokens[i].evaluate_value(global_env,
                local_env))  # type: ignore
        return operation(parent, var_list)


class OperationCallExpD(OclExpression):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[OclExpression,
            PredefinedOperationNames], ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        operation = self.tokens[0].evaluate_value()  # type: ignore
        var_list = []
        for i in range(1, len(self.tokens)):
            var_list.append(self.tokens[i].evaluate_value(global_env,
```

```
            local_env))  # type: ignore
        return operation(None, var_list)
```

C.2 ast_mtl.py

```
import time
import typing
from typing import Any, Dict, List, Optional, Tuple, Union, Set
from lark.lark import Lark
from .ast_ocl import GlobalEnvironment, LocalEnvironment, SimpleName,
    Token, VariableDeclaration, Type, OclExpression, StringLiteralExp,
     LiteralExp


# -*- encoding: utf-8 -*-


class GlobalEnvironmentMTL(GlobalEnvironment):
    def __init__(self):
        """
        :ivar variables: Dict of global variable names and their type
           and value. Dict[str, [type, Any]]
        :ivar macros: Dict of macro names and their parameters,
           OclExpression and type.
                    Dict[str, [List[VariableDeclaration],
                       OclExpression, type]
        :ivar types: Dict of globale type names and their type. Dict[
           str, type]
        """
        super().__init__()
        self.variables: Dict[str, Any] = {}
        self.macros: Dict[str, Any] = {}
        self.types: Dict[str, type] = {}
        self.create_types: Dict[str, type] = {}
        self.inverted_types: Dict[type, str] = {}
        self.meta_model_types_to_sdk_types: Dict[str, Union[type, Dict
           [str, Union[type, Dict[str, str]]]]] = {}
        self.sdk_types_to_meta_model_types: Dict[type, Union[str, Dict
           [str, Union[type, Dict[str, str]]]]] = {}

        # add some ocl basic types and variables
        self.add_var("True", bool, True)
        self.add_var("False", bool, False)
        self.add_var("self")

    def call_macro(self, name: str, arg_list: List[Any]) -> Any:
        if name not in self.macros:
            raise KeyError(f"Macro {name} hasn't been defined yet!")
        param_list, ocl_expression, _type = self.macros[name]
```

```
        if len(arg_list) != len(param_list):
            raise KeyError(f"Macro param length and given args length
                is not equal")

        # add variables to env todo: check if type of arg und param is
            the same
        var_list = []
        local_env = LocalEnvironment()
        for i in range(len(param_list)):
            var_name = param_list[i].evaluate_value(self, local_env)
            local_env.set_var_value(var_name, arg_list[i])
            var_list.append(var_name)
        # eval expression
        result = ocl_expression.evaluate_value(self, local_env)
        # remove variables vom env
        for var in var_list:
            local_env.del_var(var)
        return result

    def add_macro(self, name: str, param_list: List["
        VariableDeclaration"], ocl_expression: "OclExpression", _type:
        type = None) -> None:
        if name in self.macros:
            raise KeyError(f"Macro {name} already exist!")
        self.macros[name] = [param_list, ocl_expression, _type]

    def del_macro(self, name: str) -> None:
        del self.macros[name]

    def get_macro_type(self, name: str) -> type:
        if name not in self.macros:
            raise KeyError(f"Macro {name} hasn't been defined yet!")
        return (self.macros[name])[2]

    def add_type(self, name: str, _type: type) -> None:
        self.types[name] = _type

    def del_type(self, name: str) -> None:
        del self.types[name]

    def get_type(self, name: str) -> type:
        if name.startswith("Set"):
            item_name = name[4:len(name) - 1]
            if item_name not in self.types:
                raise KeyError(f"Type {item_name} hasn't been defined
                    yet!")
            return Set[self.types[item_name]]
        elif name.startswith("Bag"):
            item_name = name[4:len(name) - 1]
            if item_name not in self.types:
```

```
            raise KeyError(f"Type {item_name} hasn't been defined
                yet!")
        return List[self.types[item_name]]
    elif name.startswith("Sequence"):
        item_name = name[9:len(name) - 1]
        if item_name not in self.types:
            raise KeyError(f"Type {item_name} hasn't been defined
                yet!")
        return List[self.types[item_name]]
    elif name.startswith("OrderedSet"):
        item_name = name[11:len(name) - 1]
        if item_name not in self.types:
            raise KeyError(f"Type {item_name} hasn't been defined
                yet!")
        return Set[self.types[item_name]]
    if name not in self.types:
        raise KeyError(f"Type {name} hasn't been defined yet!")
    return self.types[name]

def add_create_type(self, name: str, _type: type) -> None:
    self.create_types[name] = _type

def del_create_type(self, name: str) -> None:
    del self.create_types[name]

def get_create_type(self, name: str) -> type:
    if name.startswith("Set"):
        item_name = name[4:len(name) - 1]
        if item_name not in self.create_types:
            raise KeyError(f"Type {item_name} hasn't been defined
                yet!")
        return Set[self.create_types[item_name]]
    elif name.startswith("Bag"):
        item_name = name[4:len(name) - 1]
        if item_name not in self.create_types:
            raise KeyError(f"Type {item_name} hasn't been defined
                yet!")
        return List[self.create_types[item_name]]
    elif name.startswith("Sequence"):
        item_name = name[9:len(name) - 1]
        if item_name not in self.create_types:
            raise KeyError(f"Type {item_name} hasn't been defined
                yet!")
        return List[self.create_types[item_name]]
    elif name.startswith("OrderedSet"):
        item_name = name[11:len(name) - 1]
        if item_name not in self.create_types:
            raise KeyError(f"Type {item_name} hasn't been defined
                yet!")
        return Set[self.create_types[item_name]]
```

```
        if name not in self.types:
            raise KeyError(f"Type {name} hasn't been defined yet!")
        return self.create_types[name]

    def add_inverted_type(self, _type: type, name: str) -> None:
        self.inverted_types[_type] = name

    def del_inverted_type(self, _type: type) -> None:
        del self.inverted_types[_type]

    def get_inverted_type(self, _type: type) -> str:
        if _type not in self.inverted_types:
            raise KeyError(f"Type {_type} hasn't been defined yet!")
        return self.inverted_types[_type]

    def add_meta_model_types_to_sdk_types(self, meta_model_types: Dict
        [str, Union[type, Dict[str, Dict[str, str]]]]) -> None:
        self.meta_model_types_to_sdk_types = meta_model_types
        for type_name in meta_model_types:
            if isinstance(meta_model_types[type_name], type):
                self.add_type(type_name, meta_model_types[type_name])
                    # type: ignore
                self.add_create_type(type_name, meta_model_types[
                    type_name])  # type: ignore
                self.add_inverted_type(meta_model_types[type_name],
                    type_name)
                self.sdk_types_to_meta_model_types[meta_model_types[
                    type_name]] = type_name
            else:
                self.add_type(type_name, (meta_model_types[type_name])
                    ["class"])  # type: ignore
                self.add_create_type(type_name, (meta_model_types[
                    type_name])["create"])  # type: ignore
                self.add_inverted_type((meta_model_types[type_name])["
                    class"], type_name)
                self.sdk_types_to_meta_model_types[meta_model_types[
                    type_name]["class"]] = \
                    {"class": type_name, "create": meta_model_types[
                        type_name]["create"],
                     "attributes": meta_model_types[type_name]["
                         attributes"]}

    def get_correct_attribute_name_by_meta_model_type_name(self,
        type_name: str, attribute_name: str) -> str:
        return ((self.meta_model_types_to_sdk_types[type_name])["
            attributes"])[attribute_name]  # type: ignore

    def add_sdk_types_to_meta_model_types(self, sdk_types: Dict[type,
        Union[str, Dict[str, Union[type, Dict[str, str]]]]]) -> None:
        for sdk_type in sdk_types:
```

```
            self.sdk_types_to_meta_model_types[sdk_type] = sdk_types[
                sdk_type]
            if isinstance(sdk_types[sdk_type], str):
                self.add_inverted_type(sdk_type, sdk_types[sdk_type])
                    # type: ignore
            else:
                self.add_inverted_type(sdk_type, (sdk_types[sdk_type])
                    ["class"])  # type: ignore

    def get_correct_attribute_name_by_sdk_type(self, _type: type,
        attribute_name: str) -> str:
        return ((self.sdk_types_to_meta_model_types[_type])["
            attributes"])[attribute_name]  # type: ignore


class MacroDecl(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[SimpleName, VariableDeclaration, Type
            , OclExpression], ...] = tokens

    def evaluate_value(self, global_env: GlobalEnvironment,
        package_name: Optional[str] = None) -> str:
        """
        adds the macro to the macro_list and returns the macro name
        """
        param_list: List[VariableDeclaration] = []
        for i in self.tokens:
            if isinstance(i, VariableDeclaration):
                param_list.append(i)
        if package_name:
            global_env.add_macro(package_name + "::" + self.tokens[0].
                evaluate_value(),  # type: ignore
                                 param_list,
                                 self.tokens[len(self.tokens) - 1],  #
                                     type: ignore
                                 global_env.get_type(
                                     self.tokens[len(self.tokens) -
                                         2].evaluate_value()))  # type
                                         : ignore
        else:
            global_env.add_macro(self.tokens[0].evaluate_value(),  #
                type: ignore
                                 param_list,
                                 self.tokens[len(self.tokens) - 1],  #
                                     type: ignore
                                 global_env.get_type(
                                     self.tokens[len(self.tokens) -
                                         2].evaluate_value()))  # type
                                         : ignore
```

