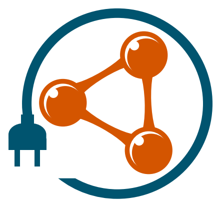

Semantic Plug & Play

Selbstbeschreibende Hardware für modulare Robotersysteme

Constantin Wanninger

Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

Fakultät für Angewandte Informatik

25. Mai 2022

Semantic Plug & Play

Erstgutachter:	Prof. Dr. Wolfgang Reif
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Tag der mündlichen Prüfung:	25. Juli 2022

„Leute, die es mit Software wirklich genau nehmen,
sollten ihre eigene Hardware bauen.“

— Alan Curtis Kay, Informatiker

Zusammenfassung

Moderne Robotersysteme bestehen aus einer Vielzahl unterschiedlicher Sensoren und Aktuatoren, aus deren Zusammenwirken verschiedene Fähigkeiten entstehen und nutzbar gemacht werden. So kann ein Knickarmroboter über die koordinierte Ansteuerung mehrerer Motoren Gegenstände greifen, oder ein Quadrocopter über Sensoren seine Lage und Position bestimmen. Eine besondere Ausprägung bilden modulare Robotersysteme, in denen sich Sensoren und Aktuatoren dynamisch entfernen, austauschen oder hinzufügen lassen, wodurch auch die verfügbaren Fähigkeiten beeinflusst werden. Die Flexibilität modularer Robotersysteme wird jedoch durch deren eingeschränkte Kompatibilität begrenzt. So existieren zahlreiche proprietäre Systeme, die zwar eine einfache Verwendung ermöglichen aber nur auf eine begrenzte Menge an modularen Elementen zurückgreifen können. Open-Source-Projekte mit einer breiten Unterstützung im Hardwarebereich, wie bspw. die Arduino-Plattform, oder Softwareprojekte, wie das Robot Operating System (ROS) versuchen, eine eben solch breite Kompatibilität zu bieten, erfordern allerdings eine sehr ausführliche Dokumentation der Hardware für die Integration.

Das zentrale Ergebnis dieser Dissertation ist ein Technologiestack (Semantic Plug & Play) für die einfache Dokumentation und Integration modularer Hardwareelemente durch Selbstbeschreibungsmechanismen. In vielen Anwendungen befindet sich die Dokumentation üblicherweise verteilt in Textdokumenten, Onlineinhalten und Quellcodedokumentationen. In Semantic Plug & Play wird ein System basierend auf den Technologien des Semantic Web vorgestellt, das nicht nur eben solch vorhandene Dokumentationen vereinheitlicht und kollektiviert, sondern das auch durch eine maschinenlesbare Aufbereitung die Dokumentation in der Prozessdefinition verwendet werden kann. Eine in dieser Dissertation entwickelte Architektur bietet für die Prozessdefinition eine API für objektorientierte Programmiersprachen, in der abstrakte Fähigkeiten verwendet werden können. Mit einem besonderen Fokus auf zur Laufzeit rekonfigurierbare Systeme können damit Fähigkeiten über Anforderungen an aktuelle Hardwarekonfigurationen ausgedrückt werden. So ist es möglich, qualitative und quantitative Eigenschaften als Voraussetzung für Fähigkeiten zu definieren, die erst bei einem Wechsel modularer Hardwareelemente erfüllt werden. Diesem Prinzip folgend werden auch kombinierte Fähigkeiten unterstützt, die andere Fähigkeiten hardwareübergreifend für ihre intrinsische Ausführung nutzen. Für die Kapselung der Selbstbeschreibung auf einzelnen Hardwareelementen werden unterschiedliche Adapter in Semantic Plug & Play unterstützt, wie etwa Mikrocontroller oder X86- und ARM-Systeme. Semantic Plug & Play ermöglicht zudem eine Erweiterbarkeit zu ROS anhand unterschiedlicher Werkzeuge, die nicht nur eine hybride Nutzung erlauben, sondern auch die Komplexität mit modellgetriebenen Ansätzen beherrschbar machen. Die Flexibilität von Semantic Plug & Play wird in sechs Experimenten anhand unterschiedlicher Hardware illustriert. Alle Experimente adressieren dabei Problemstellungen einer übergeordneten Fallstudie, für die ein heterogener Quadrocopterschwarm in hochgradig dynamischen Szenarien eingesetzt und gezielt rekonfiguriert wird.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation und Ziele . . . 2
1.2 Forschungsergebnisse . . . 7
1.3 Struktur der Arbeit . . . 10

2 Verwandte Arbeiten **13**

2.1 Semantische Hardwarebeschreibung . . . 14
2.2 Eigenschaften und Fähigkeiten . . . 15
2.3 Modellierung von Prozessen . . . 16
2.4 Simulation und digitale Abbilder . . . 18
2.5 Einsatz in realen Systemen . . . 19
2.6 Fazit und Abgrenzung zur bisherigen Forschung . . . 21

3 Fallstudie: Modulare Quadrocopterschwärme in Katastrophenszenarien **23**

3.1 Aufbau und Ablauf . . . 25
3.2 Multipotente Systeme . . . 26
3.3 Fallstudienbezogene Experimente . . . 28
3.3.1 Virtuelle Messtürme . . . 28
3.3.2 Synchrone Glasfasermessung . . . 29
3.3.3 Ausbreitungsmodellierung . . . 30
3.3.4 Der sensorbasierte Flug . . . 32
3.3.5 Pick & Place . . . 33
3.3.6 GoLive . . . 34

4 Grundlagen für die Selbstbeschreibung **37**

4.1 Semantische Beschreibung von Eigenschaften . . . 38
4.1.1 Verteilung (URI) . . . 40
4.1.2 Syntax (XML, JSON) . . . 41
4.1.3 Graphen (RDF) . . . 42
4.1.4 Schemata (RDFS) . . . 43
4.1.5 Regelwerke (Ontologien) . . . 44
4.1.6 Abfrage (SPARQL) . . . 46
4.1.7 Linked (Open) Data . . . 47
4.2 Technische Grundlagen für Fähigkeiten . . . 48
4.2.1 Experiment: Der Mobile Messturm . . . 49
4.2.2 Schnittstellen und Controller . . . 50
4.2.3 Softwarearchitekturen verteilter Komponenten . . . 53
4.2.4 Modellierung und Simulation im Robot Operating System (ROS) 56
4.2.5 Versionierung und Verteilung mit GIT . . . 58
4.2.6 Virtualisierung und Container . . . 59

5 **Konzept der Selbstbeschreibung** **63**
5.1 Die Ontologie der Selbstbeschreibung 64
5.1.1 Experiment: Ausbreitungsmodellierung 65
5.1.2 Eigenschaften und Fähigkeiten 68
5.1.3 Wissensbasis aufbauend auf Ontologien 75
5.1.4 Integration ausführbaren Codes 77
5.2 Fähigkeiten modellieren 79
5.2.1 Experiment: Synchrone Glasfasermessung 80
5.2.2 Prozesslogik 82
5.2.3 Verteilung 88
5.3 Modulare Simulation und Visualisierung 91
5.3.1 MR-Bausteine 92
5.3.2 Simulation 94
5.4 Methodologie von Semantic Plug & Play 95

6 **Architektur und Realisierung** **97**
6.1 Experiment: Pick & Place 98
6.2 Die Model-Domain-Domainmodel-Architektur 100
6.3 Domain - Objektorientierte Schnittstelle 101
6.4 Domainmodel - Mapping der Domäne 105
6.4.1 Graphensuche 106
6.4.2 Virtuelle Fähigkeiten 107
6.5 Model - Integration der Hardware 111
6.5.1 Das Self Descriptive Protocol 112
6.5.2 Die Schnittstellen-API 114
6.6 Zusammenfassung und Einordnung 115

7 **Semantic Plug & Play - Werkzeuge** **119**
7.1 SemanticSens - Erstellung einer Selbstbeschreibung 120
7.2 Integration von ROS 123
7.2.1 Experiment: Der sensorbasierte Flug 124
7.2.2 ROSSi - Modellierung von Fähigkeiten 126
7.2.3 Die Block Definition Language und das DMDE-Framework . . 132
7.3 Modulare Simulation 137
7.3.1 Experiment: GoLive 137
7.3.2 Selbstbeschreibung mittels digitalen Zwillings 139
7.3.3 Simulation 141
7.3.4 Verteilte MR-Virtualisierung und MR-Bausteine 143
7.4 Einordnung der Werkzeuge in die Methodologie 146

8 **Prototypen und Auswertung der Fallstudie** **149**
8.1 Hardwareprototypen & Trackingsysteme 150
8.1.1 Der Forschungsquadrocopter 151
8.1.2 Die Flugarena 153
8.2 Virtuelle Messtürme 155

8.3 Synchrone Glasfasermessung . . . 161
8.4 Ausbreitungsmodellierung . . . 168
8.5 Der sensorbasierte Flug . . . 172
8.6 Pick & Place . . . 178
8.7 GoLive . . . 183
8.8 Zusammenfassung der Experimente . . . 189

9 Fazit und Ausblick 193
9.1 Bewertung der erzielten Ergebnisse . . . 193
9.2 Ausblick . . . 196

Literaturverzeichnis 199

Abbildungsverzeichnis 219

Betreute Abschlussarbeiten 233

Eigene Publikationen 235

Zusammenfassung. Das Thema *selbstbeschreibende Hardware für modulare Robotersysteme* greift unterschiedliche Themengebiete auf, die in diesem Kapitel kurz zusammengefasst sind. Einleitend wird ein Überblick über die übliche Integration von Hardwareelementen gegeben und daraus werden die Forschungsfragen abgeleitet. Anschließend werden die Forschungsfragen den Forschungsergebnissen der Dissertation gegenübergestellt. Abschließend wird ein Überblick über die Struktur der Arbeit gegeben.

Einleitung

1.1 Motivation und Ziele **2**
1.2 Forschungsergebnisse **7**
1.3 Struktur der Arbeit **10**

„Wenn jedes Werkzeug auf Geheiß, oder auch vorausahnend, das ihm zukommende Werk verrichten könnte, (...) so bedarf es weder für den Werkmeister der Gehilfen noch für die Herren der Sklaven.“ – Aristoteles [280]

Robotersysteme werden in der heutigen Zeit als universelles Werkzeug betrachtet und in unterschiedlichen Szenarien eingesetzt. Dabei verwenden Roboter meist selbst Werkzeuge, wie etwa Greifer für Knickarmroboter oder Kameras bei mobilen Robotersystemen. Eine Modularisierung der eingesetzten Werkzeuge mit einfachen Wechselmechanismen steigert die Flexibilität, um das zukommende Werk verrichten zu können. Das Ziel dieser Arbeit ist eine Methodik und Architektur für die Beschreibung und Verwendung modularer Hardware. Dabei findet ein Paradigmenwechsel statt, der sich nicht an den Schnittstellen der Hardwarebausteine orientiert, sondern vielmehr an Fähigkeiten, die sich aus der Kombination von Hardwarebausteinen ergeben. Dabei wird ein besonderer Fokus auf adaptive Systeme gelegt, die zur Laufzeit rekonfiguriert werden können.

Abschnitt 1.1 motiviert zunächst die Vorteile selbstbeschreibender Hardware anhand der historischen Entwicklung, die zu der Zieldefinition dieser Arbeit führen. Die einzelnen Forschungsergebnisse der Dissertation werden in Abschnitt 1.2 vorgestellt und der Zieldefinition gegenübergestellt. Abschließend wird in Abschnitt 1.3 ein struktureller Überblick über diese Arbeit gegeben.

1.1 Motivation und Ziele

Der menschliche Alltag ist stark geprägt durch Sensoren und Aktuatoren, die in unterschiedlichsten Szenarien Verwendung finden. So bieten die Sensoren einer Smartwatch die Möglichkeiten, Fitness- oder Gesundheitsdaten zu ermitteln [274], die neben der Selbstüberwachung auch für statistische Analysen in Pay-as-you-live-Geschäftsmodellen genutzt werden [321]. Kompakte Messstationen für Gas und Feinstaub geben Aufschluss über die Luftqualität in Räumen oder im Außenbereich und werden u.a. in dem Projekt PlanetWatch für das Schürfen von Cryptowährung eingesetzt [81]. In der Hausautomatisierung finden sich intelligente Saugroboter, vernetzte Lichtschalter und Steckdosen, Temperatursensoren und Heizungssteuerungen, die über Assistenzsysteme wie Google Assist [128] mit natürlicher Sprache gesteuert oder anhand selbst erstellter Regeln, wie etwa einer Zeitschaltuhr für die Reinigung des Wohnzimmers, automatisiert werden [13]. Die Gemeinsamkeit aller Szenarien ist die Nutzung von Sensoren und Aktuatoren, um unterschiedliche Fähigkeiten anzubieten, wie die Ermittlung von Gesundheitsdaten, die Messung der Luftqualität oder die Hausautomatisierung. Auf diese Fähigkeiten können wiederum andere Fähigkeiten aufbauen, wie beispielsweise zur Auswertungen von Gesundheitsdaten für Statistiken, dem Schürfen von Cryptowährung durch die Bereitstellung der Messdaten oder sprachgesteuerten Assistenzsysteme für die Verwaltung der Sensoren einer Hausautomatisierung.

Da es jedoch eine Vielzahl an Herstellern mit unterschiedlichen physischen und logischen Schnittstellen gibt, sind die Systeme entweder nicht kombinierbar oder es werden umfangreiche Schnittstellenadapter für logische und physische Schnittstellen angeboten. So haben sich alleine in der Hausautomatisierung unterschiedliche physische Schnittstellen etabliert wie beispielsweise DECT, ZigBee, Bluetooth und WirelessLAN, die entweder direkt oder über ein zentrales Steuergerät mit einem Assistenzsystem interagieren können [246]. Der Begriff *Internet der Dinge* wird im Zusammenhang vernetzter Hardwareelemente sehr häufig verwendet, wobei nicht nur in der Literatur sehr unterschiedliche Interpretationen zu finden sind [115, 225]. Dorsemaine bildet eine Definition für den Begriff und beschreibt das Internet der Dinge übersetzt als:

> *„Gruppe von Infrastrukturen, die vernetzte Objekte miteinander verbinden und die ihre Verwaltung, Data Mining und den Zugang zu den von ihnen erzeugten Daten ermöglichen.“ [67]*

In der Hausautomatisierung kann diese Definition bereits erfolgreich angewandt werden, da die einzelnen Geräte über das Assistenzsystem mit dem Internet verbunden sind und dem Nutzer eine Verwaltung ermöglichen. Für die Verwaltung von Geräten unterschiedlicher Hersteller umfasst die Schnittstellenbibliothek von Google Assist zu Beginn des Jahres 2022 über 10.000 Hersteller mit über 50.000 Geräten [106]. Dabei sind jedoch eingeschränkte Funktionalitäten - gerade bei komplexeren Objekten wie einem Saugroboter - festzustellen. So kann ein Roborock S50/51 der Firma Xiaomi über die Schnittstelle von Google Assist gestartet, pausiert, gestoppt und an seine Ladestation geschickt werden. Über die eigene Applikation von Xiaomi hingegen können zusätzlich Kartendaten angezeigt, Sperrzonen definiert und die Nutzungszeit in Kombination mit der zu erwartenden Lebensdauer von Verschleißteilen eingesehen werden. Dieser Um-

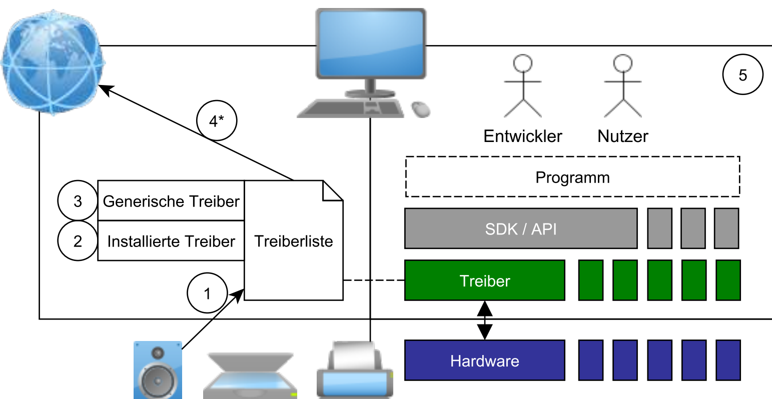

Abbildung 1.1. Übersicht über die simplifizierte Verwaltung von Treibern innerhalb eines Betriebssystems mit exemplarischem Ablauf für das Laden und die Verwendung eines Treibers innerhalb des Programms. (Eigenwerk)

stand unterliegt dem allgemeinen Prinzip der Abwägung zwischen Generalisierung und Konkretisierung, das der Mathematiker George Polya treffend auf die Spitze getrieben hat mit folgendem Zitat:

> *„Dieses Prinzip ist so vollkommen allgemein, dass keine Anwendung dafür möglich ist“ - George Polya*

Dieser Abwägung folgend bietet bei dem Beispiel des Saugroboters Google Assist die Möglichkeit, Fähigkeiten zu nutzen, die eine breite Anzahl unterschiedlicher Hersteller unterstützt. Spezielle Fähigkeiten hingegen werden in der Schnittstelle nicht beachtet. Die Integration und Wartung der Schnittstellenadapter ist bei Google Assist zentral gelöst. Systemfremde Geräte können nur über Umwege (bspw. OpenHAB [123]) integriert werden. Wird statt Google Assist ein anderer Hersteller für die Steuerung der Hausautomatisierung gewählt, so werden erneut Schnittstellenadapter benötigt. Diese stehen wiederum im Konflikt zwischen der zur Schnittstellendefinition, also der Anforderung der Anwendungsdomäne und den tatsächlich möglichen Fähigkeiten der realen Hardware. Aus diesem Konflikt leitet sich die erste Forschungsfrage der Dissertation ab:

> **Wie können Fähigkeiten einer Hardware bereitgestellt und automatisiert anhand der Domänenanforderungen systeminhärent adaptiert werden?**

Das Prinzip einer generischen Schnittstelle für Basisfähigkeiten mit möglichen Erweiterungen für spezielle Fähigkeiten findet sich hingegen in heutigen Betriebssystemen wieder. So bietet beispielsweise Linux und Windows ein breites Spektrum an generischen Treibern für die Nutzung von Hardware an mit der Möglichkeit, konkrete Treiber zu installieren [235]. Der Ablauf für das Laden eines Treibers wird in Abbildung 1.1 illustriert. Zuerst wird ein externes Hardwaregerät mit dem Rechner verbunden und über einen Identifikator in der Treiberliste des Betriebssystems nach einem passendem

Treiber gesucht (1). Priorisiert werden üblicherweise installierte, konkrete Treiber (2) und dann erst generische Treiber (3). Falls weder konkrete noch passende Treiber gefunden werden, bieten moderne Betriebssysteme eine automatisierte Suche im Internet an. Alternativ muss der Nutzer einen passenden Treiber installieren (4*). Letztendlich verwenden Programme die Treiber, um mit der angeschlossenen Hardware zu interagieren (5). Für Entwickler stehen unterschiedliche Möglichkeiten für die Interaktion mit Treibern zur Verfügung. Generische Treiber können oftmals über Betriebssystem interne Software Development Kits (SDK) angesprochen werden, wie beispielsweise eine generische Kamera über DirectShow des DirectX SDKs [43]. Konkrete Treiber haben häufig eigene SDKs oder Application Programming Interfaces (API), wie beispielsweise die Microsoft Kinect, ein Tiefenbildkamerasystem zur Erkennung von Gesten mit zugehörigem Kinect SDK. Der Entwickler arbeitet hier allerdings nicht direkt mit den Sensordaten, sondern mit dem Bodytracking, dass das SDK der Kinect aus den Sensordaten generiert. Eine Integration eigener Kameras hingegen ist im Kinect SDK nicht vorgesehen, was zu folgender Forschungsfrage führt:

> **Wie können unterschiedliche Fähigkeiten hardwareübergreifend interagieren?**

Das Kinect SDK ist nur auf dem Betriebssystem Windows und für die Programmiersprache C# verfügbar. Dies kann insbesondere dann zu Komplikationen führen, wenn ein Programm unterschiedliche Hardware verwendet und dadurch mehrere Programmiersprachen oder Betriebssysteme einbezogen werden müssen. So musste beispielsweise in dem Projekt RoKi[1] eine gestenbasierte Robotersteuerung mit einer Kinect (Kinect SDK - C#), zwei Kuka Leichtbaurobotern (Robotics API - Java) und zwei Schunk Greifern (Schunk API - C) aufwändig zusammengeführt werden. Bedingt durch das Kinect SDK ist die entstandene Anwendung nur in dem Betriebssystem Windows nutzbar. Aus dieser Gegebenheit folgt die dritte Forschungsfrage der zugrundeliegenden Dissertation:

> **Wie können Fähigkeiten plattformunabhängig genutzt werden?**

Neben der Abhängigkeiten der SDKs ist auch deren Verwendung mitunter sehr aufwändig, da die Dokumentation der Fähigkeiten stark variiert. Unter dem Begriff *Dokumentation* in der Domäne von Hardware und Treibern werden folgend die Quellcode- und Hardwaredokumentation detaillierter betrachtet:

Eine **Quellcodedokumentation** ist die Beschreibung des Quellcodes in natürlicher Sprache. Werkzeuge wie Javadoc oder Doxygen unterstützen Richtlinien und können Übersichten generieren. Zwischen Programmierern führt der Umfang oder ob überhaupt dokumentiert werden muss zu Streitigkeiten [136]. Cory House (Bekannter Entwickler) äußert sich mit folgendem übersetzten Zitat:

> *„Code ist wie Humor. Wenn man ihn erklären muss, ist er schlecht.“*
> *- Cory House*

Demgegenüber steht exemplarisch Damian Conway (Professor und berühmter Perl Entwickler) mit dem übersetzten Zitat:

[1] Ein Praxismodul des Autors im Bachelor: https://www.youtube.com/watch?v=kECNyr7v0kM

> *„Dokumentation ist wie ein Liebesbrief, den du an dein zukünftiges Selbst schreibst.“ - Damian Conway*

Damit sind zwar primär Programme adressiert, die von Nutzern verwendet und von einem kleinen Entwicklerkreis programmiert und gewartet werden. Das Problem selbst verschärft sich jedoch, wenn Projekte einem breiten Entwicklerkreis in Form eines SDKs oder einer API bereitgestellt werden. Ferner sind die Aussagen nicht zwanghaft konträr zu betrachten, da sich eine gute Einfindung in fremden Code an mehreren Merkmalen festsetzt, wie die Einhaltung von Designprinzipien oder Testverfahren, die auch exemplarisch Methodologien erklären können. Im Kontext von Treiberanbindungen werden vor allem Schnittstellendokumente verwendet, falls keine konkrete API oder SDK zur Verfügung steht. Fähigkeiten hingegen bieten ebenfalls eine Schnittstelle, die es zu beschreiben gilt, woraus sich folgende Forschungsfrage ergibt:

> **Wie können Fähigkeiten ohne SDK oder API beschrieben und für die Implementierung bereitgestellt werden?**

Die **Hardwaredokumentation** beschreibt hingegen Eigenschaften der Hardware selbst, wie etwa das Gewicht, physische Schnittstellen und Reaktionszeiten. Die Eigenschaften der Hardware sind dabei nicht zwangsweise mit Fähigkeiten assoziiert, wie beispielsweise die geometrische Beschaffenheit eines Temperatursensors. Diese Form der Dokumentation findet sich üblicherweise in Freitextdokumenten, wie beispielsweise PDF-Dokumenten, wieder. Webshops hingegen kategorisieren diese Art der Informationen, um Kaufentscheidungen durch gezielte Suchen anhand von Filteroptionen zu ermöglichen, wie beispielsweise Reichelt[2], Alternate[3], Mouser[4]. Die aus den Datenblättern extrahierten Eigenschaften der Webshops bieten allerdings analog zu den anfangs erwähnten Schnittstellenadaptern nur eine begrenzte Auswahl an. So ist es beispielsweise aktuell nicht möglich, bei den umfangreichen Filtern von Reichelt die Genauigkeit eines Temperatursensors auszuwählen. Für domänenspezifische Anwendungen kann dieser Wert aber essentiell sein. Gerade in der Entwicklung ist es nicht zuträglich, diese Art der Dokumentation in separierten Dokumenten zu suchen. Integriert man beispielsweise einen Temperatursensor über eine API, muss die Semantik des Messwertes, beispielsweise ob die Einheit in Celsius oder Fahrenheit gemessen wird, in der Hardwaredokumentation nachgeschlagen werden, sofern die Quellcodedokumentation darüber keinen Aufschluss gibt. Besonders herausfordernd wird es, wenn keine Dokumentation vorhanden ist. Dieser umständlichen Methodik sind bereits große Projekte zum Opfer gefallen, wie beispielsweise der Absturz der 125 Millionen Dollar teuren Marssonde Climate Orbiter aufgrund eines Einheitenfehlers zwischen dem metrischen und dem imperialen System. Die zugelieferte Hardware der Firma Lockheed hat die wirkende Kraft in *pound* ausgegeben [27]. Dieser Umstand folgt zur nächsten Forschungsfrage:

> **Wie können Eigenschaften maschinenlesbar abgebildet und mit den entsprechenden Fähigkeiten der Hardware verknüpft werden?**

[2] www.reichelt.de (Stand: 17.05.22)
[3] www.alternate.de (Stand: 17.05.22)
[4] www.mouser.de (Stand: 17.05.22)

Abschließend stellt sich die Frage nach der Bereitstellung beider Dokumentationsvarianten, sowie der Fähigkeiten in Form eines Treibers. Die gängigste Variante ist die Bereitstellung aktueller Treiber im Internet oder auf mitgelieferten Datenträgern. Alternativ hat der Hersteller Barco[5] für seine Clickshare Streaming Geräte einen effektiven Trick angewandt: Wenn ein Gerät angeschlossen wird, das beispielsweise für das Streamen von Medieninhalten vorgesehen ist, so wird im Betriebssystem ein generisches Speichermedium erkannt. Auf diesem Speichermedium befindet sich der konkrete Treiber sowie eine Dokumentation des Gerätes, die der Nutzer direkt installieren kann. Zwar handelt es sich bei den Geräten um Endgeräte ohne Entwicklerschnittstelle und die Dokumentation umfasst die Inbetriebnahme, aber die Verteilung der relevanten Informationen und Treiber auf das Gerät selbst ist bereits eine Art Selbstbeschreibung. Im generischen Kontext ergibt sich daraus die Forschungsfrage:

> **Kann sich ein Gerät anhand seiner Eigenschaften und Fähigkeiten selbst beschreiben?**

Während sich die Technologien zur Vernetzung unserer Welt rasant entwickeln, smarte Geräte über den 5G Standard miteinander kommunizieren [315] und der Haushalt über Sprachbefehlen gesteuert werden kann, müssen auch die Paradigmen für die Integration, Dokumentation und Interaktion unterschiedlicher Hardware neu definiert werden. Die Integration erfolgt über Freitexte innerhalb der Quelltextdokumentation, externe Dokumente werden in PDF Dateien festgehalten und die Schnittstellenbeschreibungen für die Interaktion sind ebenfalls in einer nicht maschinenlesbaren Form dokumentiert. Diese Problemstellung und ein Lösungsansatz findet sich in der Entwicklung des World Wide Webs (WWW) wieder. Als Austauschwerkzeug für Wissenschaftler in Cern [19] konnte sich durch eine breite Verfügbarkeit und sinkende Kosten der Endgeräte das WWW zu einer Plattform des Alltags entwickeln. Das Problem der fehlenden Maschinenlesbarkeit der über das WWW publizierten Inhalte hat Tim Berners Lee bereits 2001 mit der Vorstellung des Semantic Webs adressiert [24]. Eine analoge Entwicklung ist bei Sensoren und Aktuatoren, bzw. smarten Geräten zu verzeichnen. Sinkende Kosten und eine breite Verfügbarkeit führen zu einer Vielzahl an Herstellern, die ihre Inhalte in Form von Treibern und Dokumentationen bereitstellen.

Die vorliegende Dissertation vollzieht diesen Schritt, Eigenschaften und Fähigkeiten einer Hardwarekomponente zu abstrahieren, semantisch zu annotieren und über Selbstbeschreibungsmechanismen dem Entwickler und Nutzer zur Verfügung zu stellen. Verschiedene Werkzeuge und APIs verknüpfen die Selbstbeschreibung mit der Ausführungslogik, wobei ein Fokus auf Anpassungen und Erweiterungen gelegt wird. So wird der Standpunkt vertreten, dass unterschiedliche Domänen auch unterschiedliche Anforderungen besitzen und eine gemeingültige Selbstbeschreibung nicht jeder Anforderung gerecht wird. Das Prinzip der Anforderungen aus einer Domäne und der Abgleich mit den Eigenschaften und Fähigkeiten konkreter Hardware wird in einer Architektur vorgestellt. Diese unterstützt nicht nur gängige Betriebssysteme, sondern auch Mikrocontroller für die Adaptierung und Selbstbeschreibung der Hardware. Verschiedene Werkzeuge dienen der Erstellung und Anpassung der Selbstbeschreibung

[5] www.barco.com (Stand: 17.05.22)

oder vereinfachen die Integration von Erweiterungen, wie bpsw. das Robot Operating System (ROS) für Roboteranwendungen. So umfasst der Begriff **Semantic Plug & Play** nicht nur die zugrundeliegende Architektur, sondern auch alle Werkzeuge, die durch eine Selbstbeschreibung komplexe Anwendungen vereinfachen. Im folgenden Kapitel werden die erzielten Forschungsergebnisse im Rahmen der Dissertation vorgestellt und in Semantic Plug & Play verortet.

1.2 Forschungsergebnisse

Semantic Plug & Play ist im Rahmen der vorliegenden Dissertation entstanden und bietet neben einer Grundarchitektur für die Verwendung der Selbstbeschreibung auch zahlreiche Werkzeuge für die Integration, Erweiterung und Anpassung an. Motiviert wurde die Entwicklung von Semantic Plug & Play durch ein Feldexperiment in der Meteorologie [317], in welchem im Stundentakt drei Quadrocopter mit externen Temperatursensoren aufgestiegen sind und vertikale Messungen durchgeführt haben. Mühselige Fehlerdiagnosen, schwerfällige Reparaturen und eine zeitaufwändige Auswertung der Messdaten führten zu den ersten Ideen einer modularen, semantisch annotierten Lösung. Im Kontrast zu den Feldexperimenten mit realer Hardware basierten vor allem im Selbstorganisationsbereich die Versuche auf Simulationsumgebungen. Multiagentensysteme stützen sich dabei auf Fähigkeiten (auch Skills oder Capabilities) simulierter Hardware, die nach Bedarf zur Laufzeit umgerüstet werden können [148, 151, 154, 155]. Diese Lücke zwischen Abstraktion der Simulation und den Herausforderungen, die reale Hardware mit sich bringt, adressiert Semantic Plug & Play.

Die formulierten Problemstellungen berühren zahlreiche Disziplinen in den Bereichen der Softwaretechnik, semantischen Beschreibungsmechanismen, Abstraktion von Prozessen, der modellgetriebenen Entwicklung, Simulationsumgebungen und digitalen Zwillingen.

Die **Maschinenlesbare Speicherung der Eigenschaften** basiert in Semantic Plug and Play auf dem Technologiestack des Semantic Web [23]. Dafür wurde im Rahmen der Dissertation eine eigene Ontologie mit dem Namen Semantic Hardware Web entworfen, mit der Eigenschaften von Hardware beschrieben werden können. Analog zu Ontologien der Linked Open Data Community hat auch diese Ontologie verschiedene Entwicklungsstadien durchwandert. Der erste Prototyp verwendet bereits die Fähigkeiten und Eigenschaften eines Flugroboters, der mit selbst entwickelten, modularen Sensoren einen Gradientenflug absolviert [84, 309]. Darin wird illustriert, dass die Distanzsensoren trotz identischem Messwert, also der Distanz über Boden, anhand ihrer Eigenschaften eine Auswirkung auf den Prozess des Gradientenflugs haben. Die Geschwindigkeit des Quadrocopters, mit der die Teststrecke abgeflogen wird, musste entsprechend der Parameter Latenz und Genauigkeit angepasst werden. In einem zweiten Versuch mit unterschiedlichen Temperatursensoren konnte ebenfalls der Gradientenflug verwendet und durch die Eigenschaften der Selbstbeschreibung parametrisiert werden. Eine aktuelle Version des Semantic Hardware Webs findet sich in der Publikation [311] basierend auf einem Smart Home System mit über zehn Sensoren und Aktuatoren, sowie einem Recommendersystem. Eine einfache webbasierte Schnittstelle für die Instanziie-

rung der Semantic Hardware Web Ontologie bietet das Projekt SemanticSens[6], welches ebenfalls im Rahmen der Dissertation entstanden ist.

Die **Selbstbeschreibung anhand von Fähigkeiten und Eigenschaften** umfasst neben der Bereitstellung auch die Abstraktion von Fähigkeiten. Die softwaretechnische Realisierung von Semantic Plug & Play basiert in den Publikation [84, 309] auf der Robotics API [8]. Eine alleinstehende Abstraktion von Fähigkeiten und die grundlegende Architektur wird in einer kooperativen Publikation [153] vorgestellt, in der ebenfalls das Konzept für die Integration von Roboterschwärmen thematisiert ist. Die Anbindung mit dem Robot Operating System (ROS) [236], mit dem Fokus keine Abhängigkeiten aufzubauen und rekonfigurierbare Hardware zur Laufzeit zu unterstützen, wird in der Publikation [268] anhand einer Domain Specific Language (DSL) vorgestellt. Eine grafische Modellierung für die Fähigkeiten in ROS findet sich hingegen in dem veröffentlichten Projekt ROSSi [312], einer grafischen Modellierung für Semantic Plug & Play und ROS, die ebenfalls im Rahmen der Dissertation entwickelt wurde. Die Architektur mit der Integration von Domänenanforderungen ist in der Publikation [311] veröffentlicht. Als anschauliches Beispiel für das Zusammenspiel mit ROS und Semantic Plug & Play wird in der vorliegenden Dissertation ein Experiment eines Pick & Place-Quadrocopter durchgeführt.

In allen Publikationen mit Experimenten an realer Hardware [84, 269, 309, 311] werden die Eigenschaften im persistenten Speicher der Hardware abgelegt und zur Laufzeit an eine überliegende API übertragen. Bei Mikrocontrollern mit begrenztem Speicher (bspw. im Electrically Erasable Programmable Read-Only Memory - EEPROM) werden dabei lediglich HTTP-URIs verwendet, die größere Graphen verlinken. Durch die Referenz auf GIT-Repositories können auch Codefragmente zur Laufzeit heruntergeladen und in den Prozessstrukturen verwendet werden.

Eine **Übersichtliche Dokumentation der Fähigkeiten** findet sich in der Implementierung der Object Oriented Design API (OOD-API) wieder, die in der Dissertation entwickelt wurde. Die OOD-API bildet die Schnittstelle für den Entwickler und bietet eine einfache Klassenstruktur, in der Geräte, Fähigkeiten und Eigenschaften aus den einzelnen Selbstbeschreibungen bereitgestellt werden. Ein Großteil der verwendeten Sensoren kommunizieren über Schnittstellen wie UART, SPI oder I2C und werden über einen Mikrocontroller angesprochen. Die OOD-API kapselt die Fähigkeiten der Sensoren in Objekte und assoziiert die Eigenschaften mit ihnen. Zur Laufzeit kann über die OOD-API ausgegeben werden, welche Fähigkeiten zur Verfügung stehen, auf welchem Gerät sie sich befinden und welche Eigenschaften das Gerät und die Fähigkeiten haben. Damit ist die Dokumentation nicht mehr Bestandteil der API, sondern kommt als Selbstbeschreibung direkt von der verwendeten Hardware. In den Publikationen [84, 309] ist bereits die objektorientierte Programmiersprache JAVA verwendet und die Fähigkeiten gekapselt worden, die Architektur mit separierter OOD-API ist hingegen in der Publikation [311] veröffentlicht.

[6] www.semanticsens.de

Die **Plattformabhängigkeit von Fähigkeiten** wird unterteilt in zwei Sachverhalte. Im ersten Sachverhalt (**Integration**) ist der Entwickler bereits im Besitz geeigneter Hardware und will diese mittels der OOD-API nutzen. Sofern die angeschlossene Hardware noch nicht in Semantic Plug and Play integriert ist, können die Fähigkeiten anhand eines Commands Protocols implementiert werden, das ebenfalls im Rahmen der Dissertation entstanden ist. Das Commands Protocol basiert auf Serialisierungen und unterstützt durch entsprechende APIs Implementierungen in C, C++, Java, Python und kann auf beliebige Programmiersprachen erweitert werden. Die APIs und die Struktur sind in der Publikation der Architektur [311] vorgestellt. Für spezielle Anwendungen in der Robotikdomäne stehen für das Commands Protocol auch Schnittstellen für ROS zur Verfügung, die in den Publikationen [268, 312] im Fokus stehen. Für die Verteilung der Softwareartefakte wird in der vorliegenden Dissertation eine Schnittstelle zu Dockercontainern vorgestellt.

Der zweite Sachverhalt (**Planung**) umfasst die schrittweise Realisierung der gewünschten Applikation ohne vorhandene Hardware. Die Anforderungen sind in diesem Sachverhalt noch nicht oder nur wage definiert. Semantic Plug & Play sieht in diesem Szenario simulierte Fähigkeiten und Eigenschaften vor. So können Fähigkeiten einfach in einer Simulation ausgeführt und Eigenschaften der simulierten Hardware genutzt werden. Als Beispiel dienen die Ergebnisse aus dem Experiment des *sensorbasierten Flugs* [309]. Zur Erinnerung, ein Quadrocopter kann mit unterschiedlichen Distanzsensoren bestückt werden und hält über eine definierte Strecke die Distanz zum Boden. Etwaige Hindernisse auf dem Boden beeinflussen die Höhe während des Fluges. Durch die Simulation des Quadrocopters und der Sensoren kann die maximale Geschwindigkeit des Quadrocopters in Abhängigkeit der Reaktionszeit und Latenz des Sensors ermittelt werden. Das Projekt der modularen Simulation fokussiert die schrittweise Realisierung und untersucht Abhängigkeiten zwischen realen und virtuellen Sensoren. So kann beispielsweise der reale Quadrocopter einen virtuellen Sensor tragen, aber nicht vice versa. Eine weitere schrittweise Überführung ist die Verwendung realer Hardware für die simulierten Objekte. So dient ein Einplatinencomputer als erste Iteration für eine verteilte Selbstbeschreibung, dessen Position im Raum haptisch veränderbar ist. Die Positionsänderung wirkt sich ebenfalls auf die simulierten Objekte in der Simulation aus, indem Mixed Reality Mechanismen verwendet werden. Die modulare Simulation wird in einem Experiment mit dem Namen *GoLive* in Abschnitt 3.3.6 vorgestellt.

Kombinierte Fähigkeiten finden sich in vielen SDKs oder APIs implizit wieder. So stellt Microsoft in der Kinect SDK ein Skelett des Menschen zur Verfügung, das über die Sensoren aufgenommen wird. Auch im Beispiel des *sensorbasierten Flugs* [309] verwendet der Gradientenflug die Fähigkeiten *Fliegen* und *Messen*. Semantic Plug & Play bietet die Möglichkeit, durch die Einbeziehung der Hardwareeigenschaften generische Fähigkeiten zu erstellen. So kann bei der Fähigkeit *Sensorbasierter Flug* ein Distanzsensor mit einer minimalen Frequenz und Latenz als Input definiert werden, ohne konkrete Hardware zu adressieren. Der Begriff Multipotente Systeme [153] umfasst dabei ein System aus rekonfigurierbaren Hardwaresystemen, die je nach Konfiguration ein bestimmtes Set an aufeinander aufbauenden Fähigkeiten bereitstellen. Die in der Publikation beschriebene Fallstudie umfasst einen modularen Quadrocopterschwarm,

der für spezielle Szenarien konfiguriert, bzw. innerhalb eines Szenarios rekonfiguriert werden kann. Die Anforderungen an die Fähigkeiten der Hardware kommen in dieser Fallstudie jedoch nicht von einem Nutzer, sondern von einem Multiagentensystem, das bereits in mehreren Publikationen [152, 154, 155] vorgestellt wurde. Diese Fallstudie wird in der vorliegenden Dissertation detailliert eingeführt und in einzelne Experimente unterteilt.

Die **Anforderungen der Domäne** an die Granularität der Eigenschaften und Fähigkeiten unterscheiden sich maßgeblich, sodass ein generischer Ansatz ausgeschlossen ist. Hochwertige Temperatursensoren geben beispielsweise die Antwortzeit in der Zeitkonstante *tau* an, also durch eine Differenzialgleichung. Dieser Wert ist in der meteorologischen in situ Messung der Temperatur [317] elementar, um die Messwerte zu interpretieren, in anderen Domänen hingegen vernachlässigbar, wie bspw. der Hausautomatisierung [311]. Anhand der unterschiedlichen Experimente [84, 153, 309, 311] werden unterschiedliche Anforderungen definiert und in Semantic Plug & Play angewendet. Die im Rahmen der Dissertation entwickelte Model-Domain-Domainmodel (MDDM)-Architektur [311] trennt die Anforderungen der Domäne und die erweiterbare Wissensbasis und führt eine Zwischenschicht ein, um dieser Diskrepanz mit Techniken des Semantic Webs zu begegnen.

Neben den zehn bereits verorteten Publikationen sind durch die Entwicklung an Semantic Plug & Play noch sechs weitere Publikationen entstanden. Unter anderem eine Safetysteuerung für Roboterschwärme [269] oder die Grundlagen für eine am Bauteil orientierte Inspektion mit Quadrocoptern[7] [192, 266, 310], die in der vorliegenden Dissertation jedoch nicht näher ausgeführt werden. Die Fallstudie und ein Großteil der Publikationen ist im von der DFG geförderten Projekt COMBO zu verorten, in welchem selbstorganisierte Quadrocopterschwärme für Katastrophenszenarien konzipiert werden. In Zusammenarbeit mit der Feuerwehr sind dabei nicht nur die wissenschaftlichen Fragestellungen aufgegriffen worden. So sind unterschiedliche Prototypen aus glasfaser- & carbonfaserverstärkten Kunststoffen, 3D Drucken mit PLA, PETG und Harz entstanden, bis hin zum Design eigener Platinenlayouts.

1.3 Struktur der Arbeit

Die vorliegende Dissertation beginnt mit einem Überblick verwandter Arbeiten in Kapitel 2 in den Bereichen Selbstbeschreibung und modulare Hardware. Als Grundlage für die Beschreibung von Hardwareeigenschaften werden Mechanismen aus den Domänen der Industrie, Geographie und Robotik vorgestellt, die Abstraktion von Fähigkeiten untersucht und Standards für die Modellierung von Prozessen vorgestellt. Die Dokumentation, Plattformabhängigkeit und Kombinationen aus Fähigkeiten hingegen werden an simulierten und realen Anwendungen untersucht. Die Domänenproblematik wird im Fazit der verwandten Arbeiten anhand einer Gegenüberstellung der vorgestellten Arbeiten adressiert.

[7] ATV Beitrag ab Minute 19: https://www.augsburg.tv/mediathek/video/vorsprung-schwaben-roboterforschung-ladestellen-fuer-e-autos-flugtaxi/

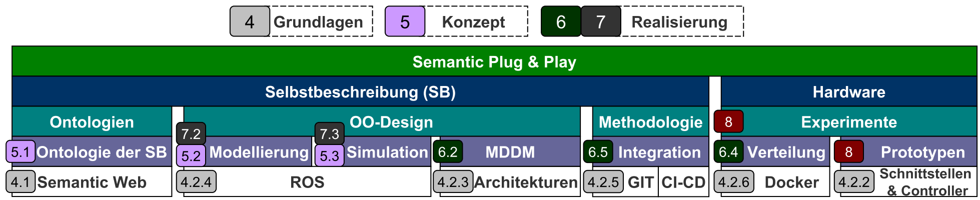

Abbildung 1.2. Übersicht der Säulen von Semantic Plug & Play und der Unterteilung der Themenbereiche, sowie die Zuordnung in die Kapitel der Dissertation. (Eigenwerk)

Im Anschluss wird in Kapitel 3 die Fallstudie aus dem Projekt COMBO vorgestellt, Multiagentensysteme eingeführt sowie die Unterteilung des Fallbeispiels in die unterschiedlichen Experimente dargelegt. Die Experimente dienen einerseits dazu, die unterschiedlichen Aspekte der Fallstudie zu beleuchten, und andererseits als leitende Beispiele für die folgenden Kapitel. So wird am Anfang eines neuen Kapitels jeweils ein Experiment detaillierter vorgestellt, um die Beispiele darauf aufzubauen.