```
        return self.tokens[0].evaluate_value()  # type: ignore


class MacroCallExp(OclExpression):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[SimpleName, OclExpression], ...] =
            tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        arg_list = []
        name_list = []
        for token in self.tokens:
            if isinstance(token, OclExpression):
                arg_list.append(token.evaluate_value(global_env,
                    local_env))  # type: ignore
            if isinstance(token, SimpleName):
                name_list.append(token.evaluate_value())
        if len(name_list) == 2:
            return global_env.call_macro(name_list[0] + "::" +
                name_list[1], arg_list)  # type: ignore
        else:
            return global_env.call_macro(name_list[0], arg_list)  #
                type: ignore


class AttributeBinding(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[SimpleName, OclExpression], ...] =
            tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Tuple[str, Any]:
        return self.tokens[0].evaluate_value(), self.tokens[1].
            evaluate_value(global_env, local_env)  # type: ignore


class ObjectLiteralExp(LiteralExp):
    def __init__(self, *tokens):
        super().__init__()
        self.tokens: Tuple[Union[Type, AttributeBinding], ...] =
            tokens

    def evaluate_value(self, global_env: GlobalEnvironment, local_env:
        LocalEnvironment) -> Any:
        type_name: str = self.tokens[0].evaluate_value()  # type:
            ignore
```

```
        _type: type = global_env.get_create_type(type_name)  # type:
            ignore

        attr_dict_tmp = {}
        for i in range(1, len(self.tokens)):
            attribute_name, attribute_value = self.tokens[i].
                evaluate_value(global_env, local_env)  # type: ignore
            attr_dict_tmp[attribute_name] = attribute_value

        attr_dict = {}
        for name in attr_dict_tmp:
            attr_dict[global_env.
                get_correct_attribute_name_by_meta_model_type_name(
                type_name, name)] = \
                 attr_dict_tmp[name]

        return _type(**attr_dict)


class TransformationDefinition(Token):
    def __init__(self, *tokens):
        super().__init__(*tokens)
        self.tokens: Tuple[Union[SimpleName, VariableDeclaration,
            LiteralExp], ...] = tokens

    def evaluate(self,
                 source_instances: List[Any],
                 attribute_name_path: List[str],
                 meta_model_types_to_sdk_types: Optional[Dict[str,
                     Union[type, Dict[str, Union[type, Dict[str, str
                     ]]]]]] = None,
                 sdk_types_to_meta_model_types: Optional[Dict[type,
                     Union[str, Dict[str, Union[type, Dict[str, str
                     ]]]]]] = None,
                 meta_model_variables: Optional[Dict[str, Any]] = None
                     ,
                 macros: Optional[Dict[str, List[MacroDecl]]] = None)
                     \
            -> Optional[Any]:
        """
        Create a new object of the given information_model_type based
            on the transformation definition using
        the given source instances

        :param source_instances: List of instances of the given
            information model type which are mapped to the source
                templates of the transformation definition
        :param attribute_name_path: The name of the attribute in the
            given information model type to check which given
                sources instance match against the required source
```

```
            templates of the transformation definition
    :param meta_model_types_to_sdk_types: List of meta model
        classes and their mapping to the corresponding
            classes in the underlying software including the
                mapping of the class attributes
    :param sdk_types_to_meta_model_types: List of classes in the
        underlying software and their mapping to the
            corresponding meta model classes including the mapping
                of the class attributes
            Note: Only different mappings to the
                meta_model_types_to_sdk_types is needed
    :param meta_model_variables: List of variables defined by the
        meta model e.g. enums
    :param macros: List of macros with are used in this
        transformation definition

    :raise KeyError: If not enough source instances are given or
        the required source instances are missing
    """
    global_env = GlobalEnvironmentMTL()
    local_env = LocalEnvironment()
    if meta_model_types_to_sdk_types is not None:
        global_env.add_meta_model_types_to_sdk_types(
            meta_model_types_to_sdk_types)

    if sdk_types_to_meta_model_types is not None:
        global_env.add_sdk_types_to_meta_model_types(
            sdk_types_to_meta_model_types)

    if meta_model_variables is not None:
        for var in meta_model_variables.items():
            global_env.add_var(var[0], None, var[1])

    # get source template list and macro list
    source_templates: Dict[VariableDeclaration, Union[None, List[
        str, LiteralExp]]] = {}
    token_length = len(self.tokens)
    object_literal_exp: ObjectLiteralExp = self.tokens[
        token_length - 1]  # type: ignore
    # add target template value to variables
    for i in range(1, token_length - 3):
        if isinstance(self.tokens[i], VariableDeclaration):
            if isinstance(self.tokens[i+1], LiteralExp):
                source_templates[self.tokens[i]] = self.tokens[i +
                    1]  # type: ignore
            else:
                source_templates[self.tokens[i]] = None  # type:
                    ignore

    # check if source instances and source templates match and add
```

```
        them to env
    if len(source_templates.keys()) > len(source_instances):
        raise KeyError("Length of source instance list must be
            greater or equal than the source template list")

    si_found = 0
    for st in source_templates:
        var_name = st.evaluate_value(global_env, local_env)  #
            type: ignore
        si_found_temp = 0
        si_list = []
        type_name = local_env.get_var_type(var_name)
        if isinstance(source_templates[st], type(None)):
            if type(type_name) == typing._GenericAlias:  # type:
                ignore
                for si in source_instances:
                    if isinstance(si, type_name.__args__):  # type
                        :ignore
                        si_found_temp += 1
                        si_list.append(si)
                        continue
            else:
                for si in source_instances:
                    if isinstance(si, type_name):
                        si_found_temp += 1
                        si_list.append(si)
                        break
        else:
            attribute_value = source_templates[st].evaluate_value(
                global_env, local_env)  # type: ignore
            si_found_temp = 0
            si_list = []
            if type(type_name) == typing._GenericAlias:  # type:
                ignore
                for si in source_instances:
                    if isinstance(si, type_name.__args__):  # type
                        :ignore
                        tmp_parent = si
                        for i in range(len(attribute_name_path) -
                            1):
                            tmp_parent = getattr(tmp_parent,
                                attribute_name_path[i])
                        if getattr(tmp_parent,
                                   attribute_name_path[len(
                                       attribute_name_path) - 1])
                                       == attribute_value:
                            si_found_temp += 1
                            si_list.append(si)
                            continue
            else:
```

```
                    for si in source_instances:
                        if isinstance(si, type_name):
                            tmp_parent = si
                            for i in range(len(attribute_name_path) -
                                1):
                                tmp_parent = getattr(tmp_parent,
                                    attribute_name_path[i])
                            if getattr(tmp_parent,
                                       attribute_name_path[len(
                                           attribute_name_path) - 1])
                                           == attribute_value:
                                si_found_temp += 1
                                si_list.append(si)
                                break
            if len(si_list) == 0:
                raise KeyError("No Instance for source template {}
                    found in source instance list".format(var_name))
            else:
                if type(type_name) == typing._GenericAlias:
                    local_env.set_var_value(var_name, si_list)
                else:
                    local_env.set_var_value(var_name, si_list[0])
                si_found += 1

        if si_found != len(source_templates):
            raise KeyError("Source instance list does not match the
                required source template list")

        # add macros to env
        if macros is not None:
            for package_name, macro_list in macros.items():
                for macro in macro_list:
                    if isinstance(macro, MacroDecl):
                        macro.evaluate_value(global_env, package_name)

        # evaluate object literal exp
        information_model = None
        if object_literal_exp is not None:
            information_model = object_literal_exp.evaluate_value(
                global_env, local_env)
        return information_model


class PackageDeclaration:
    def __init__(self, *tokens):
        self.tokens: Tuple[Union[SimpleName, TransformationDefinition,
            MacroDecl], ...] = tokens

    def get_transformation_definition(self) -> Optional[
        TransformationDefinition]:
```

```
        for token in self.tokens:
            if isinstance(token, TransformationDefinition):
                return token
        return None

    def get_macro_list(self) -> List[MacroDecl]:
        macro_list: List[MacroDecl] = []
        for token in self.tokens:
            if isinstance(token, MacroDecl):
                macro_list.append(token)
        return macro_list

    def get_needed_packages(self) -> List[str]:
        package_list: List[str] = []
        parent = self.tokens[1]
        macro_call_exp_list = self.get_macro_call_exp_list(parent)
        for macro in macro_call_exp_list:
            macro_name: List[str] = []
            for token in macro.tokens:
                if isinstance(token, SimpleName):
                    macro_name.append(token.evaluate_value())
                else:
                    break
            if len(macro_name) == 2:
                if macro_name[0] not in package_list:
                    package_list.append(macro_name[0])
        return package_list

    def get_macro_call_exp_list(self, parent: Token) -> List[
       MacroCallExp]:
        macro_call_exp_list: List[MacroCallExp] = []
        for token in parent.tokens:
            if isinstance(token, MacroCallExp):
                macro_call_exp_list.append(token)
            if isinstance(token, Token):
                macro_call_exp_list.extend(self.
                   get_macro_call_exp_list(token))
        return macro_call_exp_list
```

D Anwendungsfall 1: Firmenspezifische Informationsmodelle

D.1 ZVEI Digital Nameplate for industrial equipment (Version 1.0)

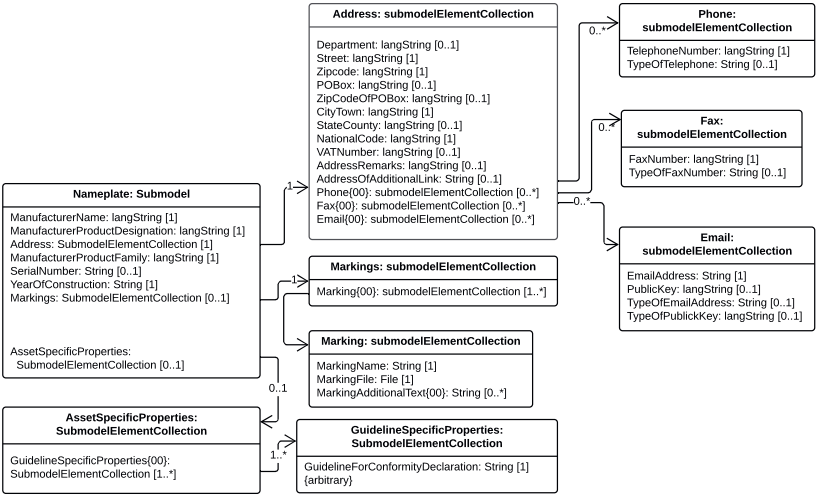

Abbildung D.1: ZVEI Digital Nameplate UML nach [14]

D.2 Digital Nameplate for Galaxie®Actuator der Firma WITTENSTEIN galaxie GmbH

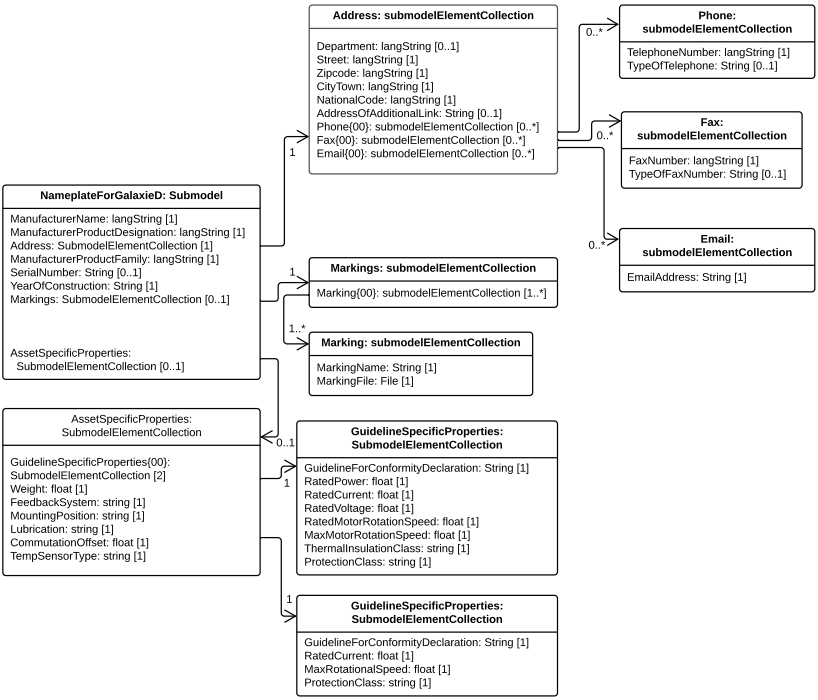

Abbildung D.2: Auszug aus dem abgeleiteten Submodel-Template Digital Nameplate for Galaxie®Actuator der Firma WITTENSTEIN galaxie GmbH nach [160]

D.3 Transformationsdefinition zwischen dem WITTENSTEIN und dem ZVEI Teilmodell-Template

```
transformationDefinition td1
  sourceTemplate:
    a: Submodel -> Reference{key: Sequence{Key{
                    idType: KeyType::IRI,
                    value: "https://wgrp.biz/sm/wgx/NameplateForGalaxieD
                       /1/0/NameplateForGalaxieD",
                    type: KeyElements::Submodel}}}
  targetTemplate:
    b: Submodel -> Reference{key: Sequence{Key{
                    idType: KeyType::IRI,
                    value: "https://admin-shell.io/zvei/nameplate/1/0/
                       Nameplate",
                    type: KeyElements::GlobalReference}}}
  value: Submodel {
    identification: a.identification,
    submodelElement: Set{
      aas_macros::copySubmodelElementByIdShort(a, "ManufacturerName"),
      aas_macros::copySubmodelElementByIdShort(a, "
         ManufacturerProductDesignation"),
      aas_macros::copySubmodelElementByIdShort(a, "
         ManufacturerProductFamily"),
      aas_macros::copySubmodelElementByIdShort(a, "SerialNumber"),
      aas_macros::copySubmodelElementByIdShort(a, "YearOfConstruction"),
      aas_macros::copySubmodelElementByIdShort(a, "Address"),
      aas_macros::copySubmodelElementByIdShort(a, "Markings"),
      let smec: Submodel = aas_macros::getSubmodelElementByIdShort(a, "
         AssetSpecificProperties") in (
        SubmodelElementCollection {
          idShort: smec.idShort,
          value: aas_macros::copySubmodelElementsBySemanticIdValue(smec,
             "https://admin-shell.io/zvei/nameplate/1/0/Nameplate/
             AssetSpecificProperties/GuidelineSpecificProperties"),
          ordered: smec.ordered,
          allowDuplicates: smec.allowDuplicates,
          semanticId: smec.semanticId,
          kind: smec.kind
        }
      )
    },
    idShort: a.idShort,
    description: a.description,
    administration: a.administration,
    semanticId: Reference{key: Sequence{Key{
                          idType: KeyType::IRI,
                          value: "https://admin-shell.io/zvei/nameplate
                             /1/0/Nameplate",
                          type: KeyElements::GlobalReference}}},
    kind: ModelingKind::Instance
  }
```

E Anwendungsfall 2: Verschiedene Versionen eines Informationsmodells

E.1 Version 1 des Teilmodell-Templates ManufacturerDocumentation basierend auf der VDI 2770 Spezifikation

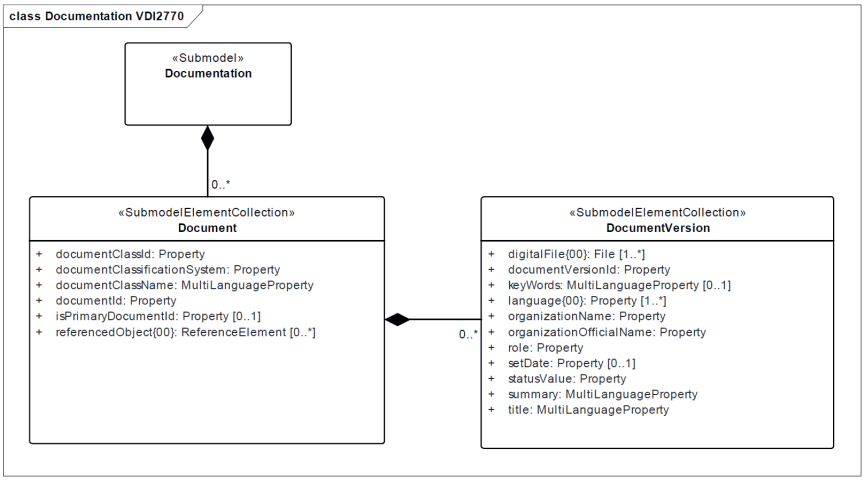

Abbildung E.1: Unveröffentlichte Version 1 des Teilmodell-Templates ManufacturerDocumentation

E.2 Version 2 des Teilmodell-Templates ManufacturerDocumentation basierend auf der VDI 2770 Spezifikation

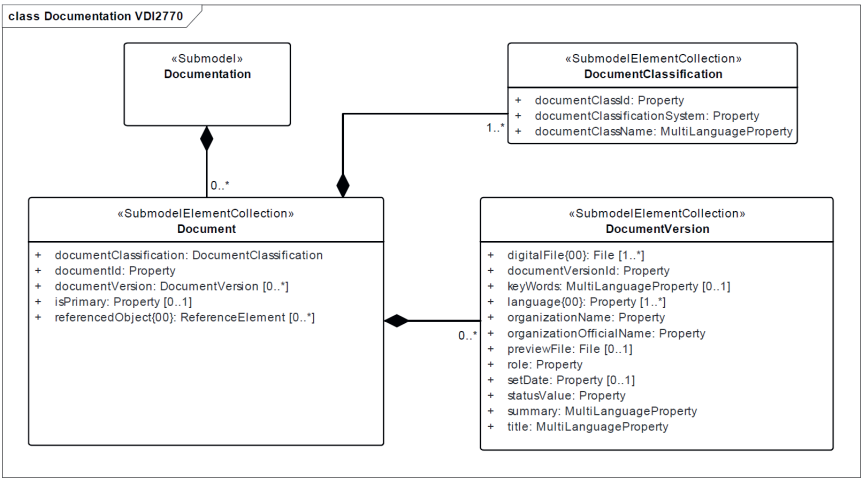

Abbildung E.2: Unveröffentlichte Version 2 des Teilmodell-Templates ManufacturerDocumentation

E.3 Version 3 des Teilmodell-Templates ManufacturerDocumentation basierend auf der VDI 2770 Spezifikation

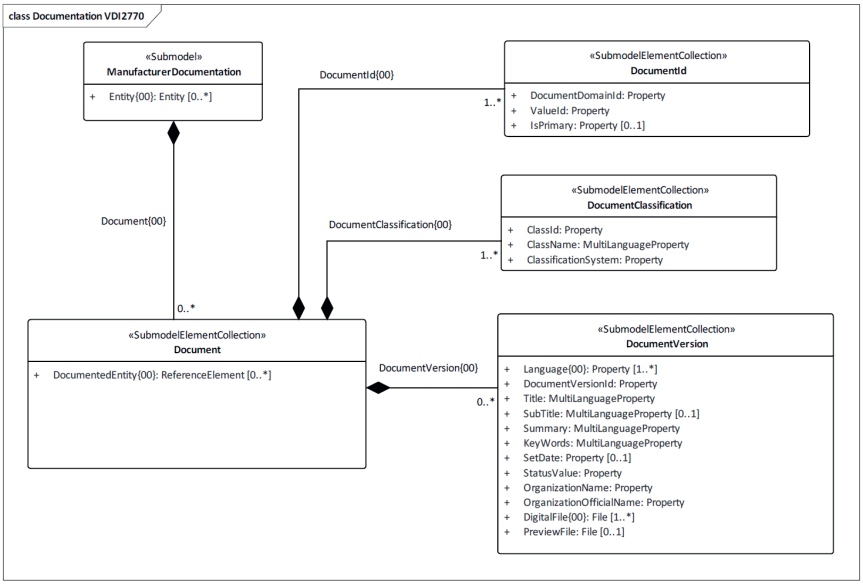

Abbildung E.3: Unveröffentlichte Version 3 des Teilmodell-Templates ManufacturerDocumentation

E.4 Transformationsdefinition zwischen den Versionen 1 und 2