Für das Verständnis der realisierten Selbstbeschreibung in Semantic Plug & Play werden in Kapitel 4 relevante Aspekte aus dem Semantic Web eingeführt, wie in Abbildung 1.2 illustriert. Für Fähigkeiten kommen hingegen Hardwareaspekte und Architekturgrundlagen aus dem Bereich der Softwaretechnik hinzu. Für die Realisierung von Fähigkeiten wird das Robot Operating System (ROS) und die Versionsverwaltung GIT mit CI-CD vorgestellt. Für die Verwendung auf realer Hardware werden Verteilungsmechanismen auf der Grundlage einer Containervirtualisierung und grundlegende Hardwareanbindungen in Form von Schnittstellen und Controller vorgestellt.

Die Vorstellung des Konzeptes der Selbstbeschreibung in Kapitel 5 unterteilt sich in drei Abschnitte. Im ersten Abschnitt werden die Mechanismen des Semantic Web in eine Ontologie der Selbstbeschreibung für die Verknüpfung von Eigenschaften und Fähigkeiten angewandt und mit ausführbarem Code assoziiert. Der ausführbare Code hingegen ist integraler Bestandteil einer Fähigkeit, deren Konzept im zweiten Abschnitt vorgestellt wird. Ein besonderer Fokus liegt in der Modellierung und den Verteilungsmechanismen, das durch das Fallbeispiel der synchronen Glasfasermessung veranschaulicht wird. Der letzte Abschnitt präsentiert ein Konzept für eine Simulation, die durch selbstbeschreibende Bausteine schrittweise in die Realität überführt werden kann.

In Kapitel 6 wird die Model-Domain-Domainmodel (MDDM)-Architektur vorgestellt, die in drei Schichten unterteilt ist. Die auf der verteilten Architektur aufbauende Nutzerschnittstelle unterstützt gängige objektorientierte Programmiersprachen und abstrahiert anhand von Fähigkeiten und Eigenschaften die Hardwarekommunikation und Graphenoperationen der darunterliegenden Schichten. Unterstützende Werkzeuge für eine einfache Annotation oder für die Integration in ROS sind Bestandteil des Kapitel 7.

Die Ergebnisse der einzelnen Experimente werden in Kapitel 8 vorgestellt und in die Fallstudie aus Kapitel 3 eingeordnet. Abschließend werden die Ergebnisse in Kapitel 9 zusammengefasst und bewertet, und es wird ein Ausblick auf zukünftige Entwicklungen im Semantic Plug & Play Kontext gegeben.

Zusammenfassung. In diesem Kapitel werden Arbeiten vorgestellt, die mit den Forschungsergebnissen der vorliegenden Dissertation verwandt sind. Primär wird das Thema der Selbstbeschreibung unter verschiedenen Perspektiven erläutert, sowie die Vor- und Nachteile der wissenschaftlichen Arbeiten beleuchtet. Abgeschlossen wird das Kapitel mit einer Gegenüberstellung des aktuellen Standes und den Ansätzen von Semantic Plug & Play.

2 Verwandte Arbeiten

2.1 Semantische Hardwarebeschreibung 14
2.2 Eigenschaften und Fähigkeiten 15
2.3 Modellierung von Prozessen 16
2.4 Simulation und digitale Abbilder 18
2.5 Einsatz in realen Systemen 19
2.6 Fazit und Abgrenzung zur bisherigen Forschung 21

Eine Selbstbeschreibung modularer Hardware umfasst mehrere wissenschaftliche Themengebiete und Fragestellungen, die in diesem Kapitel in einer Übersicht zusammengefasst sind. Für eine maschinenlesbare Annotation von Eigenschaften werden Lösungsansätze mit Hierarchien und Graphen aus unterschiedlichen Domänen vorgestellt. Fähigkeiten hingegen werden oft abstrakt definiert und finden sich auch unter den Begriffen Skill oder Capability in der Fachliteratur wieder. Einen durchgängigen Ansatz für die Verbindung abstrakter Fähigkeiten und konkreter Hardware findet sich sowohl in der Verwaltungsschale [230], einem Konzept der Forschungsinitiative *Industrie 4.0*, als auch in Betriebssystemen oder Protokollen wieder. Für die Modellierung eines Prozesses, also dem Zusammenspiel unterschiedlicher Fähigkeiten, werden im Folgenden grafische Prozessmodellierungen vorgestellt und die Einbindung von Eigenschaften erläutert. Der Übergang einer Prozessdefinition zur realen Anwendung läuft vor allem in der Robotik häufig über Simulationen und Visualisierungen. Das besondere Konzept des Digitalen Zwillings wird in der vorliegende Dissertation genauer betrachtet. Bei realen Systemen wird ein besonderes Augenmerk auf modulare, sich zur Laufzeit veränderbare Systeme im Bereich der modularen Robotik gelegt und die Verwendung von Fähigkeiten und Eigenschaften bewertet. Abschließend werden die Erkenntnisse zusammengefasst und mit dem Ansatz der vorliegenden Dissertation verglichen.

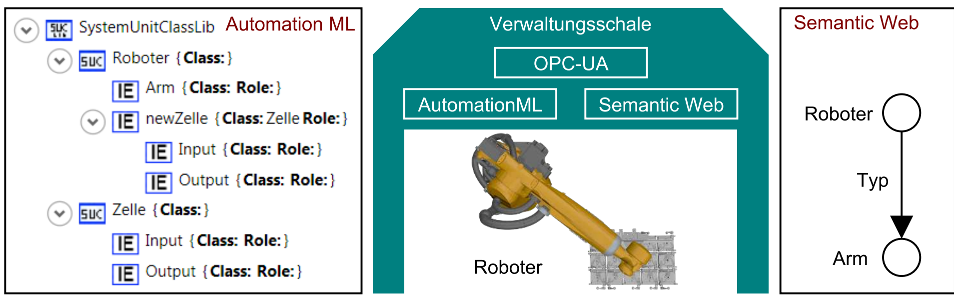

Abbildung 2.1. Beispiel einer Roboterzelle mit Verwaltungsschale modelliert in AML (links) und einer exemplarischen Modellierung in einem Graphen (rechts) als abstrakte Veranschaulichung des Semantic Web. Regelwerke für Abhängigkeiten in AML existieren nicht, so kann eine Instanz einer Zelle unter einer Roboterklasse eingefügt werden. (Eigenwerk)

2.1 Semantische Hardwarebeschreibung

Die maschinenlesbare Beschreibung von Eigenschaften bildet im Bereich der Robotik eine essentielle Brücke zwischen Insellösungen und der Wiederverwendbarkeit. Als Beispiel wird der Begriff *Industrie 4.0* eingeführt, welcher aus einer Forschungsinitiative der Bundesregierung hervorgeht [116]. Die Ziele der Industrie 4.0 sind unter anderem eine kurze Entwicklungszeit während der Produktionsplanung, einer Individualisierung nach Nachfrage und eine Flexibilität in der Produktion [169]. Für die Flexibilität in der Produktion werden das Konzept der Verwaltungsschale (Asset Administration Shell) präsentiert und unterschiedliche Technologien, wie AutomationML, OPC-UA oder das Semantic Web für die Beschreibung von Eigenschaften vorgestellt [248]. Die Verwaltungsschale wird dabei als Softwareartefakt behandelt und beispielsweise auf einem Einplatinencomputer, wie im Projekt OpenAAS [218], verteilt. AutomationML (AML) verwendet für die topologische Beschreibung das auf XML basierte Austauschformat CAEX [218] und dient der objektorientierten Beschreibung von semantischen Rollen [248]. In dem Beispiel aus Abbildung 2.1 werden die Konstrukte Roboter und Arm in der Hierarchie von AML und als gerichteter Graph in Anlehnung an das Semantic Web abgebildet. Einerseits kann durch die objektorientierte Struktur die Topologie direkt über CAEX Schnittstellen in objektorientierte Programmiersprachen überführt werden, allerdings wird durch die freie Interpretation der topologischen Anordnung die Semantik nur bedingt vermittelt. In dem Bildbeispiel aus Abbildung 2.1 wird erst durch das Tripel (Roboter, Typ, Arm) ersichtlich, dass es sich bei Arm um den Typ des Roboters handelt. Damit das Tripel nicht aus beliebigen Freitexten besteht, sieht das Semantic Web die Definition eines Vokabulars (Dictionaries) sowie Regeln auf dem Vokabular (Ontologien) vor. OPC-UA hingegen wird als Austauschformat verwendet mit eigenem Metamodell [173], dass wiederum in OPC-UA Companion Specifications konkretisiert ist [248]. Zusammengefasst begegnen sich mehrere Technologien, die ohne scharfe Abgrenzung für maschinenlesbare Eigenschaften verwendet werden können. So dienten in aktuellen Publikationen entweder AML [166] als Basis für Eigenschaften oder

es werden Semantic Web Prinzipien verwendet [289], bzw. Mischformen mit eigenen Metamodellen wie in OpenAAS [255].

Die Linked Open Data Cloud[1] illustriert sowohl die Vielfalt, als auch die größten Anwendungsgebiete im Semantic Web. Dazu zählen bspw. die Integration von Medien in das Semantic Web, wie die BBC an einem Projekt [145] illustrativ vorgeführt hat oder domänenübergreifende Projekte wie DBPedia [200], eine Überführung der Wikipedia in das Semantic Web. Bezogen auf Hardware hingegen finden sich in der Geographie zahlreiche Projekte mit dem Ziel, Messwerte von Sensoren zu teilen und anhand semantischer Annotationen interpretieren zu können. Durch die Kombinationsfähigkeit unterschiedlicher Ontologien können domänenspezifische Beschreibungen ausgedrückt und mit Messwerten assoziiert werden. Exemplarisch können die Semantic Sensor Network Ontology [210] oder OntoSensor [105] die Einheiten zu Messwerten bereitstellen und mit einer domänenspezifischen Ontologie verknüpft werden. Einige Projekte [35, 324] verfolgen diesen Ansatz mit dem Fokus, Messwerte zu annotieren und über Webservices bereitzustellen, andere kombinieren ein verteiltes Multiagentenframework mit einer Sensor- und Gebäudeontologie für die Überwachung von Gebäuden [61]. So haben sich zwar Ontologien als probates Mittel für die Beschreibung von Sensoren etabliert, eine verteilte Selbstbeschreibung für modulare Hardware mit Aktuatoren und der Beschreibung von Fähigkeiten hingegen werden in diesen Arbeiten nicht tiefergehend adressiert.

2.2 Eigenschaften und Fähigkeiten

Eine Fähigkeit definiert sich laut Duden als *durch bestimmte Anlagen, Eigenschaften geschaffene Möglichkeit, gewisse Funktionen zu erfüllen, gewissen Anforderungen zu genügen, etwas zu leisten* [74]. Der Begriff *Skill* aus der Robotik hingegen wird entweder synonym zu Fähigkeiten verwendet oder als Fertigkeit definiert, die eine erlernende Komponente in sich trägt. Fertigkeiten werden in Projekten wie KnowRob [283] oder SkillMaN [59] verwendet, um Umgebungsdaten zu klassifizieren, semantisch zu annotieren und Ähnlichkeiten zu entdecken, damit die eingesetzten Fertigkeiten, wie bspw. *navigiere zum Tisch* angewandt werden können. In beiden Projekten werden erlernte Gegebenheiten für Fertigkeiten mithilfe einer Ontologie in einer Wissensbasis gespeichert. Als Ausführungsumgebung wird das Robot Operating Framework (ROS) verwendet. In dieser findet dann die Anwendung der Fähigkeiten statt, wie zum Beispiel *fahre zu Position*, nachdem der Tisch klassifiziert und geometrisch eingeordnet wurde. ROS hingegen verwendet keine Wissensbasis mit Ontologien und folgt einem komponentenbasierten Ansatz. So müssen in der Designphase die zu verwendeten Komponenten anhand eines Launchfiles definiert werden, weshalb eine Variabilität zur Laufzeit durch die konkrete Definition der Komponenten nicht gegeben ist. Alternativ zu ROS können die Robotersysteme auch direkt über eigene SDKs angesprochen werden, wie bpsw. in dem Projekt von Huang et al. [129], das über eine Fähigkeitsmodellierung in UML-Aktivitätsdiagramm ähnlicher Struktur die Zusammensetzung von Möbeln beschreibt. Eine Variabilität der eingesetzten Hardware steht bei diesen Arbeiten nicht im Fokus.

[1]https://lod-cloud.net/

Eine alternative Betrachtungsweise ist der Fokus auf die Hardware selbst und die entsprechenden Anbindungen. Für eine gesamtheitliche Betrachtung ist wichtig, nicht nur Sensoren und Aktuatoren zu behandeln, die über übliche Schnittstellen wie dem Universal Serial Bus (USB) und einen Treiber angesprochen werden, sondern auch solche, die über einen Mikrocontroller angesprochen werden müssen. Mikrocontroller haben üblicherweise einen sehr begrenzten persistenten Speicher in Form eines Electrically Erasable Programmable Read-Only Memory (EEPROM). Ein in IEEE 1451 enthaltener Standard beschreibt Transducer Electronic Data Sheets (TEDS) für die Bereitstellung von Datenblättern auf EEPROM Speichern [232]. Dabei werden Informationen wie Hersteller, Identifikationsnummer, Typenbezeichnung und sogar Kalibrierdaten verwendet. Zwar können optional ergänzende Informationen über TEDS angelegt werden, doch die Domänenbeschränkung auf Wandler (Transducer) und der eins zu eins Beziehung einer Hardware zu einer Fähigkeit bietet ebenfalls keine Grundlage für ein modulares selbstbeschreibendes System.

Das gerade für Peripheriegeräte verwendete Plug & Play Prinzip [272] basiert auf dem Prinzip generischer Treiber für die Anbindung von Hardware. Diese können zwar zur Laufzeit angesprochen werden, spezielle Hardwareeigenschaften hingegen werden ignoriert und müssen mit konkreten Treibern ersetzt werden. Der Begriff Plug & Work oder Plug & Produce im Englischen beschreibt eine Vorgehensweise, bei der ein Gerät innerhalb einer Fabrikanlage nach dem Plug & Play Prinzip verwendet werden kann. Jedoch werden hier nicht zwangsweise generische Treiber verwendet, sondern eine mithilfe der Verwaltungsschale bereitgestellte Schnittstelle benutzt. Da die Domäne der industriellen Fertigung allerdings spezielle Problemstellungen, wie die Echtzeitfähigkeit aufeinander abgestimmter Systeme aufweist, werden die Fähigkeiten wiederum in den dafür vorgesehenen Umgebungen gekapselt, wie bspw. einer Roboterzelle, die nicht den Anspruch hat, zur Laufzeit umkonfiguriert zu werden [264]. Der Ansatz eines Softwareartefakts als Verwaltungsschale auf einem Hardwareadapter ist jedoch auch für modulare, selbstbeschreibende Systeme anwendbar. Eine Dokumentation anhand der Eigenschaften hingegen steht nicht im Fokus der Verwaltungsschale.

2.3 Modellierung von Prozessen

Für die Modellierung von Prozessen werden folgend zwei Arten differenziert betrachtet. Einerseits grafische Modellierungssprachen mit einem Fokus auf dynamischen Modellen, die für eine Codegenerierung verwendet werden, sowie andererseits Modellierungen innerhalb einer konkreten Domäne. Die Unified Modeling Language (UML) der Object Management Group [216] umfasst in ihrer aktuellen Spezifikation 2.5.1 unterschiedliche Diagrammtypen mit entsprechenden Aufgabengebieten. Grob unterteilen lassen sich die Diagrammtypen in eine dynamische Modellierung von Verhalten und Interaktion, wie beispielsweise Aktivitätsdiagramme für Kontroll- und Objektflüsse (siehe Abbildung 2.2, Diagramm (3)) und einer strukturellen Modellierung, wie zum Beispiel Verteilungsdiagramme für die Verteilung von Softwareartefakten auf Hardwareknoten oder Klassendiagramme. Die Diagrammtypen können dabei in verschiedenen Entwicklungsstufen eines Softwareprozesses eingesetzt werden, wie bspw. ein Klas-

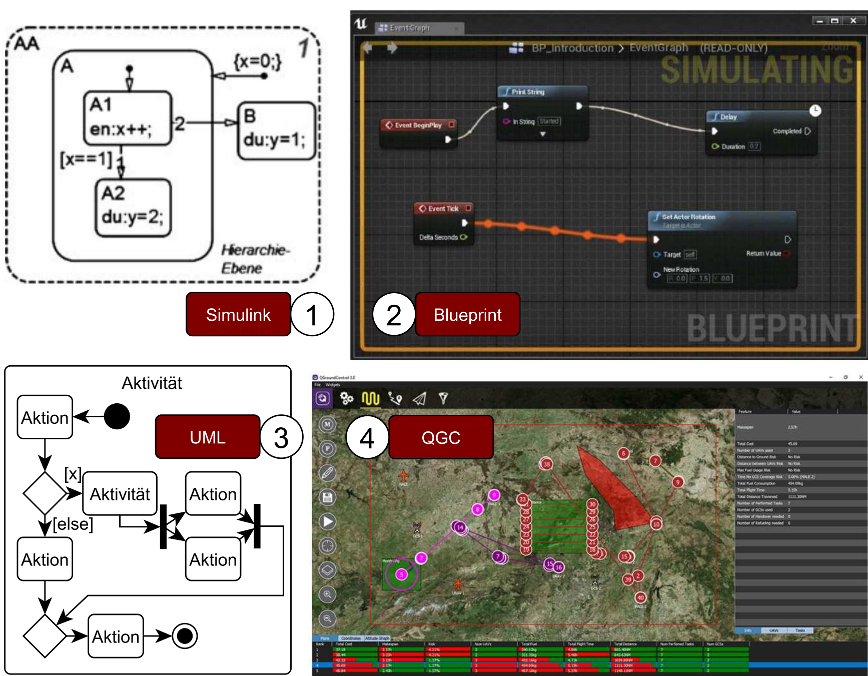

Abbildung 2.2. Abbildung unterschiedlicher grafischer Modellierungen für dynamische Abläufe. (1) zeigt eine abstrakte Modellierung in Matlab Simulink mit Hierarchieebenen [229]. In (2) ist ein Blueprint aus der Unrealengine abgebildet mit hervorgehobener Kante für die Visualisierung einer Datenübermittlung zur Laufzeit [294]. Ein abstraktes Aktivitätsdiagramm aus der UML ist in (3) illustriert, während in (4) eine konkrete Missionsplanung in einer Erweiterung der QGroundControl [238] dargestellt wird. (Die Bilder 1,2 und 4 sind aus den Referenzen der Bildbeschreibung entnommen.)

sendiagramm als Domänenmodell für die abstrakte Modellierung der Domäne oder aber als Designklassendiagramm für konkrete Designentscheidungen der Implementierung. Für einen nahtlosen Übergang des Designs in Code (Forward Engineering) bieten Modellierungstools wie MagicDraw[2] oder Modelio[3] Codegeneratoren an, einzelne dynamische Modelle werden übersetzt [295] oder Projekte verzahnen unter dem Begriff *Round-Trip-Engineering* auf dem Rückweg vom Code in das Model [9]. In der Domäne der technischen, mathematischen und physikalischen Modellierung von Blockdiagrammen finden sich Modellierungswerkzeuge wie u.a. Matlab Simulink (siehe (1) in Abbildung 2.2) mit eigener Syntax, Hierarchien und umfangreichen Werkzeugen wieder. Für konkrete Domänenprobleme hingegen, wie der Modellierung einer Mission für Quadrocopter, werden eigene Tools wie die *QGroundControl* (4) in Abbildung 2.2 mit unterliegender Karte oder in der Spieleentwicklung *Blueprints* (2), einer Modellierung von eventbasierten Abläufen innerhalb der Unrealengine[4] verwendet.

[2]www.magicdraw.com (Stand: 17.05.22)
[3]www.modelio.org (Stand 17.05.22)
[4]www.unrealengine.com (Stand: 17.05.22)

In der UML können Kontrakte in Kombination mit der Object Constraint Language (OCL) eine sehr gute Syntax und Semantik für die Beschreibung von Systemoperationen bieten und damit auch Freiraum für eine Dokumentation liefern. Allerdings basieren Kontrakte auf Freitexten und in OCL können Restriktionen spezifiziert werden, wie bspw. die Einschränkung eines Attributes auf einen Wertebereich. Die Annotation mittels eines Vokabulars und zugehöriger Ontologie ist nicht Bestandteil. Die Knoten eines Blueprints werden in Inputs und Outputs unterteilt und mit Namen und Datentyp gekennzeichnet. Implizite Konvertierungen der Datentypen und eine Hervorhebung der verwendeten Transitionen zur Laufzeit erleichtern zwar eine schnelle Realisierung kleiner Projekte, fehlende Dokumentationsmöglichkeiten mit gemeinsamem Vokabular erschweren in großen Projekten hingegen die Wiederverwendbarkeit.

2.4 Simulation und digitale Abbilder

Modulare selbstbeschreibende Systeme haben besondere Anforderungen an Simulationen und Visualisierungen, da sich Fähigkeiten erst nach einer physikalischen Verbindung ergeben, bzw. auch die Visualisierung als Teil der Selbstbeschreibung zu betrachten ist. Im Bereich der Simulationsumgebungen für Robotersysteme wie bspw. Gazebo für ROS [88] oder auch in Visualisierungen wie der Unrealengine oder Unity[5] werden Visualisierungselemente vor der Ausführung definiert. Visualisierungselemente bestehen aus einem 3D Modell (Mesh oder CAD) mit ergänzenden Informationen wie bspw. der Kinematisierung, wie in Abbildung 2.3 illustriert. Formate wie Collada [12] unterstützen dabei nicht nur Mesh-Modelle, sondern kapseln auch die Texturen und Kinematisierung in ein Dateiformat. Dieses Format wird z.B. in AutomationML verwendet und kann in den Hierarchien von CAEX verlinkt werden [69].

Als Schlüsselkonzept für die Industrie 4.0 wird als Ergänzung zu den Plug & Work Prinzipien der Begriff **Digitaler Zwilling** eingeführt und wie folgt definiert:

> Die Vision des Digitalen Zwillings selbst bezieht sich auf eine umfassende physikalische und funktionale Beschreibung zusammen mit allen verfügbaren Betriebsdaten einer Komponente, eines Produkts oder Systems, die mehr oder weniger alle Informationen enthalten, die in allen - aktuellen und nachfolgenden - Lebenszyklusphasen nützlich sind. (Übersetzung) [30]

Zu den Informationen werden auch die virtuellen Abbilder für eine Visualisierung innerhalb einer Simulationsumgebung sowie Schnittstellen zu simulierten Fähigkeiten gezählt [30]. In Kombination mit der Verwaltungsschale könnten diese Daten auch dezentral, an der Hardware selbst über einen Hardwareadapter zur Verfügung gestellt werden. Eine konkrete Umsetzung einer dezentralen Simulation mit Hardwareadaptern und digitalen Zwillingen ist dem Autor jedoch nicht bekannt.

[5] www.unity.com (Stand: 17.05.22)

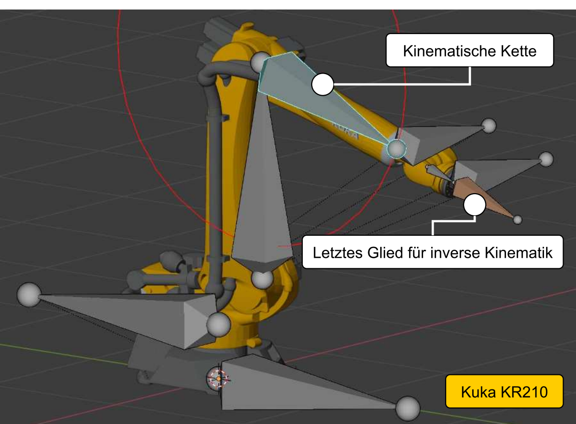

Abbildung 2.3. Abbildung eines KUKA Roboters in Blender mit kinematischer Kette in den Achsen und einem letzten Glied am Endeffektor für die inverse Kinematik. Der Verbund aus Modell und kinematischer Ketten kann in Collada-Dateien gespeichert werden. (Rendering und Kinematisierung sind Eigenwerke)

2.5 Einsatz in realen Systemen

Im Bereich der modularen Robotik gibt es unterschiedliche Ansätze der Modularität, die in Abbildung 2.4 anhand drei repräsentativer Projekte illustriert sind.

Cubelets [110] steht stellvertretend für die Variante **Fähigkeiten mit generischer Schnittstelle**. Unterteilt werden die Würfel in vier Kategorien: Sensoren, Aktuatoren, Stromversorgung und logische Bausteine. Über eine proprietäre, physikalische Schnittstelle können die unterschiedlichen Würfel miteinander verbunden werden, wobei bauartbedingt vier Freiheitsgrade möglich sind. Die logische Schnittstelle umfasst dabei lediglich ein Byte mit Werten zwischen 0 und 255. Ein Distanzsensor übergibt bspw. an alle anliegenden Würfel seinen Messwert normiert weiter und ein Würfel mit einem Rad übersetzt diesen Wert in die Drehgeschwindigkeit. Die Stromversorgung hingegen wird an alle angeschlossenen Würfel transitiv weitergeleitet. Bereits bei Zusammenschluss dreier Würfel kann ein *follow me* Roboter ohne zentrale Kontrolleinheit entstehen, der bei Näherung an ein Objekt die Drehzahl reduziert [110]. Mit den logischen Bausteinen kann das Byte manipuliert werden, wie beispielsweise ein Würfel für die Invertierung. Damit könnte bspw. der harmlose *follow me* Roboter zu einem *hit me* Roboter umfunktioniert werden, der bei Näherung beschleunigt. Der rein auf Fähigkeiten basierte, stark modulare Ansatz bringt jedoch keinerlei Selbstbeschreibung der Würfel mit sich.

Der SMC-Rover hingegen konkretisiert das Prinzip der **Fähigkeiten durch Zusammenschluss**. Aufgebaut ist der SMC-Rover aus einem Hauptmodul und mehreren Kindmodulen wie in Abbildung 2.4 zu sehen. Die Kindmodule bestehen aus einem Rad und einem Greifer und können autark fahren oder im gekippten Zustand als Greifer Montageaufgaben übernehmen. Das Basismodul, ausgestattet mit Solarpaneelen und großer Batterie, erhält die Fähigkeit *Fahren*, sobald sich mindestens drei Kindmodule selbstständig an die entsprechenden Adapterpunkte angedockt haben [140]. Der in

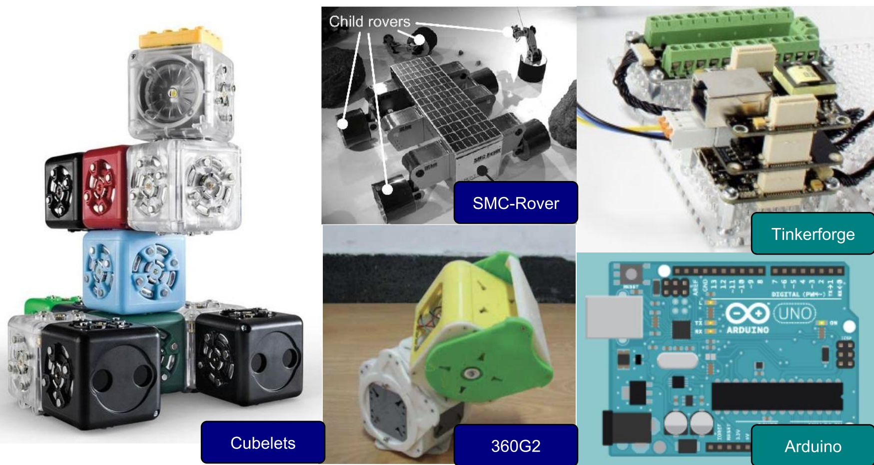

Abbildung 2.4. Abbildung drei unterschiedlicher modularer Robotersysteme. Cubelets (links) dienen als Tangible User Interface für einfache Robotersysteme [110]. 360G2 sind selbst assemblierende Würfelsysteme mit lokomotorischen Fähigkeiten [39] und der SMC-Rover besteht aus unterschiedlichen modularen Bauteilen mit eigenen Fähigkeiten [140]. Tinkerforge [287] und Arduino [11] dienen zur Integration und Steuerung von Sensoren und Aktuatoren. (Die Bilder sind den entsprechenden Quellen der Bildbeschreibung entnommen.)

Abbildung 2.4 abgebildete 360G2 Roboter besteht ebenfalls aus einzelnen Modulen, die einzeln betrachtet nur eingeschränkte Fähigkeiten besitzen, durch Zusammenschluss aber schlangenartige lokomotorische Fähigkeiten entwickeln [39]. Die sehr domänenspezifischen Anwendungen verwenden zwar keine Selbstbeschreibung, aber adressieren einen interessanten Aspekt eigenständiger Module mit Fähigkeiten, die sich aus der Kombination einzelner Elemente ergeben.

Abseits der modularen Robotik haben sich für die Integration einfacher Sensoren und Aktuatoren auch Systeme wie Arduino und Tinkerforge etabliert. Tinkerforge basiert auf einem Baukastenprinzip mit normierten Steckkontakten und verschiedenen Bausteinen, die auch Bricks, Bricklets und RED Brick genannt werden [287]. Das verfolgte Ziel ist eine einfache Integration von Sensoren und Aktuatoren ohne elektrotechnische Grundkenntnisse. So kann beispielsweise ein Temperatursensor (Bricklet) an einen stackbaren Brick angeschlossen werden, unter dem sich ein programmierbarer RED Brick befindet. Zwar wird Tinkerforge unter anderem in Projekten wie [141] in eine Objektstruktur gekapselt und einer Position im Raum zugeordnet, jedoch basiert die Zuordnung der Sensoren und Aktuatoren auf den eindeutigen Identifikatoren der proprietären Bricklets, die eine Klassifizierung erlauben. Zwar gibt es auch spezielle Bricklets für den Anschluss fremder Sensoren/Aktuatoren, eine Selbstbeschreibung hingegen ist nur durch die Integration innerhalb des RED-Bricks möglich. Deutlich verbreiteter als Tinkerforge ist die Arduino Plattform, die sowohl aus quelloffener Software als auch Hardware besteht. Neben einer IDE und reichhaltigen Bibliotheken für Sensoren und Aktuatoren existieren unterschiedlichste Hardwaremodelle (in Abbildung 2.4 ist bspw. ein Arduino UNO abgebildet). Die quelloffene Dokumentation der Hardware führt hingegen zu einer steten

Erweiterung der Produktpalette durch kompatible Replika und Erweiterungsplatinen, wie bspw. einem Motortreiber. Durch die einfache, wenn auch manuelle, Integration nahezu beliebiger Sensoren und Aktuatoren dient das Arduino Projekt auch in vielen wissenschaftlichen Projekten als Hardwarebasis, wie beispielsweise für Sensornetze [86], Robotersysteme [313], etc. Eine ID-basierte Lösung wie bei Tinkerforge ist auf der Arduino Plattform prinzipiell möglich, durch die Vielfalt der Sensoren/Aktuatoren hingegen nur mit Einschränkung und im Widerspruch der offenen Plattform realisierbar. Eine Selbstbeschreibung, die auch auf der Plattform angewandt werden kann, wird bisher nicht fundiert betrachtet.

2.6 Fazit und Abgrenzung zur bisherigen Forschung

In der Forschung existieren viele Ansätze für die Beschreibung von Eigenschaften, die Abstraktion und der Modellierung von Fähigkeiten, Simulationsumgebungen und realisierter modularer Hardware in unterschiedlichen Domänen. In der Domäne meteorologischer Messungen werden Eigenschaften der Hardware vor allem in der Nachbearbeitung von Sensordaten relevant. Fähigkeiten in Abhängigkeit zu Eigenschaften werden indes nicht verwendet und üblicherweise programmatisch gelöst, wie etwa der SMC-Rover, mit der Fähigkeit *Fahren*, wenn die Räder montiert sind. Der Ansatz modularer, selbstbeschreibender Module eröffnet auch Fragestellungen der Programmierung und Dokumentation. Zur Designzeit sollen aktuelle Hardwarekonfigurationen und deren Dokumentation zwar einsehbar, nicht aber zwingend an das System angeschlossen sein. Während der Laufzeit kann sich die Hardwarekonfiguration indes ändern. Dies klappt bei reaktiven Systemen wie Cubelets gut, beim Einsatz komplexer Middleware wie dem Robot Operating System (ROS) ist dieser Fall nicht vorgesehen.

In Semantic Plug & Play soll durch eine Erweiterung ROS nicht nur für selbstbeschreibende, modulare Hardware gerüstet, sondern auch um eine grafische Schnittstelle für die dynamische Komponentenmodellierung ergänzt werden. Das Konzept des digitalen Zwillings kann hingegen mit der Selbstbeschreibung in Einklang gebracht werden, mit besonderen Anforderungen an die Simulationsumgebungen, wie etwa der Variabilität zur Laufzeit. Ein erheblicher Anteil der Dissertation ist die Realisierung der tatsächlichen Hardware und die Beweisführung für die Tragfähigkeit der Konzepte von Semantic Plug & Play. Dabei sollen die Vorzüge der Arduino Plattform mit den einfachen Mechaniken modularer Robotersysteme kombiniert werden. Zusammengefasst werden in Semantic Plug & Play folgende Eigenschaften präsentiert:

- **Semantische Selbstbeschreibung** für Eigenschaften und Fähigkeiten von Hardwareelementen mit Speicher und Abrufmechanismen für Mikrocontroller, Einplatinencomputer und PCs.
- **Fähigkeiten durch Eigenschaften dokumentieren** mit Werkzeugen für aufeinander aufbauende Fähigkeiten, einfache Verteilung und Versionierung.
- **Modellierung generischer und domänenspezifischer Abläufe** mit Erweiterungen für vorhandene Systeme und einer objektorientierten Schnittstelle für Multiagentensysteme.

- **Adaptive Simulationsumgebung mit schrittweiser Realisierung**, in der sowohl reale Hardware als auch digitale Abbilder miteinander kombiniert und zur Laufzeit integriert werden können.
- **Modulare Hardwarerealisierung** durch eine breite Unterstützung unterschiedlicher Sensor- & Aktuator-Typen, sowie technische Zeichnungen für den 3D-Druck, Platinenlayouts und Methodologien.

Mithilfe dieser Eigenschaften werden nicht nur exemplarisch modulare Roboter vorgestellt, die innerhalb einer Domäne operieren können, sondern vielmehr eine Architektur und Methodologie präsentiert, die gestützt durch unterschiedliche Werkzeuge, eine Selbstbeschreibung ermöglicht.

Zusammenfassung. In diesem Kapitel wird die Fallstudie eines Gasunfalls vorgestellt, in der ein modularer und rekonfigurierbarer Quadrocopterschwarm eingesetzt wird. Da die Fallstudie sehr viele Themengebiete adressiert, werden einzelne Aspekte in übersichtliche Experimente verpackt und einzeln vorgestellt. Die Experimente dienen auch als leitende Beispiele in den folgenden Kapiteln.

Fallstudie: Modulare Quadrocopterschwärme in Katastrophenszenarien

3.1 Aufbau und Ablauf **25**
3.2 Multipotente Systeme **26**
3.3 Fallstudienbezogene Experimente **28**
3.3.1 Virtuelle Messtürme 28
3.3.2 Synchrone Glasfasermessung 29
3.3.3 Ausbreitungsmodellierung 30
3.3.4 Der sensorbasierte Flug 32
3.3.5 Pick & Place 33
3.3.6 GoLive 34

Im Verlauf der Dissertation werden die Ergebnisse anhand einer Fallstudie zu einem Chemieunfall bewerten, der durch ein fiktives Beispiel in Abbildung 3.1 illustriert wird. Diese Fallstudie dient ebenfalls im DFG geförderten Projekt *COMBO* (als auch in der dazu veröffentlichten Publikation [153]) als leitendes Beispiel. Leider passieren Chemieunfälle recht häufig, wie Vorfälle bei BASF in Deutschland (2016) [45], bei Arkema in Texas (2017) [46], und bei Bayernoil in Deutschland (2018) [82] exemplarisch zeigen. Bei Chemieunfällen sind die Einsatzkräfte oft mit der Bedrohung durch giftige Gase konfrontiert. Zunächst muss die Situation geklärt werden, indem der relevante Parameter (d. h. das Gas mit dem höchsten Gefährdungspotenzial sowie dessen Quelle) ermittelt wird (1). Nach der Identifizierung dieses Parameters (z. B. eines bestimmten Gases), muss seine Verbreitung kontinuierlich beobachtet werden, um seine Schädlichkeit abschätzen zu können (2). Aus der Beobachtung kann sich die Notwendigkeit geeigneter Reaktionen ergeben, z. B. die Evakuierung bedrohter bewohnter Gebiete (3). Gegenwärtig müssen die Einsatzkräfte ihre Beobachtungen manuell durchführen und ihre Entscheidungen auf der Basis begrenzter, unvollständiger oder sogar falscher Informationen treffen [55]. So werden beispielsweise Messungen der Gaskonzentrationen hauptsächlich an dem Ort, an dem sich der Unfall höchstwahrscheinlich ereignet hat, und an wenigen weiteren

Abbildung 3.1. Exemplarischer Chemieunfall innerhalb von Augsburg mit austretender Gaswolke. Die einzelnen Schritte (1)-(3) skizzieren das Vorgehen der Berufsfeuerwehr in solchen Situationen. (Bild stammt aus Google Maps)

Punkten in Bodennähe durchgeführt, da nur wenig Personal zur Verfügung steht oder Messungen durch unwegsames Gelände erschwert werden. Daher müssen sich die Einsatzkräfte in gefährdete Gebiete begeben und sich dabei möglicherweise gefährlichen Gasen aussetzen. Darüber hinaus ist es schwierig, einen vollständigen Überblick über das betroffene Gebiet zu bekommen, der nur auf diesen wenigen lokalen Beobachtungen beruht. Unter Verwendung dieser groben Schätzungen, aufgrund von wenigen Bodenmessungen, unterschiedlichen Ausdehnungsmustern der Gase oder möglicherweise veralteter Wetterdaten müssen weitreichende Entscheidungen über die Evakuierung nahe gelegener Gebiete getroffen werden, z.B. Wohnhäuser, Altenheime oder sogar Krankenhäuser. Ein typisches Vorgehen bei der Evakuierung kritischer Infrastruktur sind bspw. Lautsprecherdurchsagen oder bei ausreichender Verfügbarkeit an Personal auch das Klingeln an Türen. Dennoch ist es nicht einfach, alle Menschen vor allem in weitläufigen oder dicht bebauten Gebieten zu erreichen. Eine skalierende Technik mit wenig Personalbedarf fehlt ebenfalls, um sich in solchen Situationen einen Überblick zu verschaffen. So werden bspw. niedrig priorisierte Aufgaben vernachlässigt, wie das Aufzuspüren einzelner Personen in Gefahrengebieten, die im weitläufigen Terrain spazieren gehen. Ein Quadrocopter-Ensemble, also eine Formation zusammenarbeitender, intelligenter Geräte könnte die Bewältigung solcher Unfälle erheblich verbessern, wie das Projekt Airshield [55] demonstriert, in welchem z. B. die Verfolgung von Gaswolken mit einem Schwarm von Mikro-Quadrocopter realisiert wurde. Für den gezielten Einsatz eines modularen Quadrocopter-Ensembles für die Unterstützung aller Etappen eines Chemieunfalls werden folgend die Struktur und Maßnahmen erläutert, die dafür nötig sind.

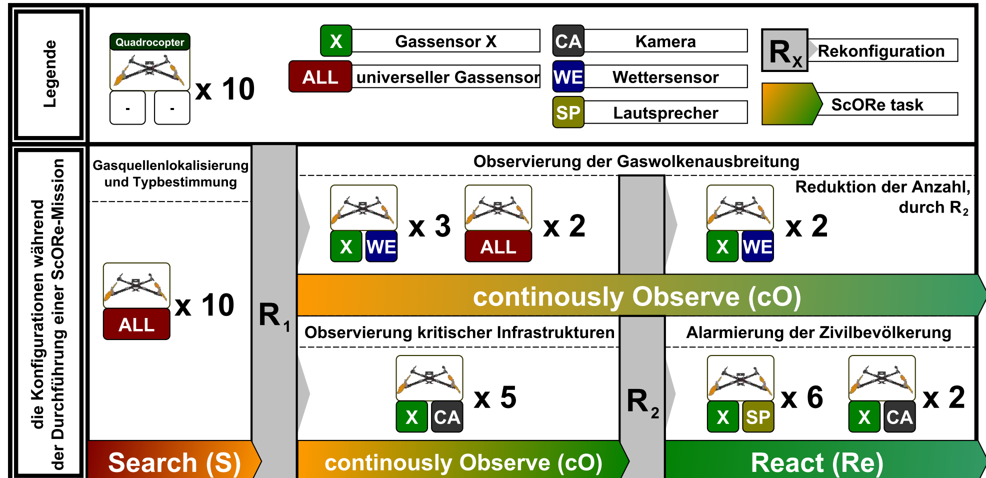

Abbildung 3.2. Untergliederung der Schritte bei einem Gasunfall in eine exemplarische ScORe-Mission mit zehn konfigurierbaren Quadrocoptern, sowie modularen Sensoren und Aktuatoren. Für die einzelnen Schritte werden jeweils Konfigurationsvorschläge sowie die nötige Rekonfigurationsphasen (R) skizziert. (Bild aus der kooperativen Publikation [153] entnommen.)

3.1 Aufbau und Ablauf

Für eine abstrakte Struktur der Phasen und den Einsatz benötigter Materialien innerhalb eines Chemieunfalls werden sogenannte ScORe-Missionen definiert, die in der kooperativen Publikation [153] vorgestellt sind. ScORe-Missionen bestehen typischerweise aus mehreren Aufgaben (ScORe-Tasks), die sequentiell oder parallel ausgeführt werden. Außerdem gibt es Abhängigkeiten zwischen Aufgaben und unterschiedliche Anforderungen an das Quadrocopter-Ensemble, um alle Aufgaben erfolgreich zu erfüllen. Für die erste Aufgabe (***S**earch*) aus Abbildung 3.2 müssen die Quadrocopter mit universellen Gassensoren ausgestattet werden, um das relevante Gas zu identifizieren, während sie für die nächste Aufgabe (*continously* ***O**bserve*) mit einem präziseren Sensor für das identifizierte Gas ausgestattet werden. Für die letzte Aufgabe (***R**eact*) müssen einige Quadrocopter z. B. mit Kameras oder Lautsprechern zur Warnung von Menschen ausgestattet werden. Neben den unterschiedlichen Aufgabenstellungen einer ScORe-Mission muss das Ensemble auch mit dynamischen Aspekten umgehen können. Beispiele für dynamische Aspekte sind Ausfälle der Hardware zur Laufzeit, Umwelteinflüsse, auf die das Ensemble reagieren muss, und situationsbedingte Anpassungen. Diese Aspekte erschweren die Berechnung eines vollständigen Plans (einschließlich Aufgabenplanung und -zuweisung) für jeden Quadrocopter.

Ein hybrider Ansatz mit abstrahierter Planung und adaptiven Reaktionsmöglichkeiten zur Laufzeit von Ensembles in ScORe-Missionen steht im Fokus der Dissertation meines geschätzten Kollegen Oliver Kosak [150]. Die modularen, selbstbeschreibenden Aktuatoren und Sensoren für Rekonfigurationen zur Laufzeit in ScORe-Mission sind dagegen Bestandteil der vorliegenden Dissertation. Das Gesamtkonzept eines modularen, rekonfigurierbaren Quadrocopter-Ensembles wird im nächsten Abschnitt vorgestellt.