```
transformationDefinition td1
  sourceTemplate:
    a: Submodel -> Reference{key: Sequence{Key{
                    idType: KeyType::IRI,
                    value: "http://admin-shell.io/vdi/2770/1/0/
                        Documentation",
                    type: KeyElements::GlobalReference}}}
  targetTemplate:
    b: Submodel -> Reference{key: Sequence{Key{
                    idType: KeyType::IRI,
                    value: "http://admin-shell.io/vdi/2770/1/1/
                        Documentation",
                    type: KeyElements::GlobalReference}}}
  value: Submodel {
    identification: a.identification,
    submodelElement:
```

```
let smc_document_list: Set(SubmodelElementCollection) =
  aas_macros::getSubmodelElementsBySemanticIdValue(
    a,
    "http://admin-shell.io/vdi/2770/1/0/Document")
in (
  smc_document_list->iterate(
    smc_document: SubmodelElementCollection;
    new_set: Set(SubmodelElement) = Set{} |
    new_set->including(
      aas_macros::copySubmodelElementCollectionWithValue(
        smc_document,
        Set{
          aas_macros::copySubmodelElementByIdShort(smc_document,
              "DocumentId"),
          aas_macros::copySubmodelElementByIdShort(smc_document,
              "IsPrimaryDocument"),
          aas_macros::copySubmodelElementsBySemanticIdValue(
             smc_document, "http://admin-shell.io/vdi
             /2770/1/0/Document/ReferencedObject"),
          SubmodelElementCollection{
            idShort: "DocumentClassification00",
            description: LangStringSet{
              langString: Set{
                langString{
                  language: "en",
                  name: "This SubmodelElementCollection holds
                     the information for a VDI2770
                     DocumentClassification entity"
                }
              }
            },
            value: Set{
              aas_macros::copySubmodelElementByIdShort(
                 smc_document, "DocumentClassId"),
              aas_macros::copySubmodelElementByIdShort(
                 smc_document, "DocumentClassName"),
              aas_macros::copySubmodelElementByIdShort(
                 smc_document, "DocumentClassificationSystem")
            },
            ordered: False,
            allowDuplicates: False,
            semanticId: Reference{key: Sequence{Key{
                          idType: KeyType::IRI,
                          value: "http://admin-shell.io/vdi
                             /2770/1/0/DocumentClassification/
                             DocumentClassification",
                          type: KeyElements::GlobalReference}}},
            kind: ModelingKind::Instance
          },
          aas_macros::copySubmodelElementSet(
```

```
                aas_macros::getSubmodelElementsBySemanticIdValue(
                  smc_document,
                  "http://admin-shell.io/vdi/2770/1/0/DocumentVersion"
                )
              )
            }
          )
        )
      )
    ),
    idShort: a.idShort,
    description: a.description,
    administration: a.administration,
    semanticId: Reference{key: Sequence{Key{
                  idType: KeyType::IRI,
                  value: "http://admin-shell.io/vdi/2770/1/1/
                      Documentation",
                  type: KeyElements::GlobalReference}}},
    kind: ModelingKind::Instance
}
```

F Anwendungsfall 3: Integration von Komponenten und zugehörigen Informationsmodellen

```
transformationDefinition td1
  sourceTemplate:
    a: Set(Submodel) -> Reference{key: Sequence{Key{
                          idType: KeyType::IRI,
                          value: "https://acplt.org/PowerMonitoring",
                          type: KeyElements::GlobalReference}}}
  targetTemplate:
    b: Submodel -> Reference{key: Sequence{Key{
                            idType: KeyType::IRI,
                            value: "https://acplt.org/PowerMonitoring
                                ",
                            type: KeyElements::GlobalReference}}}
  value: Submodel {
    identification: Identifier{
                      id: "https://acplt.org/SM/TestSM",
                      idType: IdentifierType::IRI},
    submodelElement: Set{
      Property {
        idShort: "MaxPowerConsumption",
        valueType: Integer,
        value: a->iterate(sm: Submodel; max_power: Integer = 0 |
          max_power + aas_macros::getSubmodelElementByIdShort(sm,
            "MaxPowerConsumption").value)
      },
      Property {
        idShort: "RatedVoltage",
        valueType: Integer,
        value: a->iterate(sm: Submodel; max_rated_voltage: Integer = 0
            |
          max_rated_voltage.max(
            aas_macros::getSubmodelElementByIdShort(sm,
              "RatedVoltage").value))
      },
      Property {
        idShort: "PowerTypes",
        valueType: String,
        value:
          let power_type: String = ""
          in (
            a->iterate(sm: Submodel; power_type_list: Set(String) =
                Set{} |
              let tmp_string: String = aas_macros::
                  getSubmodelElementByIdShort(sm,
                    "RatedVoltage").value.toString() + aas_macros::
                      getSubmodelElementByIdShort(sm, "PowerType").
                      value
```