3.2 Multipotente Systeme

Die Verwendung heterogener Roboter mit unterschiedlichen Fähigkeiten findet sich bspw. in Suchmissionen [56, 207, 260], der verteilten Überwachung kritischer Infrastruktur [193, 195] oder der Umweltforschung [70, 284, 317] wieder. Technologien, die eine Zusammenarbeit der unterschiedlichen Roboter ermöglichen, sind in vielen Forschungsdisziplinen zu finden. Diese reichen von komplexen Planungs- und Koordinationsansätzen zur Ermittlung sinnvoller Handlungssequenzen [76], bis zu sogenannten Schwarmintelligenztechnologien [64, 111], die lokale Interaktionen zwischen Robotern (Emergenz [99]) ausnutzen, um robuste und skalierbare Ausführungen zu ermöglichen. Für jeden neuen Anwendungsfall werden aktuell jedoch neue Roboter mit neuen Fähigkeiten für neue Anwendungen entworfen, gebaut und mit individualisierter Software programmiert (vgl. aktuelle Ansätze [56, 70, 193, 260]). Für einen wiederverwendbaren Ansatz werden folgende Problemstellungen diagnostiziert:

Ressourcen und technischer Aufwand
Der Bedarf an Robotern wächst proportional mit der Anzahl an verschiedenen Aufgaben und Anwendungsfällen. Im Beispiel aus Abbildung 3.2 wären 24 spezialisierte Quadrocopter nötig, statt 10 konfigurierbarer.

Abwägung zwischen Robustheit und Vielseitigkeit
Die Heterogenität von Roboter-Ensembles führt zu einer geringeren Robustheit gegenüber Ausfällen, d.h. wenn ein spezialisierter Roboter ausfällt, muss er durch einen gleichwertig ausgestatteten Roboter ersetzt werden. Homogene Systeme, also ein Roboter-Ensemble, in welchem jeder Roboter die gleichen Fähigkeiten besitzt, sind dagegen auf eine bestimmte Aufgabe spezialisiert und können nicht für andere Aufgaben eingesetzt werden. Sie sind jedoch robust gegenüber Ausfällen.

Planungs- und Zuweisungskomplexität
Das Erstellen von Plänen, die einen Ablauf der Ausführung definieren, ist NP-vollständig [80], während die Zuteilung dieser Aufgaben an Agenten NP-schwer ist und durch größere Roboterteams sowie deren Heterogenität verschlimmert wird [95].

Proprietäre Softwarelösungen
In vielen Projekten wird die Software, d.h. die realisierten Fähigkeiten des Robotersystems, genau für die gegebenen Anwendungsfälle und die verwendete Hardware entwickelt [87, 251, 284]. Systeme mit rekonfigurierbaren Fähigkeiten sind jedoch derzeit entweder nicht erweiterbar [110] oder nur in Simulationen vorhanden [29].

Spezialisierte Hardwareplattform
Hardwareplattformen sind stark von den Szenarien abhängig [111, 251, 263] und können nicht ohne großen technischen Aufwand für andere Zwecke verwendet werden.

Mit dem Ansatz multipotenter Systeme für Roboter-Ensembles sollen diese Probleme überwunden werden, indem im Rahmen der Dissertation einzelne Roboter modular, rekonfigurierbar und mit Selbstbeschreibungsmechanismen ausgestattet werden. Für

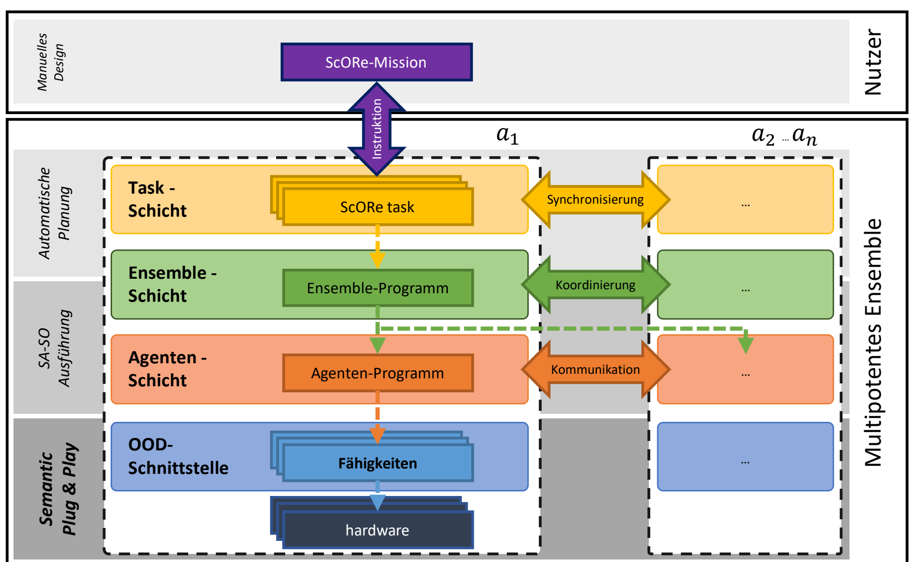

Abbildung 3.3. Mehrschichtige Systemarchitektur mit Fokus auf Roboter a_1 eines Multipotenten Ensembles A und dessen Interaktionen (durchgezogene Pfeile) mit dem Benutzer sowie anderen Robotern $\{a_2, ..., a_n\}$. Gestrichelte Pfeile symbolisieren eine Instanziierung auf der Ensemble-, Agenten- und Fähigkeits-Schicht. Unterschiedliche Hintergrundfarben stellen Änderungen in den Technologien dar, um Situationen zu bewältigen, auf die das ScORe-Aufgabenmuster angewendet werden kann. (Die Abbildung der Architektur ist aus der kooperativen Publikation [153] entnommen.)

Ensembles wird eine Hybridvariante homogener und heterogener Systeme verfolgt, die in der kooperativen Publikation [153] vorgestellt und in der Dissertation von Kosak [150] finalisiert ist. So sollen Agenten multipotenter Systeme durch ihre Laufzeitheterogenität sehr individuell gestaltet werden können [148] und sich aufgrund ihrer Homogenität zur Entwurfszeit im Falle von Ausfällen leicht gegenseitig kompensieren.

Der durchgängige Ansatz zwischen dieser Dissertation und der Dissertation von Herrn Kosak ist in einer Referenzsystemarchitektur (siehe Abbildung 3.3) für Ensembles, die auf Multipotenten Systemen aufbaut und für ScORe-Missionen ausgelegt ist. Zur Definition von ScORe-Missionen dient die sogenannte **Task-Schicht**, die eine domänenspezifische Problembeschreibungssprache aufbauend auf hierarchischen Aufgabennetzwerken (HTN) [93] verwendet. HTNs sind besonders praktisch für autonome Systeme, die in realen Umgebungen eingesetzt werden, und ermöglichen den Entwurf von wiederverwendbaren, parametrisierbaren Plänen, die für verschiedene ScORe-Missionen modular zusammengestellt werden können [186]. Primitive Aufgaben von angepassten HTNs werden in der **Ensemble-Schicht** als ScORe-Aufgaben auf kollektiver Ebene formuliert. Dafür kommen vor allem Algorithmen aus der Selbstorganisation zum Tragen, um den gewünschten Effekt als Ergebnis der Emergenz [99] zu erzielen. Die **Agenten-Schicht** ist für die Ausführung der notwendigen lokalen Regeln verantwortlich unter Verwendung von Fähigkeiten. Über die **Object Oriented Design (OOD)-Schnittstelle** werden abstrakte Fähigkeiten für Multiagentensysteme bereitgestellt. Rekonfigurierbare

Hardware, Selbstbeschreibungsmechanismen und eine Methodologie für die Integration von Robotersystemen (Quadrocopter und mobile Bodenroboter) fallen unter den Begriff Semantic Plug & Play und werden detailliert im Verlauf der vorliegenden Dissertation vorgestellt.

Die vorgestellte Fallstudie des Chemieunfalls in Abbildung 3.1 und das exemplarische Vorgehensmodell aus Abbildung 3.2 bieten eine Vielzahl unterschiedlicher wissenschaftlicher Fragestellungen, die in einzelnen Experimenten veranschaulicht werden. Dabei wird das Anwendungsprinzip der ScORe-Missionen angewendet, das nicht auf die exakte Einhaltung der Abfolge beschränkt ist. Neben einfachen ***S**earch*-Missionen mit einem Quadrocopter [221] können auch Kombinationen wie ***S**earch & **Re**act*-Missionen mit heterogenen Robotersystemen [94] realisiert werden. Folgend werden ScORe-Missionen vorgestellt, die das Prinzip von Semantic Plug & Play als Grundstein für Multipotente Systeme anhand unterschiedlicher Experimente illustrieren. Alle Experimente befassen sich mit den Herausforderungen der Fallstudie, so werden in situ Messungen mit Quadrocoptern durchgeführt, heterogene Schwärme untersucht und modulare Hardware für Rekonfigurationen getestet.

3.3 Fallstudienbezogene Experimente

Im Rahmen dieser Arbeit wurden mehrere Experimente konzipiert und durchgeführt, die Teilaspekte der Fallstudie behandeln. Jedes Experiment bildet eine eigene ScORe-Mission in unterschiedlichen Varianten. In Kapitel 5, Kapitel 6 und Kapitel 7 dienen die einzelnen Experimente je nach Fokus als leitendes Beispiel mit einer detaillierteren Einführung in den Aufbau und Ablauf. Da alle Experimente mit realer Hardware durchgeführt wurden, sind in Kapitel 8 die konkrete Umsetzung sowie die Ergebnisse zusammenfasst.

3.3.1 Virtuelle Messtürme

Das Experiment der *virtuellen Messtürme* fokussiert die in situ Messungen mit Quadrocoptern der Fallstudie. Die atmosphärische (oder planetarische) Grenzschicht spielt in vielen Bereichen eine wichtige Rolle, darunter Luftverschmutzung, landwirtschaftliche Meteorologie, Hydrologie, Flugmeteorologie, mesoskalige Meteorologie, Wettervorhersage und Klima [92]. Um die Grenzschichten besser zu erforschen und Klimamodelle zu verifizieren, werden unterschiedliche Messinstrumente verwendet [317], wie bspw. klassische Messtürme, exemplarisch illustriert in Abbildung 3.4. Die Idee des Experiments der *virtuellen Messtürme* ist es, in situ Messungen mit auf- und absteigenden Quadrocoptern analog zu den Messpunkten eines Messturms durchzuführen. Die Quadrocopter verweilen eine kurze Zeit in den vorgesehenen Messpunkten, um eine präzise Datenauswertung mehrerer Messungen durchzuführen. Der Vorteil dieser Methodik liegt in der Mobilität, einer einfachen Skalierbarkeit und der Möglichkeit, auf interessante Messwerte mit Ablaufänderungen zu reagieren. In Abbildung 3.4 werden drei Quadrocopter mit einem Kombinationssensor ausgestattet, der sowohl die Temperatur, als auch die Luftfeuchte misst. Sofern nicht auf meteorologische Phänomene reagiert wird, ist der virtuelle Messturm als klassische **c**ontinously **O**bserve Mission zu klassifizieren. Eine besondere Herausforderung an Semantic Plug & Play liegt in der Verwertung der Daten,

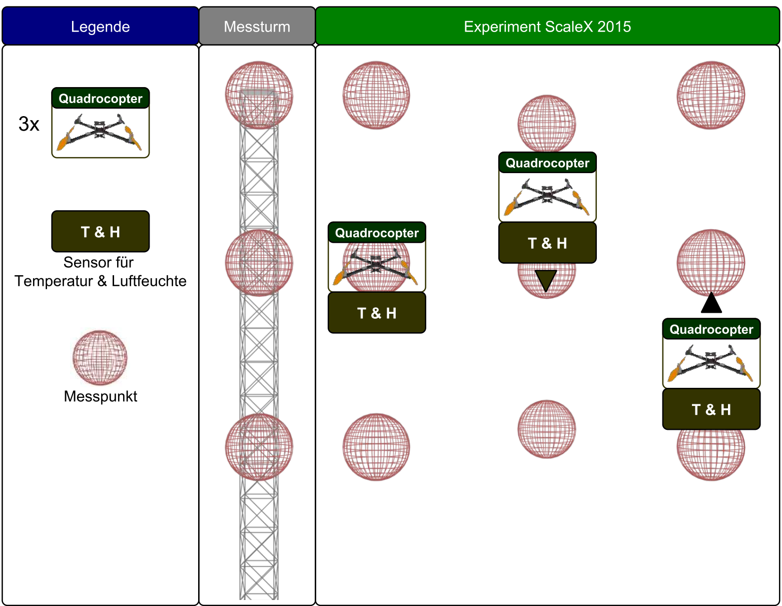

Abbildung 3.4. Links ist als Referenz ein statischer Messturm mit drei Messpunkten skizziert. Rechts werden drei virtuelle Messtürme durch auf und absteigende Quadrocopter illustriert, die an den Messpunkten stehenbleiben, um die Temperatur und Luftfeuchte zu messen. (Eigenwerk)

denn es müssen nicht nur die Daten der Sensoren ausgewertet werden, sondern auch ein Zeitstempel und die Position des Quadrocopters. Für die Einordnung in die Fallstudie des Chemieunfalls sind besonders die Messwertgradienten durch in situ Messungen relevant.

3.3.2 Synchrone Glasfasermessung

Das Experiment der *synchronen Glasfasermessung* demonstriert die Anbindung der Agentenschicht an abstrakte Fähigkeiten modularer Quadrocopter. Der Grenzschichtforschung aus dem vorherigen Experiment folgend sind nicht nur vertikale Messungen von Interesse, sondern auch horizontale, um beispielsweise den Verlauf einer Grenzschicht zu verfolgen. Ein besonderes Werkzeug für die horizontale Messung ist die sogenannte faseroptische Temperaturmessung. Üblicherweise wird ein Lichtwellenleiterkabel (LWL-Kabel) direkt am Boden oder in Bodennähe gespannt, das in Kombination mit einem Auswertungsgerät (DTS) die Temperatur im Zentimeterbereich (je nach technischer Ausführung) am LWL-Kabel entlang messen kann [18]. Das Konzept des Experiments der synchronen Glasfasermessung ist, dass ein LWL-Kabel zwischen mehreren Quadrocoptern gespannt und das schwere DTS mit einem mobilen Bodenroboter transportiert wird. Durch eine Synchronisierung des heterogenen Ensembles soll nicht nur das Glasfaserkabel angehoben und getragen werden können, sondern auch eine koordinierte Bewegung des gesamten Ensembles möglich sein. Zwei zusätzliche, kalibrierte Temperatursensoren an den Quadrocoptern sollen die Messwerte der Glasfasermessung

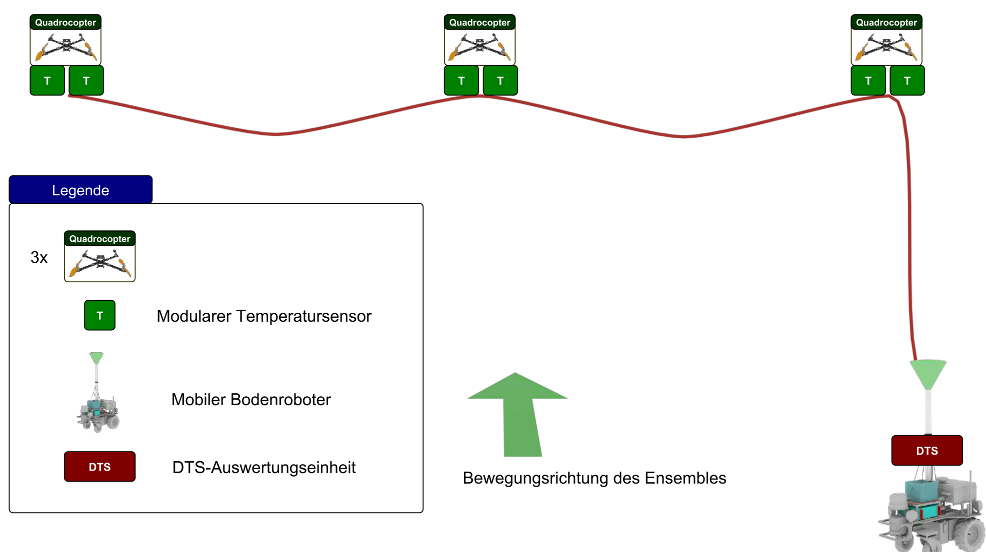

Abbildung 3.5. Dieses Experiment verwendet ein über mehrere Quadrocopter gespanntes Lichtwellenleiterkabel, welches an einem Auswertungsgerät an einem mobilen Bodenroboter montiert ist. Durch Synchronisierungsmechanismen kann sich das heterogene Ensemble zur Laufzeit in einer Achse bewegen. (Eigenwerk)

verifizieren. Obwohl es sich hierbei ebenfalls um eine **c**ontinously **O**bserve Mission handelt, bieten sich dennoch andere Herausforderungen. Darunter fällt in erster Linie die Synchronisierung eines heterogenen Schwarms aus Multipotenten Systemen und die deutlich aufwändigeren Messverfahren. Für eine einfachere Auswertung der Messwerte, als auch zur Überwachung der Glasfasermessungen müssen die Temperatursensoren kalibriert und die Rohwerte zur Laufzeit berichtigt werden. Im Kontext der Fallstudie sind die Faktoren der Synchronisierung, die Verteilungsaspekte, sowie die modularen Temperatursensoren dieses Experiments relevant.

3.3.3 Ausbreitungsmodellierung

Das Experiment der *Ausbreitungsmodellierung* gliedert sich in zwei Disziplinen der Fallstudie, wie in Abbildung 3.6 illustriert. Für die *lokale Ausbreitungsmodellierung* (1) von Gasen oder Aerosolen zur gezielten Prognose sind in situ Messungen nötig. Aerosole werden gemäß Duden als *die feinste Verteilung schwebender fester oder flüssiger Stoffe in Gasen, besonders in der Luft (z. B. Rauch, Nebel)* [72] definiert. Zwar verhalten sich Gase und Aerosole in der Ausbreitung nicht identisch, für die Experimente wird diese Annahme dennoch getroffen, da rechtliche Bedingungen den Einsatz von Gasen erschweren und die Visualisierungsmöglichkeit durch farbigen Rauch die Algorithmen besser illustriert. Lösungen für die lokale Ausbreitung von Gasen werden üblicherweise mit Installationen der Messgeräte an statischen Positionen durchgeführt [37].

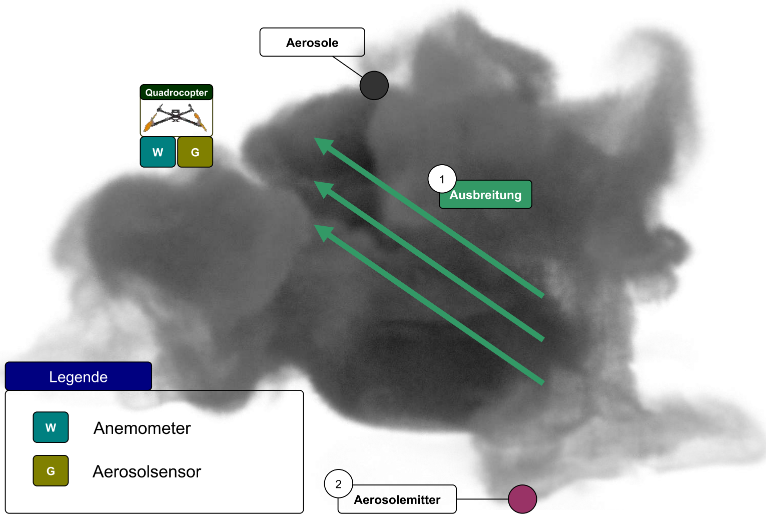

Abbildung 3.6. In diesem Experiment wird ein Partikelemitter eingesetzt, um eine Partikelwolke zu erzeugen. Um externe Einflüsse wie die Windstärke und -geschwindigkeit zu messen, werden ein Anemometer und ein Partikelsensor an den Quadrocopter angebracht. (Rendering und Simulation der Gaswolke sind Eigenwerke)

Der Herausforderung einer Ausbreitungsmodellierung mit mobilen Geräten und zeitabhängigen Messungen stellen sich Lilienthal et al. mit der Vorstellung ihres Kernel DM Algorithmus [177]. Dieser unterteilt das zu untersuchende Terrain in ein Raster mit einer Gaußverteilung für jedes Rastersegment anhand mehrerer Messwerte. Dieses Verfahren erfreut sich in der Domäne großer Beliebtheit und wird um die Varianz [175], den Einbezug von Windinformationen [240] oder die Anwendung im dreidimensionalen Raum [25] erweitert. Für die Berücksichtigung der Windinformation wird im vorliegenden Experiment ein Anemometer zur Bestimmung der Windrichtung und -stärke verwendet.

Die **Lokalisierung des Aerosolemitters** (2, s. Abbildung 3.6) stellt ebenfalls eine typische Problemstellung im wissenschaftlichen Raum dar. Eine sehr gute Übersicht liefert dabei die Dissertation von Lochmatter [179], in welcher zwei Algorithmenkategorien vorgestellt werden. Einerseits die Bio-inspirierten Algorithmen, wie etwa der auf der Beobachtung von Motten basierende *upwind surge* Algorithmus, und andererseits probabilistische Algorithmen, wie die sequenzielle *Monte-Carlo-Methode*, auch bekannt als *Partikel-Filter*. Durch die Kombination der Lokalisierung mit der Ausbreitungsmodellierung ist eine volle Abdeckung des Suchraums vorausgesetzt, den die Bio-inspirierten Ansätze nicht erfüllen. Die Monte-Carlo-Methode wird bereits zur Quellenlokalisierung für mobile Bodenroboter [175] oder Quadrocopter [211] angewendet.

Dieses Experiment stellt eine besondere Form der **S**earch, **c**ontinously **O**bserve Mission dar, da die Suche nach der Aerosolquelle mit den kontinuierlichen Messungen für die Ausbreitungsmodellierung zu kombinieren ist. Das Anemometer ist ein Beispiel für eine Klasse an Sensoren, die die Mechanismen von Semantic Plug & Play vor eine

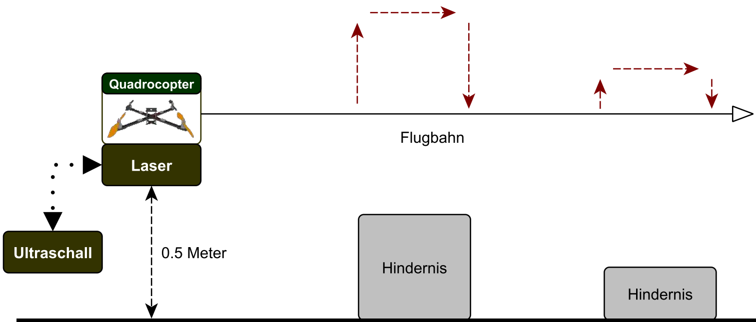

Abbildung 3.7. Ein Quadrocopter ist mit einem modularen Distanzsensor ausgestattet und soll über mehrere Hindernisse fliegen. Der Distanzsensor misst dabei den Abstand über Boden, der durch die Hindernisse variiert. (Eigenwerk)

Herausforderung stellen. Durch die Eigenschaften und Fähigkeiten eines Quadrocopters können in Kombination mit einem Positionssystem wie bspw. dem Global Positioning System (GPS) die Windstärke und -geschwindigkeit als virtuelle Fähigkeit eingesetzt werden, ohne externe Sensorik zu verwenden [322].

3.3.4 Der sensorbasierte Flug

Die Fallstudie setzt je nach erkanntem Gastyp unterschiedliche Gassensoren für die *Observierung der Gaswolkenausbreitung* ein. Die Algorithmik der Observierung soll jedoch nicht für jeden Gassensor angepasst werden, sondern auf der Sensorklasse aufbauen und gegebenenfalls Eigenschaften konkreter Gassensoren verwenden. Das Experiment untersucht die Eigenschaften und Auswirkungen zweier unterschiedlicher Distanzsensoren in einem sensorbasierten Flug, wie in Abbildung 3.7 abgebildet. Das Ziel ist es, den Abstand zwischen Boden und Quadrocopter konstant über die Flugstrecke zu halten. Die Position sowie die Größe der Hindernisse sind zur Designzeit nicht bekannt, können also nicht in die Planung einfließen. Ein Quadrocopter ist bereits ein komplexes System bestehend aus mehreren Sensoren und Aktuatoren, die über ein mathematisches Modell die Lage [183] regeln kann. In Kombination mit externen Positionssystemen wie bspw. einem *motion capture system* können auch im Innenbereich Positionen vorgegeben werden [101]. Da in diesem Experiment austauschbare Sensoren mit nicht bekannten Eigenschaften an das System angeschlossen werden sollen, wird eine Middleware eingesetzt, die eine vektorielle Positionssteuerung ermöglicht. Im ScORe Kontext handelt es sich um eine **S**earch, **Re**act Mission, in der Hindernisse gesucht werden und durch die Anpassung der Flugbahn reagiert wird. Eine Rekonfiguration ist zwar zur Laufzeit nicht nötig, aber die Austauschbarkeit der Distanzsensoren für die Untersuchung potentieller Auswirkungen zwischen Eigenschaften und Fähigkeiten durch Selbstbeschreibungsmechanismen ist essentiell.

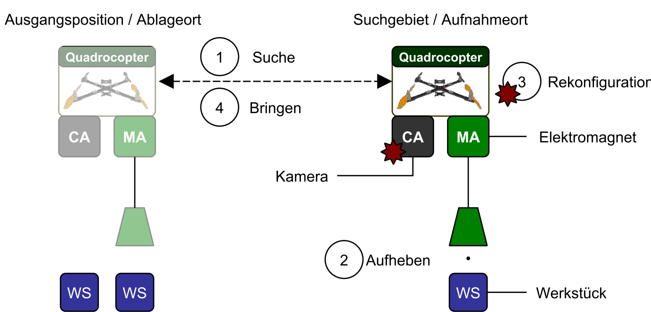

Abbildung 3.8. In diesem Experiment soll ein Quadrocopter mit einer Kamera ein Werkstück suchen (1). Nach erfolgreicher Suche wird versucht, über einen Elektromagnet das Werkstück aufzuheben (2). Durch das zusätzliche Gewicht muss die Kamera demontiert werden (3), bevor das Werkstück zum Ausgangspunkt transportiert werden kann (4). (Eigenwerk)

3.3.5 Pick & Place

Die Flexibilität der Fallstudie basiert unter anderem auf der Rekonfiguration des Ensembles. Eine neue Zusammensetzung der modularen Hardwareelemente resultiert jedoch nicht zwangsweise in den gewünschten Fähigkeiten. So können die montierten Sensoren zu einer Überschreitung der maximalen Tragfähigkeit des Quadrocopters führen und die Fähigkeit *Fliege zu Position* geht verloren. Das Experiment *Pick & Place* fokussiert sich auf kombinierte Fähigkeiten und die Rekonfiguration zur Laufzeit. Zwei austauschbare Module für eine spezielle Kamera und für einen Elektromagnet werden an einen Quadrocopter montiert, wie in Abbildung 3.8 abgebildet. Die Kamera kann durch eine anwendungsspezifische integrierte Schaltung (ASIC) Farbkennungen zur Laufzeit erkennen und die relativen Positionsdaten bereitstellen [2]. Das motion capturing System des vorhergehenden Experiments kommt auch hier wieder für die Positionserkennung des Quadrocopters zum Einsatz. In der ersten Phase wird mithilfe der Kamera das Werkstück gesucht. Wird das Werkstück gefunden, so versucht der Quadrocopter das Werkstück aufzuheben. Beim Anheben wird überprüft, ob die Gesamtlast die Tragfähigkeit des Quadrocopters überschreitet. Dafür werden die Gewichtsdaten der Kamera und des Elektromagneten berücksichtigt, sowie die Parametrisierung der PID-Regler, die eine Auswirkung auf die Maximallast haben. Durch die Belastung der Motoren in Kombination mit der Positionssteuerung kann die kombinierte Fähigkeit *Gewichtsabschätzungen* vorgenommen werden [40]. Durch die Demontage der Kamera zur Laufzeit und dem dadurch eingesparten Gewicht kann das Werkstück angehoben und transportiert werden. Im Hinblick auf ScORe Missionen handelt es sich in diesem Experiment um eine **S**earch, **Re**act Mission, in welcher das Werkstück gesucht und durch Rekonfiguration des Gesamtsystems transportiert wird. Für Semantic Plug & Play stehen die Gewichtseigenschaften der Selbstbeschreibungen im Fokus und die Einschränkungen der Fähigkeiten, falls die Summe der Gewichte aller Module sowie Gewichte außerhalb des Systems die Flugfähigkeit beeinträchtigt. Die Rekonfiguration hingegen erfordert austauschbare Module, die zur Laufzeit ausgewechselt werden können.

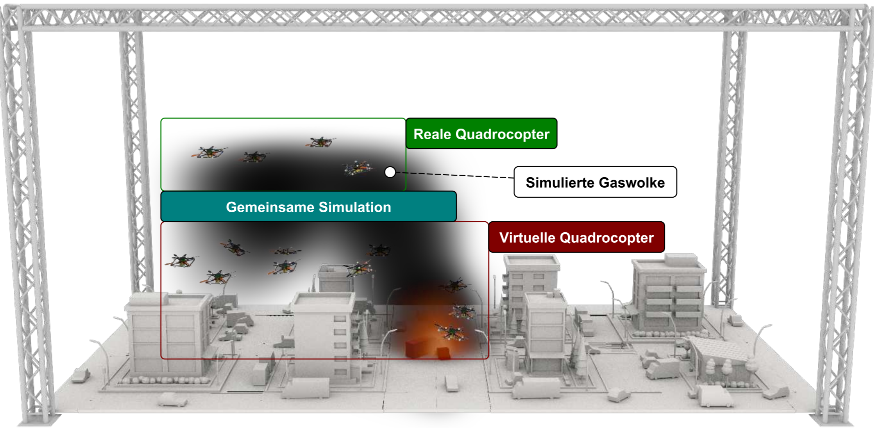

Abbildung 3.9. Dieses Experiment hat das Ziel, reale und virtuelle Repräsentation in einer Simulationsumgebung zu berücksichtigen. So sollen Katastrophenszenarien wie ein Chemieunfall simuliert und schrittweise mit realer Hardware getestet werden können. (Eigenwerk)

3.3.6 GoLive

Die bereits vorgestellten Experimente behandeln unterschiedliche Aspekte der Fallstudie und tragen zur Entwicklung der Gesamtlösung bei. Die Realisierung der Fallstudie bildet jedoch eine erhebliche Hürde, da eine Vielzahl realer Hardwareelemente eingesetzt werden soll. Um diese Hürde zu überwinden wird in diesem Experiment das Konzept einer schrittweisen Überführung von der Simulation in die Realität demonstriert.

Die Visualisierung digitaler Zwillinge in der Produktion wird bereits in verschiedenen Architekturen vorgestellt [270] oder anhand der Microsoft Hololens [328] realisiert. Bei realen Objekten soll die AR-Komponente primär bei der Steuerung, Konfiguration und Überwachung Vorteile bringen [237, 328]. Das digitale Abbild hingegen wird in der Produktionsplanung und für Simulationsschleifen verwendet [245]. Eine Simulationsschleife ist, dass auf eine neue Gegebenheit eine Simulation der geplanten Aktion und erst danach die Aktion auf der realen Hardware ausgeführt wird, was wiederum eine neue Gegebenheit schafft. Einen ähnlichen Ansatz verfolgen auch *Hardware in the Loop Simulationen* (HIL), allerdings werden bspw. Sensordaten simuliert und an die Steuerung einer Hardware übermittelt [170]. Die Ergebnisse der Hardware, bpsw. eines Mikrocontrollers, werden mit den Erwartungswerten abgeglichen und wiederum in die Simulation gespeist.

Das Experiment *GoLive* stellt nun ein neues Konzept der *schrittweisen Überführung* von der Simulation zur Realität mit einer dynamischen Simulationsumgebung vor, die zur Laufzeit verändert werden kann. Die in Abbildung 3.9 skizzierte gemeinsame Simulation bildet eine Simulationsschleife für virtuelle Quadrocopter, die ebenfalls

mit HIL Mechanismen erweitert werden und mit realen Quadrocoptern kooperieren können. So soll bspw. durch einfache Kollisionserkennungs- und -vermeidungsstrategien innerhalb der Simulation nicht zwischen realen und virtuellen Elementen unterschieden werden. Ein virtueller Quadrocopter kann die Trajektorie eines realen Quadrocopters beeinflussen und vice versa. Eine besondere Herausforderung für die Simulation bildet die Modularität von Semantic Plug & Play: So kann ein realer Quadrocopter einen virtuellen Sensor montieren, ein virtueller Quadrocopter jedoch keinen realen Sensor. Die Simulation soll neben der Rekonfiguration bekannter Elemente auch zur Laufzeit erweiterbar sein, so dass neue Objekte hinzugefügt werden können, wie etwa eine neue Gefahrenquelle, um die Vielfalt der Szenarien der Fallstudie abzudecken. Durch diese Dynamik können alle Varianten der **S**earch, **c**ontinously **O**bserve, **Re**act Missionen abgebildet und die umfangreiche Fallstudie mit Multipotenten Systemen schrittweise realisiert werden.

Zusammenfassung. Eine Selbstbeschreibung modularer Robotersysteme soll zu einem Paradigmenwechsel in der Programmierung und zu einer Realisierung komplexer Robotersysteme führen. Dabei müssen etablierte Techniken und Methodologien nicht etwa neu erfunden, sondern vielmehr durch Erweiterungen und gezielte Wechselwirkung kombiniert werden. Für die Nachvollziehbarkeit der kombinatorischen Konzepte wird in diesem Kapitel ein Überblick über die Grundlagen gewährt.

Grundlagen für die Selbstbeschreibung

4.1 Semantische Beschreibung von Eigenschaften . . . 38
4.1.1 Verteilung (URI) . . . 40
4.1.2 Syntax (XML, JSON) . . . 41
4.1.3 Graphen (RDF) . . . 42
4.1.4 Schemata (RDFS) . . . 43
4.1.5 Regelwerke (Ontologien) . . . 44
4.1.6 Abfrage (SPARQL) . . . 46
4.1.7 Linked (Open) Data . . . 47
4.2 Technische Grundlagen für Fähigkeiten . . . 48
4.2.1 Experiment: Der Mobile Messturm . . . 49
4.2.2 Schnittstellen und Controller . . . 50
4.2.3 Softwarearchitekturen verteilter Komponenten . . . 53
4.2.4 Modellierung und Simulation im Robot Operating System (ROS) . . . 56
4.2.5 Versionierung und Verteilung mit GIT . . . 58
4.2.6 Virtualisierung und Container . . . 59

Für die Realisierung einer Selbstbeschreibung modularer Robotersysteme mit Plug & Play Mechanismen können Techniken aus den Domänen der Webentwicklung, Robotik und der Softwaretechnik eingesetzt werden. In diesem Kapitel werden die Grundlagen aus diesen Domänen, wie in Abbildung 4.1 illustriert, vorgestellt. Für eine erweiterbare semantische Beschreibung der Eigenschaften und Fähigkeiten werden Techniken des Semantic Web vorgestellt und mit der Methodologie von Tim Berners Lee, den Linked (Open) Data Prinzipien, abgerundet. Die Realisierung von Fähigkeiten wird mit einem Experiment und einer hardwarenahen Einführung physischer Schnittstellen und Hardwarecontroller eingeleitet, um die Basis der Problemstellungen herzustellen. Aus dem Bereich der Softwaretechnik wird anschließend ein kurzer Überblick über ausgewählte Architekturen vorgestellt und anhand des Robot Operating System (ROS) konkretisiert. Abgeschlossen werden die Grundlagen mit einer Methodologie für das Bereitstellen von

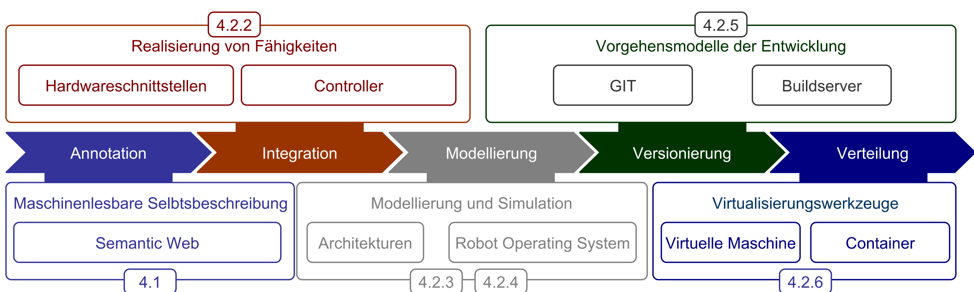

Abbildung 4.1. Übersicht der behandelten Themen in diesem Kapitel mit entsprechenden Unterkategorien und deren jeweiligem Schwerpunkt. So steht beispielsweise im Unterkapitel der Versionierung das Thema *Vorgehensmodelle der Entwicklung* im Fokus. (Eigenwerk)

Softwareartefakten in gängigen Versionierungsumgebungen und der containerbasierten Ausführungsumgebung mit Docker.

4.1 Semantische Beschreibung von Eigenschaften

Die Entwicklung des World Wide Webs (WWW) kann in 3 Entwicklungsstufen unterteilt werden, die in Abbildung 4.2 illustriert sind. Das Web 1.0 dient der Bereitstellung von Daten, also statischen Inhalten, die über das Internet abgerufen werden. Die Entwicklung des Web 2.0 hingegen erlaubt dem Nutzer, an Inhalten mitzuwirken. Beispiele hierfür bilden Foren, freie Enzyklopädien und soziale Plattformen. Obwohl die Ursprünge der sozialen Netzwerke bereits 1972 in der Berkely Community Memory oder Plato IV zu finden sind [77], so gelten Entwicklungen wie das Rich Site Summary (RSS) 1999 oder Wikipedia 2001 als Grundstein für das Web 2.0 [4]. Der Begriff selbst wurde jedoch erst ab 2004 von O'Reilly geprägt [143]. Bis zur ersten Erwähnung wurde stattdessen häufig der Begriff *social web* verwendet. Das Web 3.0 hingegen widmet sich nicht der Nutzerinteraktion sondern vielmehr der Semantik von Daten und wird nach John Markoff 2006 als Zusammensetzung der Konzepte des Web 2.0 mit den Konzepten des **Semantic Web** definiert [190]. Das Web3 hingegen wird häufig irreführend synonym für das Web 3.0 verwendet und soll auf dezentralen Blockchaintechnologien mit Fokus auf Non Fungible Tokens (NFT) basieren [299]. Fehlende technische Realisierungen und schwammige Konzepte führen jedoch zu harschen Kritiken, wie beispielsweise von Elon Musk, der es auf Twitter mehr als Schlagwort denn Realität einstuft [180]. Das Semantic Web hingegen wurde bereits 2001 von Tim Berners Lee vorgestellt und umfasst einen umfangreichen Technologiestack [24]. Das Ziel ist die Maschinenlesbarkeit der Information, die sich in den Freitexten der Nutzerinhalte befindet. Die Zusammenhänge der Informationen werden dafür in Graphen ausgedrückt, mit einem Schema vereinheitlicht und durch Regeln in der Verwendung eingeschränkt. Typische Themengebiete des Semantic Web sind unter anderem semantische Annotationen von Enzyklopädien wie Wikipedia in dem Projekt *DBPedia* [200] oder für die Modellierung sozialer Netzwerke mit *FOAF* [108].

Beschreibungen von Sensoren und Aktuatoren finden üblicherweise im Rahmen statischer Webseiten oder anhand online gestellter Dokumente statt. Als Beispiel wird der

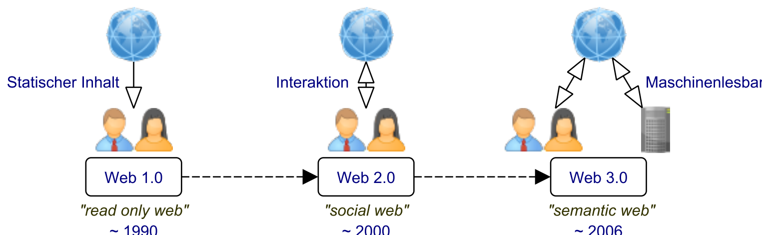

Abbildung 4.2. Entwicklungsstufen des Webs mit einer groben Zeiteinteilung durch maßgebliche Entwicklungen. Das Datum des Web 1.0 bezieht sich auf den Vorschlag von Tim Berners Lee 1989 [19] und der tatsächlichen Implementierung. Die jeweiligen Stufen des Webs ergänzen vorhergehende und sind weiterhin in Verwendung. (Eigenwerk)

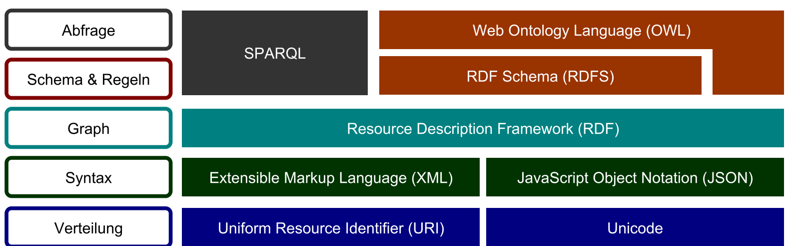

Abbildung 4.3. Ausschnitt der Schichten des Semantic Web nach Tim Berners Lee, angelehnt an die vier Versionen der Architektur [118]. Additiv befindet sich auf der Syntax Ebene ebenfalls JSON, das von JSON-LD 1.1 verwendet [142] und seit 2020 von der W3C empfohlen wird [301]. (Eigenwerk in Anlehnung an [118])

Sensor *SHT75* des Herstellers Sensirion angeführt, der sowohl die Temperatur als auch die Luftfeuchte messen kann. Auf der Webseite des Herstellers [271] finden sich die Spezifikationen und unterschiedliche Dokumente, wie unter anderem das Datenblatt, das Kalibrierungszertifikat und Anwendungshinweise. Die darin gespeicherten Informationen werden für die Integration des Sensors und der Interpretation der Messwerte benötigt, aber auch Portale wie Mouser oder RS-Elektronik verwenden ein Subset dieser Informationen, um den Sensor anhand der Eigenschaften zu kategorisieren. Mangels Maschinenlesbarkeit der von Sensirion bereitgestellten Eigenschaften unterscheiden sich die Subsets teils gravierend. So werden in den Angaben des Herstellers eine typische Temperaturungenauigkeit von 0.3°C [271] angegeben, bei Mouser sind es 0.4°C [78] und bei RS-Elektronik wird keine Angabe gemacht [49]. Folgend den Grundlagen des Semantic Web sollen nun die Spezifikationen dieses Sensors sukzessive semantisch annotiert und die Vorteile dargestellt werden, um das Konzept des Semantic Web zu verdeutlichen. Dabei werden, wie in Abbildung 4.3 skizziert, die Spezifikationen in **Uniform Resource Identifier (URI)** oder Literale (Unicode) unterteilt. Anhand geeigneter Syntax (XML, JSON) werden die URIs in einen Graphen mit Hilfe des **Resource Description Frameworks (RDF)** überführt, durch das **RDF-Schema (RDFS)** vereinheitlicht und mit der **Web Ontology Language (OWL)** eingeschränkt. Der daraus resultierende Graph kann

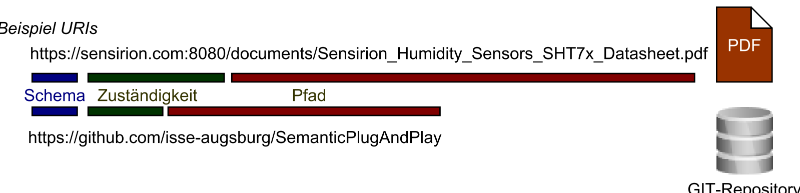

Abbildung 4.4. Aufbau einer URI nach RFC3986 [20] mit den drei wichtigsten Fragmenten (Schema, Zuständigkeit, Pfad). Das obere Beispiel steht für eine PDF-Datei und das untere für ein GIT-Repository. (Eigenwerk)

mit **SPARQL** abgerufen werden. Dieses Vorgehen bildet jedoch kein absolutes Novum, da mit den Technologien des Semantic Web bereits domänenrelevante Eigenschaften von Sensoren für meteorologische Messungen [35, 50] mit der *Semantic Sensor Network Ontology* (SSN) beschrieben wurden.