```
            in (
              if power_type_list->includes(tmp_string)
              then power_type_list
              else power_type_list->including(tmp_string)
              endif
            )
          )->toString()
        )
    }
  },
  idShort: "PowerMonitoring",
  administration: AdministrativeInformation{
                    version: "1",
                    revision: "0"},
  semanticId: Reference{key: Sequence{Key{
                        idType: KeyType::IRI,
                        value: "https://acplt.org/PowerMonitoring",
                        type: KeyElements::GlobalReference}}},
  kind: ModelingKind::Instance
}
```

G Testergebnisse der Versuchreihen

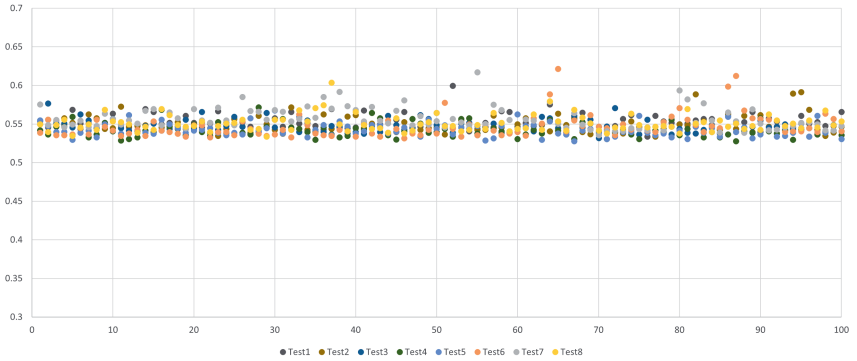

Abbildung G.1: Darstellung der Messwerte für die Erstellung des Parsers mit System 1

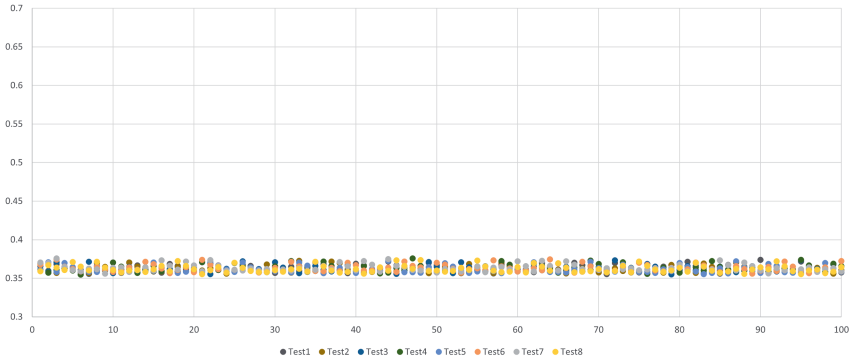

Abbildung G.2: Darstellung der Messwerte für die Erstellung des Parsers mit System 2

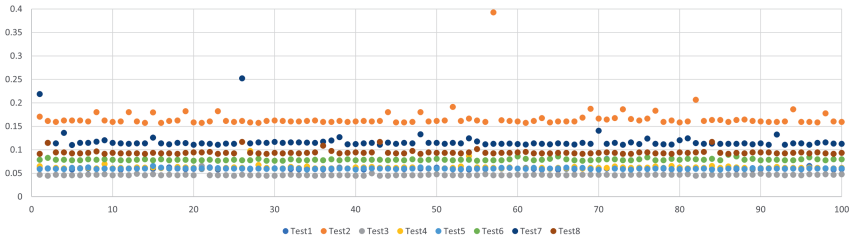

Abbildung G.3: Darstellung der Messwerte für das Parsing der Transformationdefinition-Datei mit System 1

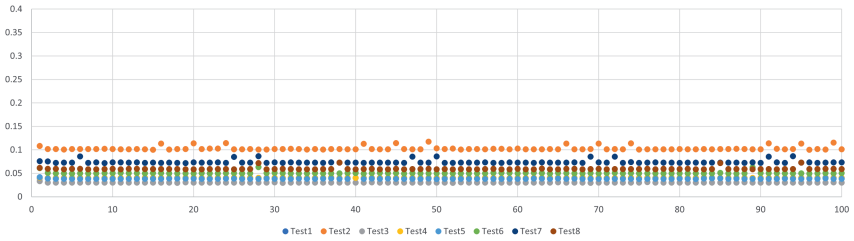

Abbildung G.4: Darstellung der Messwerte für das Parsing der Transformationdefinition-Datei mit System 2

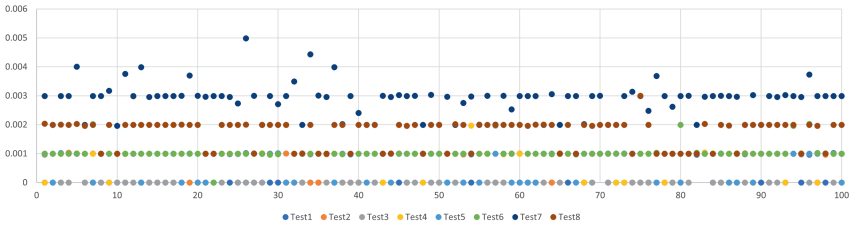

Abbildung G.5: Darstellung der Messwerte für die Ermittlung der benötigten zusätzlichen Dateien mit System 1

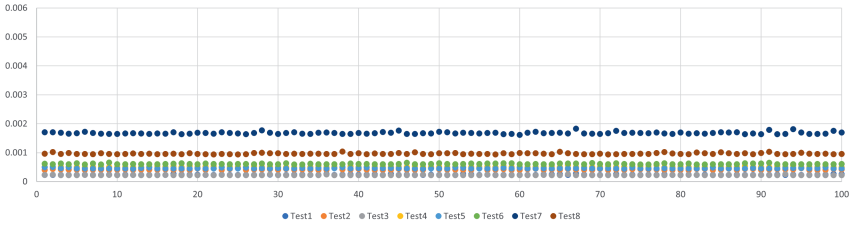

Abbildung G.6: Darstellung der Messwerte für die Ermittlung der benötigten zusätzlichen Dateien mit System 2

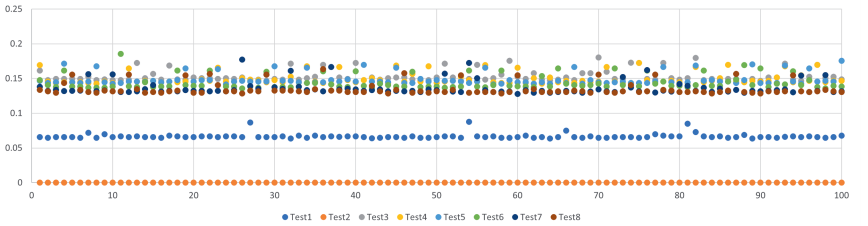

Abbildung G.7: Darstellung der Messwerte für das Parsing der zusätzlichen Dateien mit System 1

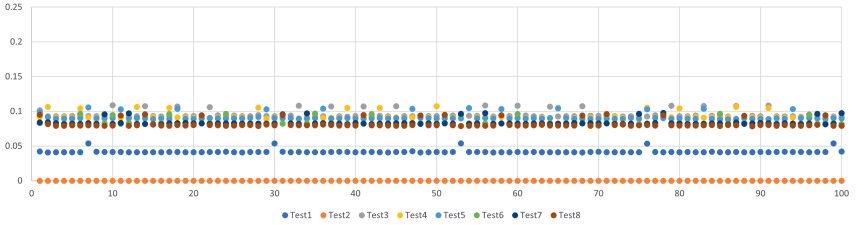

Abbildung G.8: Darstellung der Messwerte für das Parsing der zusätzlichen Dateien mit System 2

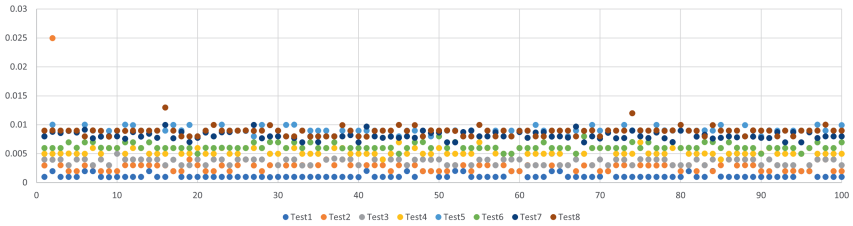

Abbildung G.9: Darstellung der Messwerte für die Anwendung des ausführbaren abstrakten Syntaxbaums mit System 1

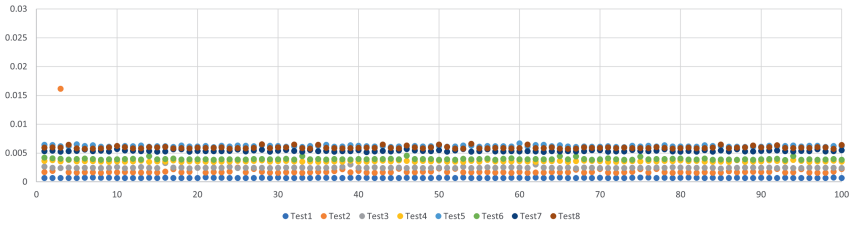

Abbildung G.10: Darstellung der Messwerte für die Anwendung des ausführbaren abstrakten Syntaxbaums mit System 2

Literaturverzeichnis

[1] B. für Wirtschaft und Energie, "Was ist so revolutionär an Industrie 4.0," https://www.bmwi.de/Redaktion/DE/FAQ/Industrie-40/faq-industrie-4-0.html?cms_artId=401710, eingesehen am xx.xx.xxxx. [Online]. Available: https://www.bmwi.de/Redaktion/DE/FAQ/Industrie-40/faq-industrie-4-0.html?cms_artId=401710

[2] VDI/VDE Industrie 4.0 Begriffe/Terms, VDI, Düsseldorf, Statusreport, 2019.

[3] T. Tauchnitz, "Die Verwaltungsschale - Lösung für das Datenchaos!" *atp magazin*, vol. 62, no. 6-7, pp. 50–59, 2020.

[4] Plattform Industrie 4.0, *Details of the Asset Administration Shell - Part 1 - The exchange of informationen between partners in the value chain of Industrie 4.0 (Version 3.0RC01)*. Bundesministerium für Wirtschaft und Energie, 2020.

[5] ——, *Details of the Asset Administration Shell - Part 2 - Interoperability at Runtime - Exchanging Information via Application Programming Interfaces (Version 1.0RC01)*. Bundesministerium für Wirtschaft und Energie, 2020.

[6] ——, *Functional View of the Asset Administration Shell in an Industrie 4.0 Systen Environment*. Bundesministerium für Wirtschaft und Energie, 2021.

[7] W. Mahnke, "Informationsmodellierung mit OPC Unified Architecture," *atp magazin*, vol. 62, no. 3, pp. 58–65, 2020.

[8] IEC, *IEC TS 62832-1 Ed.1.0 - Industrial-process measurement, control and automation - Digital factory framework - Part 1: General principles*. Beuth Verlag, Berlin, 2020.

[9] ——, *IEC TS 62832-2 Ed.1.0 - Industrial-process measurement, control and automation - Digital factory framework - Part 2: Model elements*. Beuth Verlag, Berlin, 2020.

[10] ——, *IEC TS 62832-3 Ed.1.0 - Industrial-process measurement, control and automation - Digital factory framework - Part 3: Application of Digital Factory for life cycle management of production systems*. Beuth Verlag, Berlin, 2020.

[11] S. Kaebisch, T. Kamiya, M. McCool, and V. Charpenay, "Web of Things (WoT) thing description," *First Public Working Draft, W3C*, 2017.

[12] S. Kaebisch, T. Kamiya, M. McCool, V. Charpenay, and M. Kovatsch, "Web of Things (WoT) thing description," *Recommendation, W3C*, 2020.

[13] Plattform Industrie 4.0, "Specification Submodel Templates of the Asset Administration Shell: Generic Frame for Technical Data for Industrial Equipment in Manufacturing (Version 1.1)," Bundesministerium für Wirtschaft und Energie, Berlin, Standard, 2020.

[14] ——, "Specification Submodel Templates of the Asset Administration Shell: ZVEI Digital Nameplate for industrial equipment (Version 1.1)," Bundesministerium für Wirtschaft und Energie, Berlin, Standard, 2020.

[15] ——, "Relationships between I4.0 Components – Composite Components and Smart Production: Continuation of the Development of the Reference Model for the I4.0 SG Models and Standards ," Bundesministerium für Wirtschaft und Energie, Berlin, Standard, 2017.

[16] M. Both and J. Müller, "Deep Learning in Industrie 4.0 Umgebungen als Wegbereiter für automatisierte Abbildung von Ontologien," *Tagungsband Automation*, pp. 675–686, 2020.

[17] J. Nilsson, F. Sandin, and J. Delsing, "Interoperability and machine-to-machine translation model with mappings to machine learning tasks," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, vol. 1. IEEE, 2019, pp. 284–289.

[18] J. Bakakeu, M. Brossog, J. Zeitler, J. Franke, S. Tolksdorf, H. Klos, and J. Peschke, "Automated reasoning and knowledge inference on opc ua information models," in *2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS)*. IEEE, 2019, pp. 53–60.

[19] T. Miny, M. Thies, U. Epple, S. Wein, and C. Diedrich, "Konzept für die automatisierte Erstellung von Verwaltungsschalen-Teilmodellen mit Hilfe domänenspezifischer Transformationssprachelemente," *Tagungsband Automation*, pp. 103–104, 2020.

[20] T. Deppe, L. Nothdurft, and U. Epple, "DIN SPEC 92000 als Enabler für Plug-and-Produce-Konzepte," *atp magazin*, vol. 62, no. 4, pp. 78–85, 2020.

[21] Miny, Torben and Thies, Michael and Epple, Ulrich and Diedrich, Christian, "Modeltransformation for Asset Administration Shells," in *IECON 2020*, 2020.

[22] P. Janich, *Sprache und Methode: eine Einführung in philosophische Reflexion*. UTB, 2014.

[23] S. Strahringer, "Ein sprachbasierter Metamodellbegriff und seine Verallgemeinerung durch das Konzept des Metaisierungsprinzips," in *Modellierung*, vol. 98, 1998, pp. 15–20.

[24] M. Ulrich, *Die Sprache als Sache: Primärsprache, Metasprache, Übersetzung: Untersuchungen zum Übersetzen und zur Übersetzbarkeit anhand von deutschen, englischen und vor allem romanischen Materialen*. Gunter Narr Verlag, 1997, vol. 49.

[25] J. Mittelstraß, *Enzyklopädie Philosophie und Wissenschaftstheorie: Band 2: HO*. Wissenschaftliche Buchgesellschaft, 2013.

[26] S. Strahringer, *Metamodellierung als Instrument des Methodenvergleichs: Eine Evaluierung am Beispiel objektorientierter Analysemethoden*. Shaker, 1996.

[27] G. Klaus and M. Buhr, *Philosophisches Wörterbuch*. Verlag Enzyklopädie Leipzig, 1964.

[28] M. Polke, *Prozeßleittechnik*. Oldenbourg, 1994.

[29] B. Rumpe, *Modellierung mit UML*. Springer Berlin Heidelberg, 2011.

[30] H. Stachowiak, *Allgemeine Modelltheorie*. Springer, 1973.

[31] A. Fleischmann, S. Oppl, W. Schmidt, and C. Stary, *Ganzheitliche Digitalisierung von Prozessen: Perspektivenwechsel-Design Thinking-wertegeleitete Interaktion*. Springer, 2018.