4.1.1 Verteilung (URI)

Für die Verteilung, insbesondere der Bereitstellung von Informationen oder generell Ressourcen, werden folgend knapp die üblichen Nomenklaturen anhand des Sensorbeispiels vorgestellt. Zuerst werden die Verteilungsmechanismen, wie in Abbildung 4.3, auf der untersten Ebene vorgestellt.

Ein **Uniform Resource Identifier (URI)** nach RFC3986 ist ein eindeutiger Bezeichner für eine Ressource in Form einer Zeichenfolge, unabhängig davon, ob die Ressource physischer oder abstrakter Natur ist. Demnach kann hinter einer URI sowohl eine Datei als auch eine Person stehen. Objekte ohne Informationsgehalt werden als Non-Information Resources tituliert [22]. In Abbildung 4.4 wird der Aufbau einer URI nach RFC3986 [20], wie sie aktuell benutzt wird, klassifiziert. Das Schema definiert den Kontext und bezeichnet so den Typ der URI, dabei wird in den Beispielen lediglich das **HyperText Transfer Protocol Secure (HTTPs)** verwendet, Beispiele für Alternativen sind das File Transfer Protocol (FTP) oder Telnet. Das HTTPs nach RFC2660 [244] erweitert das HyperText Transfer Protocol (HTTP), welches als Standardinstrument für das heutige World Wide Web (WWW) gilt und hauptsächlich für die Kommunikation zwischen Browser und Server verwendet wird. Die Zuständigkeit hängt von dem verwendeten Schema ab und ist im Fall des Beispiels der Domainname mit optionaler Portangabe z.B. (8080). Viele Schemata verwenden Standardports, wie etwa HTTPs:443, HTTP:80, FTP:20,21 und Telnet:23. Diese müssen nicht angegeben werden, sofern sie nicht bewusst geändert sind. Der Pfad dient der Identifizierung der Ressource, in dem Beispiel aus Abbildung 4.4 eine PDF-Datei mit den Spezifikationen des Sensors oder ein GIT-Repository.

Der **Unicode-Standard** des Unicode-Konsortiums legt fest, wie Schrift elektronisch gespeichert wird [147]. Im Rahmen des Semantic Webs werden hier vor allem einzelne Wörter betrachtet, die als sogenannte Literale stehen, Unicode gibt dabei den Rahmen der Interpretation vor. Die Interaktionsart, wie in Abbildung 4.2 illustriert, bietet noch keine Veränderung der üblichen Praxis des WEB 1.0, das ebenso HTTP(s) URIs verwendet

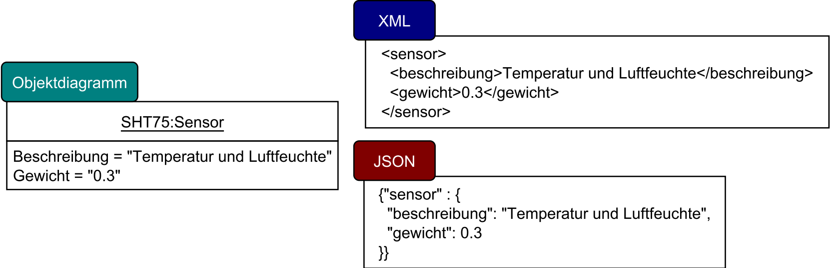

Abbildung 4.5. Objektdiagramm mit einer Instanz der Klasse Sensor und den Attributen Typ und Genauigkeit. Daneben ist einmal die Serialisierung der Instanz mittels XML und JSON. (Eigenwerk)

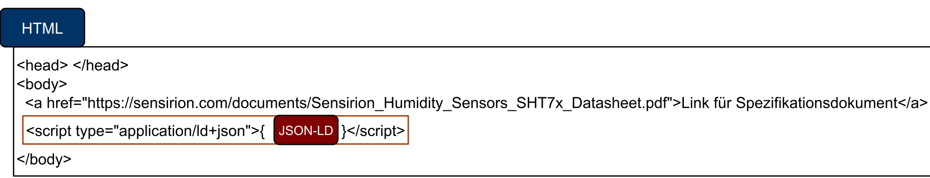

Abbildung 4.6. Einfache HTML-Seite mit einem Link für die PDF-Datei mit den Spezifikationen des SHT75 Sensors. Über das *script* Tag kann eine JSON-LD Beschreibung eingebettet werden. (Eigenwerk)

und die Spezifikationen innerhalb einer PDF-Datei verbirgt. Als nächsten Schritt für die semantische Annotation werden Strukturen betrachtet, mit denen ein strukturierter Austausch von Daten stattfinden kann.

4.1.2 Syntax (XML, JSON)

Im Semantic Web müssen für semantische Annotationen bereits existierende Ressourcen nicht verändert werden. So muss die Spezifikation in Form einer PDF-Datei nicht gelöscht werden, sondern kann in die semantische Annotation mit aufgenommen werden. Die semantische Annotation findet anhand einer Syntax etablierter Standards statt, wie die **Extensible Markup Language (XML)** oder die **JavaScript Object Notation (JSON)**, die ebenfalls in objektorientierten Programmiersprachen als Austauschformat serialisierter Objekte dienen [185]. In Abbildung 4.5 wird eine Beispiel skizziert, in welchem ein Objekt sowohl in XML als auch in JSON serialisiert ist. Beide Varianten haben Vor- und Nachteile bezüglich der Performance, dem Overhead und der Lesbarkeit [117, 185]. Im Rahmen des Semantic Webs werden diese Strukturen angewendet, um semantische Informationen bspw. in übliche **Hypertext Markup Language (HTML)** Webseiten einzubetten, wie in Abbildung 4.6 illustriert. Dabei stehen mehrere Optionen offen, welche auch über die bereits vorgestellte Syntax von XML und JSON hinausgehen, wie bspw. HDT, N-Triples oder Turtle [258]. RDF/XML wird üblicherweise als Referenz in der Standardliteratur verwendet [23, 127], aufgrund der besseren Lesbarkeit werden in der vorliegenden Dissertation die Beispiele in JSON-LD aufgeführt.

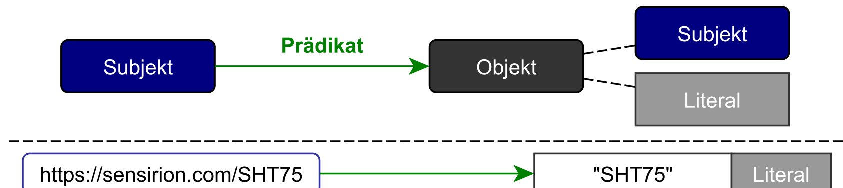

Abbildung 4.7. Darstellung des Aufbaus eines RDF-Triples in einer gerichteten Graphendarstellung. Darunter ist ein Fragment aus dem JSON-LD-Beispiel exemplarisch aufgeführt. (Eigenwerk)

JSON-LD ist eine spezielle Serialisierung für **L**inked **D**ata[1] unter Verwendung der JSON Syntax. So ist nach RFC 4627 jeder JSON-LD Text ein gültiger JSON-Text [53]. Als Beispiel sei folgend der Code für die Beschreibung des Sensorobjektes aus Abbildung 4.5 skizziert.

```
1  {
2    "@context": {
3      "gr": "http://purl.org/goodrelations/v1#",
4      "pto": "http://www.productontology.org/id/"
5      }
6    },
7    "gr:name": "SHT75",
8    "gr:description": "Temperature und Luftfeuchte",
9    "pto:weight": 0.3
10   }
11 }
```

Dieses Beispiel soll lediglich die Syntax von JSON-LD vorstellen, für eine Interpretation der Elemente werden im nächsten Abschnitt die technischen Grundlagen erläutert.

4.1.3 Graphen (RDF)

Das **Resource Description Framework (RDF)** ist ein Modell zur Repräsentation einzelner Aussagen und wird wie JSON-LD von der W3C in der Version 1.1 empfohlen [303]. Eine Aussage wird in RDF-Tripel genannt und besteht aus drei Teilen (Subjekt, Prädikat, Objekt) wie in Abbildung 4.7 illustriert. Analog zum Satzbau der deutschen Sprache wird auch in RDF die Reihenfolge der einzelnen Elemente strikt geordnet. Das **Subjekt** beschreibt eine URI, über die etwas gesagt wird, wie beispielsweise die URI des SHT75 Sensors. Das **Prädikat** formuliert die Aussage über das Subjekt anhand einer URI. Im Beispiel wird aus einem Dictionary für den Verkauf von Gegenständen eine URI ausgewählt. Das **Objekt** kann entweder ein weiteres Subjekt oder ein Literal in Unicode wie

[1]Prinzipen für die Erstellung von RDF-Graphen, siehe Abschnitt 4.1.7

exemplarisch *SHT75* referenzieren und repräsentiert den Inhalt der Aussage [124]. Diese Art der Verknüpfungen kann ebenfalls in eine Graphenform überführt werden, indem Subjekte und Objekte als Knoten und die Prädikate als Kanten interpretiert werden. Neben der Einbettung in HTML-Seiten, können die Graphen bestehend aus RDF-Trippeln auch in Datenbanken gespeichert und über Tools verwaltet werden, wie bspw. 3store, Virtuoso, Apache Jena und RDF-3X, (um nur ein paar zu nennen) [212].

4.1.4 Schemata (RDFS)

Damit in RDF ein gemeinsamer Konsens für die Verknüpfung von Trippeln gewährleistet werden kann, muss eine gemeinsame Basis für Prädikate geschaffen werden. Die deutsche Sprache allein reicht für eine eindeutige Benennung dabei nicht aus, da ein Prädikat (im Sinne des RDF) meist in kontextuellem Zusammenhang steht. Bereits das Tripel $\{(https://sensirion.com/SHT75), (hatdenNamen), (SHT75)\}$ bietet einen breiten Spielraum an unterschiedlichen Variationen. Durch den Kontext innerhalb der vorliegenden Dissertation wird das Prädikat $(hatdenNamen)$ als Name des Produkts interpretiert, also dem Sensor von Sensirion und nicht als der Name der Webseite. Weiterhin sind für das Prädikat auch Varianten wie $(traegtdenNamen)$ oder $(hatNamen)$ denkbar. So gibt RDF nur die Syntax von Tripeln vor, doch für die Interpretation wird ein gemeinsames **Vokabular** benötigt. Das Vokabular hingegen bezieht sich immer auf einen Kontext. So gibt es Vokabularien für die Beschreibung elektronischer Ressourcen wie bspw. *Dublin Core* oder für den Verkauf von Produkten, wie *GoodRelations*, das Vokabular aus dem Beispiel in Abbildung 4.7.

Das **Resource Description Framework Schema (RDFS)** ist eine Erweiterung für RDF und wird ebenfalls von der W3C empfohlen [304]. RDF-Schema erweitert RDF, indem es bestimmten Ressourcen eine extern spezifizierte Semantik zuweist, z. B. *rdfs:subclassOf, rdfs:Class*. RDFS bietet Konzepte zur Beschreibung von Klassen, Ressourcen und Properties, sowie deren Zusammenhänge.

Neben RDFS existieren weitere Beschreibungssprachen wie bspw. OIL mit ergänzenden logischen Operationen (and, or, not) [33] oder die weit verbreitete **Web Ontology Language (OWL)** [197], einer Weiterentwicklung von OIL (über DAML+OIL). Die offizielle Abkürzung OWL beruht auf einem Vorschlag von Hendler [302], einem der Mitbegründer des Semantic Web neben Tim-Berners Lee [23]. OWL ist ebenfalls eine Empfehlung des W3C Konsortiums [197] und umfasst eine Spezifikation, um Ontologien anhand einer formalen Beschreibungssprache zu erstellen. Zusätzlich zu RDF und RDF-Schema werden weitere Sprachkonstrukte eingeführt, die es erlauben, Ausdrücke ähnlich der Prädikatenlogik zu formulieren, um die Ausdrucksmächtigkeit zu steigern oder einzuschränken. Dabei verwendet OWL das Konzept der Open World Assumption, also der Annahme, dass etwas existieren kann, bis es widerlegt wird. Dies wird vor allem relevant für sogenannte **Reasoner**, die logische Konsequenzen aus den mit OWL aufgestellten Ontologien generieren können. Da in der vorliegenden Dissertation die Verwendung von Reasonern nicht im Fokus steht, aber einen logischen nächsten Schritt darstellt, sei an dieser Stelle auf die Reasoner Pellet [275] und HermiT [273] verwiesen, die sich auch in unterschiedlichen Evaluationen [103, 224] behaupten konnten.

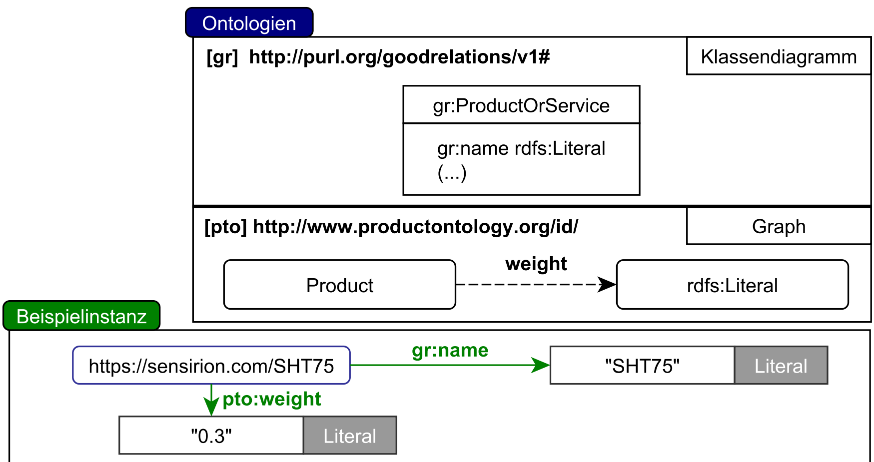

Abbildung 4.8. Beispiel aus Abschnitt 4.1.2 mit Ausschnitten der verwendeten Ontologien. Prinzipiell werden zwei Visualisierungen verwendet, entweder als Klassendiagramm oder in Form eines gerichteten Graphen. Die Beispielinstanz verwendet die Ontologien anhand der in JSON-LD definierten Kürzel gr und pto. (Eigenwerk)

4.1.5 Regelwerke (Ontologien)

Der Begriff *Ontologie* hat seinen Ursprung in der Philosophie und die Begriffsdefinition ist innerhalb der Informatik ähnlich vielfältig ausgeprägt. So beschreibt bspw. ein Beitrag in Dialogform die Sichtweisen beider Domänen mit dem Resultat, dass sich mehr als zehn wesentliche Definitionen im Bereich der Informatik gefunden haben, die sich teilweise deutlich unterscheiden [38]. Eine weitgefasste Definition nach Gruber et al. lautet:

> *„Eine Ontologie ist eine formale, explizite Spezifikation einer gemeinsamen Konzeptualisierung.“* (Übersetzung) [112]

Mit RDF wurde bereits eine Syntax für Tripel-Notationen vorgestellt, und mit Vokabularien ein gemeinsamer Konsens geschaffen. RDFS und OWL hingegen bieten die Basis, Vokabularien mit Regeln zu ergänzen. Da eine Ontologie ein Vokabular bedingt, wird in vielen Fällen der Begriff *Vokabular* und *Ontologie* symbiotisch verwendet. So bezeichnet sich die Ontologie GoodRelations (gr) auf ihrer Internetpräsenz als *„standardisiertes Vokabular (auch bekannt als Schema, Data Dictionary oder Ontologie)“* (Übersetzung) [104]. Folgend werden die Grundkonstrukte einer Ontologie nach Rehbein et al. [241] zusammengefasst, die sich teilweise in RDFS und vollständig in OWL wiederfinden.

- **Klassen (classes)** dienen der Abstraktion von Dingen, die bestimmte Eigenschaften besitzen. Durch eine Vererbung können Hierarchien gebildet werden, wobei die Eigenschaften der Oberklasse(n) vererbt werden. Beispiele für Klassen sind *Mann*, *Frau*, die von der Klasse *Person* erben.

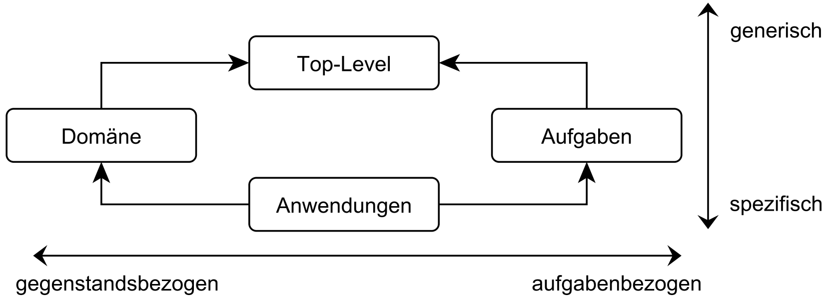

Abbildung 4.9. Klassifizierung von Ontologien nach Rehbein et. al. [241] in Abhängigkeit des Bezuges (x-Achse) und der Abstraktion (y-Achse). Die Assoziationen zwischen den Klassifizierungen stehen für eine strukturelle Abhängigkeit. (Eigenwerk in Anlehnung an [241])

- **Attribute oder Eigenschaften (attributes, properties)** beschreiben die Objekte der Ontologie. Dabei können sowohl Literale, Instanzen oder Klassen referenziert werden. Eine Referenz zu einer Klasse wird auch Relation genannt. Ein Beispiel bildet das Attribut *Vorname* der Klasse *Person.*
- **Bedingungen (constraints)** werden benutzt, um eine logische Konsistenz zu wahren, wie bspw. dass eine Instanz der Klasse *Person* nur einer seiner beiden Unterklassen *tote Person* oder *lebendige Person* angehören kann (Schrödingers Paradoxon ist damit nicht abbildbar) [241].
- **Aussagen (propositions)** umfassen Klasseneigenschaften, Relationen und Bedingungen und bilden das logische Regelwerk.
- **Instanzen oder Individuen (instances, individuals)** sind letztlich die konkreten Objekte und gehören in der Regel einer oder mehreren Klassen an. So ist im Beispiel das Attribut *Vorname* der Klasse *Person* mit der Instanz *„Ludwig“* assoziiert.

Das in Abschnitt 4.1.2 vorgestellte Beispiel des SHT75 Sensors durch eine JSON-LD Datei referenziert zwei Ontologien: GoodRelations (gr) [104] und die Product Types Ontology (pto) [126], wie in den Prädikaten der Abbildung 4.8 illustriert. Das abgebildete Klassendiagramm definiert *gr:ProductOrService* als Klasse und schränkt das Attribut *gr:name* durch die implizite Bedingung auf ein RDFS:Literal ein. Die Visualisierung anhand eines Graphen funktioniert hier analog, so ist das Produkt ohne Angabe einer Klasse und über das Attribut *weight* auf ein RDFS:Literal eingeschränkt. Eine einheitliche Visualisierung ist leider nicht definiert und so existieren neben Graphen und Klassendiagrammen auch zahlreiche Alternativen [139]. Neben der Vielfalt an Visualisierungen existieren auch zahlreiche Ontologien für unterschiedlichste Einsatzzwecke. Für eine thematische Zuordnung unterteilt Gomez et al. Ontologien in sieben Kategorien [102], Rehbein et al. hingegen verwendet ein Modell aus vier Gruppen und skizziert auch deren Abhängigkeiten [241], wie in Abbildung 4.9 dargestellt. Die **Top-Level**

Ontologie umfasst allgemeine Begriffe und kann unabhängig von einem spezifischen Gegenstands- oder Aufgabenbereich angewendet werden. Für die **Domäne** werden die Top-Level-Ontologien spezialisiert und auf Gegenstände angewendet, wie bspw. in der pto Ontologie, die Produkte für den Verkauf spezifiziert. Eine für **Aufgaben** ausgelegte Ontologie fokussiert die tatsächliche Verwendung, wie bspw. die *gr* Ontologie für den Verkauf von Waren oder die Semantic Sensor Network Ontology (SSN) [300] für Sensormessungen. Konkrete **Anwendungen** hingegen kombinieren die Aufgaben und Domänen und sind hochgradig speziell. Dieser Unterteilung folgend können und sollen neue Ontologien mit bereits existierenden verknüpft und Duplikate vermieden werden.

4.1.6 Abfrage (SPARQL)

Wie bereits in Abschnitt 4.1.3 eingeführt, können RDF-Tripel auf unterschiedliche Art und Weise gespeichert und über Tools verwaltet werden. Diese Tools bieten in der Regel auch eine Anfrage-Engine für die graphenbasierte Abfragesprache ***S****PARQL* ***P****rotocol* ***A****nd* ***R****DF* ***Q****uery* ***L****anguage* **(SPARQL)** an, mit welcher SQL ähnliche Abfragen von RDF-Graphen möglich sind. Ebenso wie für RDF, RDFS und OWL gibt es für SPARQL in der Version 1.1 eine Empfehlung des W3C Konsortiums [305]. SPARQL bietet ein XML-basiertes Ausgabeformat und eine Kommunikation über HTTP oder alternativ einen lokalen Endpoint für lokale RDF-Graphen an.

```
1  PREFIX ssn: <http://www.w3.org/ns/ssn/>
2  SELECT ?produkt
3  WHERE
4  {
5      ?produkt name              "SHT75".
6               ssn:hasSubsystem  TemperatureSensor.
7               ssn:hasSubsystem  HumiditySensor.
8  }
```

Das Beispiel bezieht sich auf die Instanz aus Abbildung 4.8 und bildet eine Anfrage für alle Produkte mit dem Namen SHT75, die nach der SSN Spezifikation [300] sowohl einen Temperatur- als auch einen Luftfeuchtesensor besitzen. Im Gegensatz zu SQL werden in der *Where* Klausel die Anfragen in Tripeln formuliert, wobei nicht alle Informationen in einer Datenbank liegen müssen. SPARQL erlaubt die übergreifende Suche über verschiedene Endpoints, wie etwa GeoNames, einer Ontologie mit über 10 Millionen Toponymen [241]. Diese Art der Vernetzung stellt die Basis für Semantische Netze her, die in ihrer größten Ausprägung mit 1301 Datensätzen und 16283 Links die Linked Open Data (LOD) Cloud bildet (Stand 05.2020) [196]. Grundvoraussetzung für den Beitritt zur LOD-Cloud ist die Einhaltung der Methodologie zu Linked Open Data.