[32] O. Vogel, I. Arnold, A. Chughtai, E. Ihler, T. Kehrer, U. Mehlig, and U. Zdun, "Software-Architektur: Grundlagen–Konzepte," *Praxis*, vol. 2, 2009.

[33] A. Fleischmann, S. Oppl, W. Schmidt, and C. Stary, "Modellierungssprachen," in *Ganzheitliche Digitalisierung von Prozessen*. Springer, 2018, pp. 71–128.

[34] M. Kobler, *Qualität von Prozessmodellen: Kennzahlen zur analytischen Qualitätssicherung bei der Prozessmodellierung*. Logos Verlag Berlin GmbH, 2010.

[35] N. Chomsky, *Aspekte der Syntax-Theorie*. Suhrkamp, 1969.

[36] ISO/IEC, "CSA ISO/IEC 9899:2019-10-01: Information technology - Programming languages - C (Adopted ISO/IEC 9899:2018, fourth edition, 2018-07)," Standard, 2019.

[37] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, 1981.

[38] D. Abel, *Petri-netze für Ingenieure: Modellbildung und Analyse diskret gesteuerter Systeme*. Springer-Verlag, 2013.

[39] Object Management Group, "OMG Unified Modeling Language (OMG UML), Version 2.5.1," Object Management Group, Standard, 2017.

[40] A. Ferdjoukh, A.-E. Baert, A. Chateau, R. Coletta, and C. Nebut, "A CSP Approach for Metamodel Instantiation," in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. IEEE, 2013, pp. 1044–1051.

[41] M. Scheidgen, "Metamodelle für Sprachen mit formaler Syntaxdefinition, am Beispiel von SDL-2000," *Humboldt-Universität zu Berlin*, 2004.

[42] J. Cabot and M. Gogolla, "Object Constraint Language (OCL): a Definitive Guide," in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 2012, pp. 58–90.

[43] M. Fowler, *Domain-Specific Languages*. Pearson Education, 2010.

[44] M. Strembeck and U. Zdun, "An approach for the systematic development of domain-specific languages," *Software: Practice and Experience*, vol. 39, no. 15, pp. 1253–1292, 2009.

[45] ISO/IEC, *ISO/IEC 9075-1:2016: Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework)*. Beuth Verlag, Berlin, 2016.

[46] T. Stahl, S. Efftinge, A. Haase, and M. Völter, *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt. verlag, 2012.

[47] M. Broy and O. Spaniol, *VDI-Lexikon Informatik und Kommunikationstechnik*. Springer-Verlag, 2013.

[48] G. Kappel and M. Schrefl, *Objektorientierte Informationssysteme: Konzepte, Darstellungsmittel, Methoden*. Springer-Verlag, 2013.

[49] ISO/TS 29002-5:2009 Industrial automation systems and integration — Exchange of characteristic data — Part 5: Identification scheme, 2009.

[50] T. Berners-Lee, R. T. Fielding, and L. Masinter, "RFC 3986: Uniform Resource Identifier (URI): Generic Syntax," *Proposed Standard, January*, 2005.

[51] P. Leach, M. Mealling, and R. Salz, "RFC 4122: A universally unique identifier (UUID) URN namespace," *Proposed Standard, July*, 2005.

[52] J. Winkelmann, "Spezifikation von Visual OCL: Eine Visualisierung der Object Constraint Language," Ph.D. dissertation, Techn. Univ., Fak. IV, Elektrotechnik und Informatik, 2005.

[53] Object Management Group, *Object Constraint Language V2.4*, 2014.

[54] M. Schleipen, *Adaptivität und semantische Interoperabilität von Manufacturing Execution Systemen (MES)*. KIT Scientific Publishing, 2013, vol. 12.

[55] A. Zeid, S. Sundaram, M. Moghaddam, S. Kamarthi, and T. Marion, "Interoperability in Smart Manufacturing: Research Challenges," *Machines*, vol. 7, no. 2, p. 21, 2019.

[56] IEC, "IEC 21823-1: Internet of Things (IoT) - Interoperability for IoT Systems - Part 1: Framework," International Electrotechnical Commission, Geneva, CH, Standard, 2019.

[57] I. S. C. Committee *et al.*, "IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)," *CA: IEEE Computer Society*, vol. 169, 1990.

[58] H. Kubicek, A. Breiter, and J. Jarke, "Daten, Metadaten, Interoperabilität," *Handbuch Digitalisierung in Staat und Verwaltung*, pp. 1–13, 2019.

[59] A. Tolk, "Composable Mission Spaces and M&S Repositories - Applicability of Open Standards," in *Spring simulation interoperability workshop, Arlington (VA)*, 2004.

[60] S. Pantsar-Syväniemi, A. Purhonen, E. Ovaska, J. Kuusijärvi, and A. Evesti, "Situation-based and self-adaptive applications for the smart environment," *Journal of Ambient Intelligence and Smart Environments*, vol. 4, no. 6, pp. 491–516, 2012.

[61] NATO Standardization Office, "Allied Joint Doctrine for Communication and Information Systems Edition A Version 1," Standard, 2017.

[62] M. Noura, M. Atiquzzaman, and M. Gaedke, "Interoperability in Internet of Things: Taxonomies and Open Challenges," *Mobile Networks and Applications*, vol. 24, no. 3, pp. 796–809, 2019.

[63] ISO/IEC 19941: Information technology - Cloud computing - Interoperability and portability, International Electrotechnical Commission, Geneva, CH, Standard, 2017.

[64] ISO/IEC 7498-1: Information technology - Open System Interconnection - Basic Reference Mode: The Basic Model, International Electrotechnical Commission, Geneva, CH, Standard, 1994.

[65] IEC 21823-2: Internet of Things (IoT) - Interoperability for IoT Systems - Part 2: Transport Interoperability, International Electrotechnical Commission, Geneva, CH, Standard, 2019.

[66] IEC 21823-3: Internet of Things (IoT) - Interoperability for IoT Systems - Part 3: Semantic Interoperability, International Electrotechnical Commission, Geneva, CH, Standard, 2019.

[67] M. Noura, M. Atiquzzaman, and M. Gaedke, "Interoperability in Internet of Things Infrastructure: Classification, Challenges, and Future Work," in *International Conference on Internet of Things as a Service*. Springer, 2017, pp. 11–18.

[68] ETSI, "ETSI TR 103 535: SmartM2M; Guidelines for using semantic interoperability in the industry," 2019.

[69] ——, "ETSI TR 103 536: SmartM2M; Strategic/technical approach on how to achieve interoperability/interworking of existing standardized IoT Platforms," 2019.

[70] ——, "ETSI TR 103 537: SmartM2M; PlugtestsTM preparation on Semantic Interoperability," 2019.

[71] IEC, "White Paper Semantic interoperability:2019 - Semantic interoperability: challenges in the digital transformation age," 2019.

[72] P. Wegener, "GERMAN STANDARDIZATION ROADMAP Industrie 4.0 Version 4," *DIN e*, vol. 2020.

[73] T. Pellegrini and A. Blumauer, "Semantic Web," *Wege zur vernetzten Wissensgesellschaft. Berlin [ua] Springer*, 2006.

[74] H. van der Veer and A. Wiles, "Achieving Technical Interoperability - the ETSI Approach," *European telecommunications standards institute*, 2008.

[75] L. Christiansen, M. Hoernicke, and A. Fay, "Modellgestütztes Engineering," *atp magazin*, vol. 56, no. 03, pp. 18–27, 2014.

[76] M. Hoernicke, L. Christiansen, and A. Fay, "Anlagentopologien automatisch erstellen," *atp magazin*, vol. 56, no. 04, pp. 28–40, 2014.

[77] A. Donaubauer, A. Fichtinger, T. Kutzner, and M. Schilcher, "Semantische Modelltransformation im Kontext von INSPIRE," *Newsletter e-geo. ch*, no. 22, pp. 10–13, 2009.

[78] IEC 21823-4: Internet of Things (IoT) - Interoperability for IoT Systems - Part 4: Syntactic Interoperability, International Electrotechnical Commission, Geneva, CH, Standard, 2020.

[79] T. Mersch, U. Epple, and A. Schürr, "Regelbasierte Modelltransformation in prozessleittechnischen Laufzeitumgebungen," Fachgruppe für Materialwissenschaft und Werkstofftechnik, Tech. Rep., 2017.

[80] S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development," *IEEE software*, vol. 20, no. 5, pp. 42–45, 2003.

[81] A. G. Kleppe, J. Warmer, J. B. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

[82] H. Kern, S. Kühne, and D. Fötsch, "Merkmale und Werkzeugunterstützung für Modelltransformationen im Kontext modellgetriebener Softwareentwicklung," *Fähnrich, K.-P.; Kühne, S.; Speck, A*, 2006.

[83] V. Gruhn, D. Pieper, and C. Röttgers, *MDA®: Effektives Software-Engineering mit UML2® und Eclipse*. Springer-Verlag, 2007.

[84] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[85] T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation," *Electronic notes in theoretical computer science*, vol. 152, pp. 125–142, 2006.

[86] A. Metzger, "A Systematic Look at Model Transformations," in *Model-driven Software Development*. Springer, 2005, pp. 19–33.

[87] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, and D. Varró, "Survey and classification of model transformation tools," *Software & Systems Modeling*, vol. 18, no. 4, pp. 2361–2397, 2019.

[88] E. Visser, "A Survey of Rewriting Strategies in Program Transformation Systems," *Electronic Notes in Theoretical Computer Science*, vol. 57, no. 2, 2001.

[89] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, "Reuse in model-to-model transformation languages: are we there yet?" *Software & Systems Modeling*, vol. 14, no. 2, pp. 537–572, 2015.

[90] S. Nolte, *QVT-operational mappings: Modellierung mit der Query views Transformation*. Springer-Verlag, 2009.

[91] J.-M. Jézéquel, O. Barais, and F. Fleurey, "Model Driven Language Engineering with Kermeta," in *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 2009, pp. 201–221.

[92] D. Akehurst and S. Kent, "A Relational Approach to Defining Transformations in a Metamodel," in *International Conference on the Unified Modeling Language*. Springer, 2002, pp. 243–258.

[93] S. Nolte, *QVT-Relations Language*. Springer Science & Business Media, 2009.

[94] A. Schürr, "Specification of Graph Translators with Triple Graph Grammars," in *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 1994, pp. 151–163.

[95] A. Schürr and F. Klar, "15 Years of Triple Graph Grammars," in *International Conference on Graph Transformation*. Springer, 2008, pp. 411–425.

[96] C. Ermel, M. Rudolf, and G. Taentzer, "The AGG approach: Language and environment," in *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 2: Applications, Languages and Tools*. World Scientific, 1999, pp. 551–603.

[97] E. D. Willink, "UMLX: A graphical transformation language for MDA," in *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.

[98] T. Vogel and H. Giese, *Model-Driven Engineering of Adaptation Engines for Self-Adaptive Software: Executable Runtime Megamodels*. Universitätsverlag Potsdam, 2013.

[99] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of computer programming*, vol. 72, no. 1-2, pp. 31–39, 2008.

[100] A. Kalnins, J. Barzdins, and E. Celms, "Model transformation language MOLA," in *Model Driven Architecture*. Springer, 2004, pp. 62–76.

[101] D. Varró and A. Balogh, "The model transformation language of the VIATRA2 framework," *Science of Computer Programming*, vol. 68, no. 3, pp. 214–234, 2007.

[102] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The Epsilon Transformation Language," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2008, pp. 46–60.

[103] T. Baar and J. Whittle, "On the Usage of Concrete Syntax in Model Transformation Rules," in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 2006, pp. 84–97.

[104] R. Grønmo, *Using Concrete Syntax in Graph-based Model Transformation*, 2009.

[105] B. Rumpe and I. Weisemöller, "A Domain Specific Transformation Language," *arXiv preprint arXiv:1409.2309*, 2014.

[106] I. Weisemöller, *Generierung domänenspezifischer Transformationssprachen*. Shaker, 2012.

[107] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, "Explicit Transformation Modeling," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2009, pp. 240–255.

[108] T. Reiter, E. Kapsammer, W. Retschitzegger, W. Schwinger, and M. Stumptner, "A generator framework for domain-specific model transformation languages," in *ICEIS (3)*, 2006, pp. 27–35.

[109] J. Steel and R. Drogemuller, "Domain-Specific Model Transformation in Building Quantity Take-Off," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 198–212.

[110] K. Hölldobler, *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen*. Shaker, 2018.

[111] E. Syriani, J. Gray, and H. Vangheluwe, "Modeling a Model Transformation Language," in *Domain Engineering*. Springer, 2013, pp. 211–237.

[112] J. S. Cuadrado, E. Guerra, and J. de Lara, "Towards the Systematic Construction of Domain-Specific Transformation Languages," in *European Conference on Modelling Foundations and Applications*. Springer, 2014, pp. 196–212.

[113] A. Petter, "Modell-zu-Modell-Transformation von Modellen von Benutzerschnittstellen," Ph.D. dissertation, TU-Prints, 2012.

[114] J. I. Irazábal, C. Pons, and C. Neil, "Model transformation as a mechanism for the implementation of domain specific transformation languages," *Electronic Journal of SADIO (EJS)*, vol. 9, pp. 49–66, 2010.

[115] E. Kalnina, A. Kalnins, A. Sostaks, E. Celms, and J. Iraids, "Tree Based Domain-Specific Mapping Languages," in *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 2012, pp. 492–504.

[116] C. Wagner, J. Grothoff, U. Epple, R. Drath, S. Malakuti, S. Grüner, M. Hoffmeister, and P. Zimermann, "The Role of the Industry 4.0 Asset Administration Shell and the Digital Twin during the life cycle of a plant," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2017, pp. 1–8.

[117] M. Grieves, "Digital twin: manufacturing excellence through virtual factory replication," *White paper*, vol. 1, pp. 1–7, 2014.

[118] M. Shafto, M. Conroy, R. Doyle, E. Glaessgen, C. Kemp, J. LeMoigne, and L. Wang, "Modeling, Simulation, Information Technology & Processing Roadmap," *Technology Area*, vol. 11, 2010.

[119] F. Tao, H. Zhang, A. Liu, and A. Y. Nee, "Digital Twin in Industry: State-of-the-Art," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 4, pp. 2405–2415, 2018.

[120] K. Panetta, "Gartner's Top 10 Strategic Technology Trends for 2017," *Smarter With Gartner*, vol. 18, 2016.

[121] D. CeArley, B. Burke, S. Searle, and M. J. Walker, "Top 10 Strategic Technology Trends for 2018," *The Top*, vol. 10, 2016.

[122] G. Top, "Strategic Technology Trends for 2019," *David Cearley, Brian Burke*, 10.

[123] DIN, "DIN SPEC 91345: Reference Architecture Model Industrie 4.0 (RAMI4.0)," DIN - German Institute for Standardization, Berlin, DE, Standard, 2016.

[124] M. Jacoby and T. Usländer, "Digital Twin and Internet of Things—Current Standards Landscape," *Applied Sciences*, vol. 10, no. 18, p. 6519, 2020.

[125] ISO 13584-42: Industrial automation systems and integration - Part 42: Description methodology: Methodology for structuring parts families, International Standardization Organisation, Geneva, CH, Standard, 2010.

[126] IEC 61360-1: Standard data elements types with associated classification scheme for electric items - Part 1: Definitions - Principles and methods, International Electrotechnical Commission, Geneva, CH, Standard, 2017.

[127] ISO 10303-11: Industrial automation systems and integration – Product data representation and exchange – Part 11: Description methods: The EXPRESS language reference manual, International Standardization Organisation, Geneva, CH, Standard, 2004.

[128] IEC 61360-2: Standard data elements types with associated classification scheme for electric items - Part 2: EXPRESS, International Electrotechnical Commission, Geneva, CH, Standard, 2012.

[129] IEC 62714-1: Engineering data exchange format for use in industrial automation systems engineering - Automation Markup Language - Part 1: Architecture and general requirements, International Electrotechnical Commission, Geneva, CH, Standard, 2018.

[130] IEC 61987-10: Industrial-Process Measurement and Control - Data Structures and Elements in Process Equipment Catalogues - Part 10: Lists of Properties (LOPs) for Industrial-Process Measurement and Control for Electronic Data Exchange. Fundamentals, International Electrotechnical Commission, Geneva, CH, Standard, 2009.

[131] IEC 61804-3 (2015): Function Blocks (FB) for process control and Electronic Device Description Language (EDDL) - Part 3: EDDL syntax and semantics, International Electrotechnical Commission, Geneva, CH, Standard, 2015.

[132] IEC 62453-1: Field device tool (FDT) interface specification - Part 1: Overview and guidance, International Electrotechnical Commission, Geneva, CH, Standard, 2016.

[133] IEC 62769-1: Field device integration (FDI) - Part 1: Overview, International Electrotechnical Commission, Geneva, CH, Standard, 2015.

[134] IEC 62832-1: Industrial-process measurement, control and automation - Digital factory framework - Part 1: General principles, International Electrotechnical Commission, Geneva, CH, Standard, 2016.

[135] IEC 62541-1: OPC Unified Architecture - Part 1: Overview and Concepts, International Electrotechnical Commission, Geneva, CH, Standard, 2016.

[136] B. Boss, S. Bader, A. Orzelski, M. Hoffmeister, M. ten Hompel, B. Vogel-Heuser, and T. Bauernhansl, "Verwaltungsschale," in *Handbuch Industrie 4.0: Produktion, Automatisierung und Logistik*. Springer Berlin Heidelberg, 2019.

[137] W. Dorst, *Umsetzungsstrategie Industrie 4.0: Ergebnisbericht der Plattform Industrie 4.0*. Bitkom Research GmbH, 2015.

[138] D. SPEC, "91345: 2016-04 Referenzarchitekturmodell Industrie 4.0 (RAMI4. 0)," Tech. rep., DIN Deutsches Institut für Normung e, Standard, 2016.

[139] F. Palm and U. Epple, "openAAS-Die offene Entwicklung der Verwaltungsschale," *Tagungsband Automation*, pp. 103–104, 2017.

[140] Plattform Industrie 4.0, *Details of the Asset Administration Shell - Part 1 - The exchange of informationen between partners in the value chain of Industrie 4.0 (Version 1.0)*. Bundesministerium für Wirtschaft und Energie, 2018.

[141] IEC 63278-1: Asset administration shell for industrial applications - Part 1: Administration shell structure, International Electrotechnical Commission, Geneva, CH, Standard, 2020.

[142] T. Bray *et al.*, "RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format," *Proposed Standard, December*, 2017.

[143] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," *Recommendation, W3C*, 2008.

[144] G. Klyne, J. J. Carroll, and B. McBride, "RDF 1.1 Concepts and Abstract Syntax," *Recommendation, W3C*, 2014.

[145] M. Kovatsch, R. Matsukura, M. Lagally, T. Kawaguchi, K. Toumura, and K. Kajimoto, "Web of Things (WoT) Architecture," *Recommendation, W3C*, 2020.

[146] D. Hardt *et al.*, "RFC 6749: The OAuth 2.0 Authorization Framework," *Proposed Standard, December*, 2012.

[147] Plattform Industrie 4.0, *Verwaltungsschale in der Praxis - Wie definiere ich Teilmodelle, beispielhafte Teilmodelle und Interaktion zwischen Verwaltungsschalen (Version 1.0)*, 2019.

[148] ——, *Verwaltungsschale in der Praxis - Wie definiere ich Teilmodelle, beispielhafte Teilmodelle und Interaktion zwischen Verwaltungsschalen (Version 1.1)*, 2020.

[149] A. Belyaev and C. Diedrich, "Aktive Verwaltungsschale von I4.0-Komponenten," 07 2019.

[150] S. Wein, Y. Dassen, T. Deppe, S. Storms, and C. Brecher, "Konzept einer Autonomen Industrie 4.0-Komponente auf Basis Agenten-basierter Ansätze," 05 2020.

[151] VDI/VDE 2193 Blatt 1: Sprache für I4.0-Komponenten, VDI, Düsseldorf, Standard, 2019.

[152] C. Diedrich, J. B. JensVialkowitsch, T. Deppe, O. Schell, A. Willner, F. Vollmar, T. Schulz, F. Pethig, J. Neidig, T. Usländer *et al.*, "I4. 0-Sprache–Vokabular, Nachrichtenstruktur und semantische Interaktionsprotokolle der I4. 0-Sprache," *Plattform I4. 0, Herausgeber: Bundesministerium für Wirtschaft und Energie (BMWi) Öffentlichkeitsarbeit*, vol. 11019.

[153] VDI/VDE 2193 Blatt 2: Sprache für I4.0-Komponenten. Interaktionsprotokoll für Ausschreibungsverfahren, VDI, Düsseldorf, Standard, 2019.

[154] L. Lehto *et al.*, *Real-time content transformations in a Web service based delivery architecture for geographic information*. Helsinki University of Technology, 2007.

[155] J. S. Cuadrado, "Towards a Family of Model Transformation Languages," in *Theory and Practice of Model Transformations*. Springer, 2012, pp. 176–191.

[156] I. Kurtev, "State of the Art of QVT: A Model Transformation Language Standard," in *International Symposium on Applications of Graph Transformations with Industrial Relevance*. Springer, 2007, pp. 377–393.

[157] H. Krahn, B. Rumpe, and S. Völkel, "MontiCore: a framework for compositional development of domain specific languages," *International journal on software tools for technology transfer*, vol. 12, no. 5, pp. 353–372, 2010.

[158] N. Wirth, *Grundlagen und Techniken des Compilerbaus*. Oldenbourg Wissenschaftsverlag, 2012.

[159] F. L. DeRemer, "Practical translators for LR (k) languages," Ph.D. dissertation, Massachusetts Institute of Technology, 1969.

[160] Bernd Vojanec, "Arbeiten mit Submodel Templates," 2021. [Online]. Available: https://www.youtube.com/watch?v=aV2dA8ZY2v0

[161] M. Both, N. Maisch, and J. Müller, "Semantische Interoperabilität durch Natural Language – Processing als Basis für Self-X-Fähigkeiten von Verwaltungsschalen in semantisch heterogenen Asset-Netzwerken," *Tagungsband Automation*, pp. 571–584, 2021.

[162] P. Haase and J. Völker, "Ontology learning and reasoning—dealing with uncertainty and inconsistency," in *Uncertainty reasoning for the semantic web I*. Springer, 2006, pp. 366–384.

VDI verlag

Alle 23 Reihen der „Fortschritt-Berichte VDI“
in der Übersicht – bequem recherchieren unter:
elibrary.vdi-verlag.de

Und direkt bestellen unter:
www.vdi-nachrichten.com/shop

Reihe 01 Konstruktionstechnik / Maschinenelemente
Reihe 02 Fertigungstechnik
Reihe 03 Verfahrenstechnik
Reihe 04 Bauingenieurwesen
Reihe 05 Grund- und Werkstoffe/Kunststoffe
Reihe 06 Energietechnik
Reihe 07 Strömungstechnik
Reihe 08 Mess-, Steuerungs- und Regelungstechnik
Reihe 09 Elektronik/Mikro- und Nanotechnik
Reihe 10 Informatik/Kommunikation
Reihe 11 Schwingungstechnik
Reihe 12 Verkehrstechnik/Fahrzeugtechnik
Reihe 13 Fördertechnik/Logistik
Reihe 14 Landtechnik/Lebensmitteltechnik
Reihe 15 Umwelttechnik
Reihe 16 Technik und Wirtschaft
Reihe 17 Biotechnik/Medizintechnik
Reihe 18 Mechanik/Bruchmechanik
Reihe 19 Wärmetechnik/Kältetechnik
Reihe 20 Rechnergestützte Verfahren
Reihe 21 Elektrotechnik
Reihe 22 Mensch-Maschine-Systeme
Reihe 23 Technische Gebäudeausrüstung

Ingenieure wollen immer alles ganz genau wissen. Wie wär's mit einem E-Paper- oder Zeitungs-Abo?

Mehr Meinung. Mehr Orientierung. Mehr Wissen.
Wesentliche Informationen zu neuen Technologien und Märkten.
Das bietet VDI nachrichten, Deutschlands meinungsbildende Wochenzeitung zu Technik, Wirtschaft und Gesellschaft, den Ingenieuren. Sofort abonnieren und lesen.
Donnerstagabends als E-Paper oder freitags als Zeitung.

Jetzt abonnieren: Leser-Service VDI nachrichten, 65341 Eltville
Telefon: +49 6123 9238-201, Telefax: +49 6123 9238-244, vdi-nachrichten@vuservice.de

100 JAHRE VDI nachrichten
TECHNIK IN SZENE GESETZT.

www.vdi-nachrichten.com/abo

VDI

REIHE 10
INFORMATIK/
KOMMUNIKATION

NR. 876 **ISBN 978-3-18-387610-5**

BAND
1|1

VOLUME
1|1