4.1.7 Linked (Open) Data

Methodologien für die semantische Annotation eigener Ressourcen und die Verknüpfung zu semantischen Netzen existieren in unterschiedlichsten Variationen. So finden sich in der Domäne der *Open Government Data* über 16 Methodologien, verabschiedet vom W3C Konsortium, für die Verwaltung öffentlicher Dokumente der Vereinten Nationen bis hin zu speziellen Methodologien, die im Weißen Haus verwendet werden [162]. Die ursprüngliche Methodologie mit vier Prinzipien hat Tim Berners Lee bereits 2006 aufgestellt [21].

- *Uniform Resource Identifiers (URIs) should be used to name and identify individual things.*
- *HTTP URIs should be used to allow these things to be looked up, interpreted, and subsequently "dereferenced".*
- *Useful information about what a name identifies should be provided through open standards such as RDF, SPARQL, etc.*
- *When publishing data on the Web, other things should be referred to using their HTTP URI-based names.*

Diese nicht ganz eindeutigen Prinzipien dienten drei Jahre als Grundstock für den Aufbau der LOD Cloud. Bereits 2009, als die LOD Cloud noch unter 100 Datensätze umfasste [196], revidierte Tim Berners Lee die Prinzipien und fasste diese während einer TED Konferenz in drei neue Prinzipien zusammen [21]:

- *All conceptual things should have a name starting with HTTP.*
- *Looking up an HTTP name should return useful data about the thing in question in a standard format.*
- *Anything else that that same thing has a relationship with through its data should also be given a name beginning with HTTP.*

Die LOD Cloud konnte seitdem mit 1301 Datensätzen im Jahr 2021 einen rasanten Wachstum verzeichnen [196]. Betrachtet man hingegen die schwergewichtigen Knoten wie bspw. DBPedia, so zeichnet sich das Muster der automatisierten Annotation ab. DBPedia verwendet die Semistruktur von Wikipedia, um automatisiert Triples abzuleiten, wie bspw. 416.000 Instanzen von Personen [200]. Im Hardwarebereich hingegen werden immer noch Spezifikationen als PDF-Dokumente bereitgestellt, und die Sammlungen großer Onlinevertriebe bieten ebenfalls nur ein für die jeweils verwendeten Filter relevantes Datenset. Neben der Verwendung von Ontologien für die Beschreibung ist auch die Bereitstellung der semantisch annotierten Daten nach den Prinzipien von Tim Berners Lee ein Thema, auf die die vorliegende Dissertation aufbaut und diese auf die Selbstbeschreibung modularer Hardware anwendet.

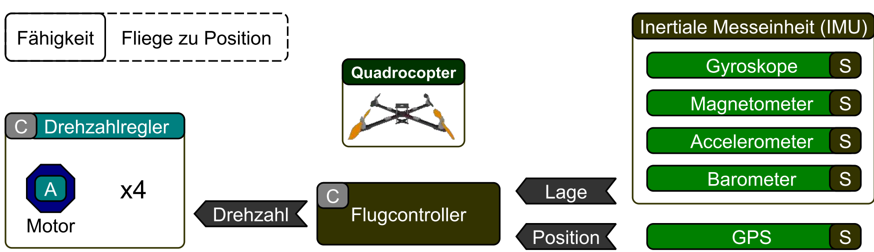

Abbildung 4.10. Beispiel für die Realisierung der Fähigkeit *Fliege zu Position* mit einem Quadrocopter. Über die fünf mit einem *(S)* gekennzeichneten Sensoren können sowohl die Lage, als auch die Position bestimmt werden. Die Controller *(C)* verarbeiten die Sensorwerte und Stellwerte und steuern dadurch die 4 Motoren *(A)* (Eigenwerk)

4.2 Technische Grundlagen für Fähigkeiten

Nachdem im vorherigen Kapitel eine Möglichkeit zur Beschreibung von Sensoren und Aktuatoren anhand von Ontologien vorgestellt wurde, wird in diesem Kapitel der Fokus auf die tatsächliche Ausführung von Fähigkeiten gelegt. In der Modellierung von Fähigkeiten (capability) oder Fertigkeiten (skill), wird die Realisierung mit echter Hardware entweder nicht betrachtet [66], simuliert [70] oder mit nicht modularen Systemen und fest definierten Fähigkeiten realisiert [226, 283]. So bieten Simulationen für Quadrocopter wie RotorS [91] abstrakte Fähigkeiten wie das *Fliege zu Position* an, die bspw. für eine Visualisierung von Formationsflügen [149] verwendet werden kann. Die Realisierung der Fähigkeit *Fliege zu Position* mit einem echten Quadrocopter erfordert hingegen die Fusion unterschiedlicher Sensoren, um über einen Regelkreis die Aktuatoren zu steuern, wie in Abbildung 4.10 illustriert. So wird in der Inertialen Messeinheit (IMU) anhand mehrerer Gyroskope, Magnetometer und Accelerometer die Lage des Quadrocopters bestimmt, wobei ein optionales Barometer einen Aufschluss über die relative Höhe geben kann [209]. Die Position hingegen benötigt externe Trackingsysteme wie beispielsweise das Global Positioning System (GPS), welches in Kombination mit dem optionalen Barometer verwendet werden kann. Im Flugcontroller können die Sensorwerte bspw. mit einem Kalman Filter gefiltert [135] und in einem Regelkreis mit mehreren PID Reglern [233] verarbeitet werden. Wenn für eine Indooranwendung das GPS durch ein Motion Capturing Trackingsystems (MCS) getauscht wird, entstehen mehrere Komplikationen.

Die Positionsdaten des MCS müssen in das System über eine **physikalische Schnittstelle** gelangen. Der Flugcontroller muss auf die neue **logische Schnittstelle** des MCS angepasst werden. Die Kalman Filter sowie die PID Regler benötigen ebenfalls eine Anpassung, da die Positionsdaten des MCS in der Regel deutlich genauer und zuverlässiger sind als das auf Satellitendaten basierende GPS. Diese Änderungen müssen auf den Flugcontroller unter Beachtung der verwendeten Plattform **verteilt** werden. Eine zusätzliche **Versionierung** unterstützt dabei die Codehistorie und ermöglicht, das beide Varianten als Versionen geführt und bei Bedarf getauscht werden können. Dieses Beispiel illustriert die Komplexität, die sich hinter einer einfachen *Fliege zu*

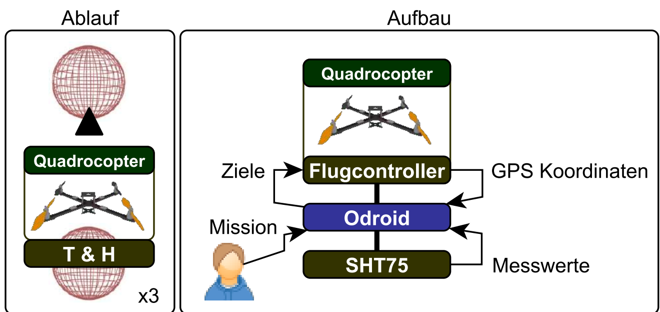

Abbildung 4.11. Grobe Struktur des Ablaufs (links) und des Aufbaus (rechts) des in Abschnitt 3.3.1 skizzierten Experiments der mobilen Messtürme. Die Pfeile signalisieren auf der linken Seite des Aufbaus den Ablauf Missionsverteilung und auf der rechten Seite die kontinuierlichen Messdaten des Quadrocopters und SHT75 Sensors. (Eigenwerk)

Position Fähigkeit verbirgt, und die Problemstellung für ein durchgängiges Konzept eines modularen, selbstbeschreibenden Systems.

Als leitendes Beispiel für die Realisierung einer Fähigkeit mit realer Hardware wird das Experiment *Der Mobile Messturm* aus Abschnitt 3.3.1 aufgegriffen und in Abschnitt 4.2.1 mit dem Fokus auf die verwendeten Fähigkeiten detaillierter ausgeführt. In dem Abschnitt 4.2.2 *Schnittstellen und Controller* werden die technischen Grundlagen mit einem Fokus auf die verwendeten Plattformen und deren Kommunikation miteinander vorgestellt. Für die Entwicklung der Codebausteine werden in Abschnitt 4.2.3 speziell Architekturkonzepte untersucht, die auf mehreren Plattformen verteilt sind und miteinander interagieren. Eine besondere Realisierung einer verteilten Architektur stellt das Robot Operating System (ROS) in Abschnitt 4.2.4 dar, das auch eine Simulation und unterschiedliche Visualisierungen mitbringt. Der Aspekt versionierbaren Codes wird in Abschnitt 4.2.5 anhand von GIT vorgestellt, wobei die Grundlagen sehr kurz zusammengefasst werden und ein Fokus auf eine sinnvolle Verwendung der Werkzeuge gelegt wird. Abschließend wird in Abschnitt 4.2.6 Docker vorgestellt, eine Containervirtualisierung für die Isolierung von Anwendungen.

4.2.1 Experiment: Der Mobile Messturm

Das in Abschnitt 3.3.1 vorgestellte Experiment hat das Ziel, mit drei Quadrocoptern einen automatisierten Auf- und Abstieg durchzuführen, wobei eine Synchronisierung der Quadrocopter untereinander nicht erforderlich ist. Die Messungen des eingesetzten Temperatur- und Luftfeuchtesensors *SHT75* sind kontinuierlich. Die Messpunkte dienen dazu, einen Mittelwert an einer Position zu ermitteln. Zu den Messungen müssen Zeit und Position gespeichert werden, also ein Tripel (h, t, gps), wobei h die Luftfeuchte, t die Zeit in Millisekunden seit 01.01.1970 und gps die Position in 3D Polarkoordinaten darstellt. Die Mission des Auf- und Abstiegs hingegen wird über einen Nutzer eingespielt und gestartet. Der Aufbau besteht, wie in Abbildung 4.11 dargestellt, aus einem Quadrocopter mit eigenem Einplatinencomputer (Flugcontroller), einem zentralen Steuergerät,

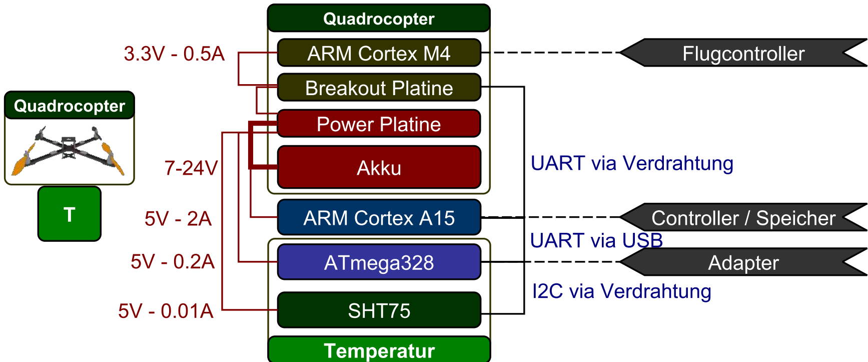

Abbildung 4.12. Aufbau eines nicht modularen Quadrocopters mit einem SHT75 Temperatur- und Luftfeuchtesensors. Die Elemente sind unterteilt in Platinen mit eigenem Mikrocontroller oder Prozessor (ATmega328, ARM) und Platinen ohne programmierbare Schnittstelle wie die Power Platine oder die Breakout Platine. Die linken Assoziationen stehen für elektrische Verbindungen und die rechten für physikalische Schnittstellen. (Eigenwerk)

ebenfalls ein Einplatinencomputer (Odroid), und dem angeschlossenem SHT75. Das in Abbildung 4.11 abgebildete Schema dient der Beschreibung des Ablaufs auf einer sehr abstrakten Ebene. Die für die Realisierung nötigen Grundlagen werden in den folgenden Kapiteln vorgestellt.

4.2.2 Schnittstellen und Controller

Für die Modularität der selbstbeschreibenden Hardwareelemente werden in diesem Abschnitt die Grundlagen für Schnittstellen und Controller betrachtet. Jeder Sensor und Aktuator besitzt eigene Anforderungen bezüglich des Strombedarfs und der physischen Schnittstelle, die meist in Abhängigkeit zueinander auftreten. In der Vernetzung zahlreicher Sensoren im Automotive Bereich wird beispielsweise häufig ein CAN-Bus für die Kommunikation zwischen den Steuergeräten [54] und eine durch die Autobatterie (eigentlich Akkumulator) bedingten Spannung zwischen 12 - 15V verwendet. Zumindest für das Auslesen von Fehlercodes gibt es einen standardisierten On-Board-Diagnose (OBD) Port [5]. Im Bereich der Robotik sind bspw. für Manipulationswerkzeuge an einem Knickarmroboter eine Spannung von 24 - 48V und der Industriestandard EIA-485 als physische Schnittstelle [254] üblich. Andere Domänen sind ebenfalls geprägt durch eigene Standards und Konventionen, sodass ein allumfassender Standard durch die unterschiedlichen Anforderungen an Stromversorgung und physische Schnittstelle schwer oder nicht möglich ist. Eine Möglichkeit, der Problematik zu begegnen, wird in dem Aufbau des Quadrocopters mit einem physikalischen *Adapter* für einen SHT75 Sensor in Abbildung 4.12 illustriert. Der interne Aufbau des für das Experiment angefertigten Prototyps besteht aus mehreren Elementen, die über einen Akku mit Strom versorgt werden, sowie drei unterschiedliche physische Schnittstellen. Die Schnittstellen verwenden entweder den **Universal Serial Bus (USB)** oder werden direkt **verdrahtet**.

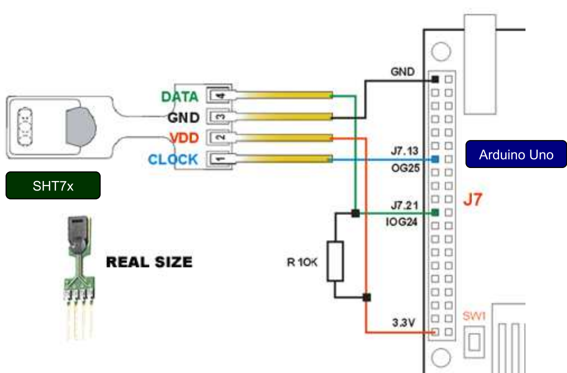

Abbildung 4.13. Pinlayout des SHT7x Sensors von Sensirion mit einem Vorschlag für die Verkabelung an einen Arduino Uno [1]. Da die Sensoren SHT75 und SHT71 einen identischen Aufbau haben, wird hier der Begriff SHT7x verwendet. (Bildquelle [1])

Verdrahtung (GPIO)

Sensoren wie bspw. der verwendete SHT75 haben keinen Stecker, sondern werden über Pins an *General Purpose Input/Output* (GPIO) Ein- und Ausgänge angeschlossen, deren Anordnung nicht standardisiert ist, wie in Abbildung 4.13 illustriert. Typischerweise werden zusätzliche elektrische Bauteile benötigt, wie der abgebildete Widerstand (R 10K) zwischen der Spannungsversorgung (VDD) und dem Data Pin. Für spezielle Anwendungen wie bspw. Breakout Platinen oder bestückte Power Platinen, wie in Abbildung 4.12 skizziert, kommen sehr kleine SMD-Bauteile zum Einsatz. Eine detaillierte Beschreibung möglicher Bauteile und deren Funktion wird in der vorliegenden Dissertation nicht gegeben, aber es sei an dieser Stelle auf die Grundlagenliteratur verwiesen [242]. Als Kommunikationsprotokoll kommt beim SHT75 eine I2C ähnliche Variante zum Einsatz [271]. Bei einer Verdrahtung können Lochrasterplatinen zum Löten oder Steckkontakte mit geeigneten Pin Headern verwendet werden. Alternativ kann aus einer breiten Auswahl an Steckverbindern eine Eigenkonfiguration realisiert werden, wobei insbesondere bei standardisierten Steckverbindern wie bspw. USB eine Belegung vorgeschrieben ist. In Semantic Plug & Play werden die über Kabel verbundenen Sensoren an einen Adapter angeschlossen, um einerseits die physische Schnittstelle zu kapseln und andererseits über eine logische Schnittstelle *Plug & Play* Fähigkeiten zu ermöglichen, wie sie das USB-Protokoll anbietet.

Universal Serial Bus

Die wohl am häufigsten im Alltag verwendete physische Schnittstelle ist der Universal Serial Bus (USB), der in den unterschiedlichen Versionen nicht nur einen Steckverbinder spezifiziert, sondern auch eine logische Schnittstelle, die in Form des USB-Protokolls standardisiert ist. Die Version USB 2.0 mit dem Steckverbinder Typ A wird im Folgen-

dem konkreter ausgeführt, da sie vor allem in den Experimenten eingesetzt wird. Das Pin Layout eines Typ A Steckverbinders besteht aus vier Konnektoren, zwei für die Stromversorgung und zwei für die Datenübertragung. Im Standard des USB 2.0 werden 5V als Spannung angegeben und eine maximale Stromstärke von 0.5A [206]. Erst in späteren Versionen wurde die maximale Stromstärke erhöht und durch optional höhere Spannungen ergänzt. Über das USB-Protokoll muss sich jedes Gerät anhand von drei Kategorien identifizieren. Dazu zählen der verwendete Standard, die Geräteklasse und der Hersteller [125]. Der verwendete Standard bestimmt zum Beispiel die Energieverwaltungsfunktionen. Die Geräteklasse bestimmt den Einsatzzweck, wie etwa die Klassen Drucker oder Massenspeicher. Unter der Kategorie Hersteller können zusätzliche Informationen ausgetauscht werden. Im Beispiel des Quadrocopters mit SHT75 Sensor aus Abbildung 4.11 wird über den USB Steckverbinder ein ARM Cortex A15 mit einem ATmega328 verbunden, wobei das USB-Protokoll für die Geräteidentifizierung verwendet wird. Der ARM Cortex A15 mit einem Linux Betriebssystem dient als Controller und Speicher, in dem die Messwerte gesammelt und persistiert werden. Der SHT75 Sensor ist an den ATmega328 Mikrocontroller angeschlossen, der als Adapter dient, um die Sensorwerte über ein USB Kabel an den ARM Cortex A15 weiterzugeben. Der ATmega328 identifiziert sich über das USB-Protokoll an dem ARM Cortex A15 über einen CH340 USB-Chip, der in Linux als serielle Schnittstelle (UART) zu erkennen gibt. Je nach Betriebssystem und Treiberdatenbank kann der ARM Cortex A15 keinen generischen Treiber für den CH340 Chip finden und eine manuelle Installation ist erforderlich. An diesen Punkt knüpft Semantic Plug & Play an und bietet über eine Selbstbeschreibung einen Zugriff auf die Hardware, die sich grundlegend von Treibern unterscheidet. Generische Treiber können auch in Implementierungen verwenden werden, konkrete Treiber erfordern hingegen eine API oder ein SDK. Semantic Plug & Play abstrahiert Fähigkeiten und versucht nicht ein Schema für alle möglichen Geräteklassen zu definieren. Die Kategorisierung der Fähigkeiten findet stattdessen in Eigenschaften statt, die als Teil der Selbstbeschreibung eine stete Verwendung in der Implementierung ermöglichen.

Controller

Im wissenschaftlichen Bereich sind Einplatinencomputer für den prototypischen Aufbau von Experimenten sehr beliebt [257]. Doch auch in OpenAAS im Rahmen der Verwaltungsschale kommt der weit verbreitete Einplatinencomputer Raspberry Pi zum Einsatz [255]. Dieser basiert wie viele Vertreter der Einplatinencomputer auf der ARM-Architektur, die entgegen einem X86 Prozessor (CISC) einen reduzierten Befehlssatz (RISC) verwendet und durch einen geringen Stromverbrauch überzeugen kann [256]. In Abbildung 4.12 werden zwei unterschiedliche ARM Prozessoren verwendet, einer für den Flugcontroller und ein weiterer für die Speicherung und Verwaltung der Datenströme als zentraler Controller. Der zentrale Controller ist ein Odroid XU4 mit einem ARM Cortex A15, sowie einer 30-Pin und 12-Pin Erweiterung für externe Geräte. Zwar würde der Odroid über die Erweiterung auch über das vom SHT75 verwendete I2C Protokoll verfügen, allerdings wird für die Pins intern eine Spannung von 1.8V für die Signale verwendet. Die Spannung kann durch elektrische Bauteile (level shifter) angepasst werden oder es wird ein Adapter verwendet.

Als Adapter wird in Abbildung 4.12 ein Arduino Nano Mikrocontroller verwendet, der auf einem ATmega328 Chip basiert. Der Vorteil an einem Mikrocontroller ist die einfache Integration von low-level Hardware, sowie einer kompletten Kontrolle der Ausführungszyklen. So können bspw. Interrupts an den Pins anliegen, müssen aber selbstständig implementiert werden. Die Leistung eines Mikrocontrollers ist nicht mit dem eines ARM vergleichbar und auch der persistente Speicher ist in der Regel nur für den Programmcode gedacht. Im Beispiel des Arduino Nano steht ein 1KB großer Electrically Erasable Programmable Read-only Memory (EEPROM) zur Verfügung. Semantic Plug & Play bietet keine generische Lösung für physische Schnittstellen, Spannungsunterschiede oder die eingesetzte Hardware, wie etwa Tinkerforge mit einer begrenzten Menge eigen produzierter Hardwareelemente [141]. Der Standpunkt von Semantic Plug & Play ist vielmehr eine breite Unterstützung für unterschiedliche Hardware, deren physische Schnittstellen adaptiert werden können. So bietet Semantic Plug & Play Selbstbeschreibungsmechanismen für Mikrocontroller, Einplatinencomputer und X86 basierte Systeme, die physikalische Schnittstellen adaptieren können. Das für den Prozess zuständige Modul (Controller) für die Verarbeitung der Selbstbeschreibung kann hingegen sowohl auf Einplatinencomputern, als auch auf X86 basierten Systemen ausgeführt werden. Durch die unterschiedlichen Adapter und Controller müssen auch Softwareartefakte in verschiedenen Programmiersprachen verteilt werden, die über Schnittstellen miteinander kommunizieren. Im Bereich der Softwaretechnik finden sich bereits Ansätze, die diese Thematik aufgreifen.

4.2.3 Softwarearchitekturen verteilter Komponenten

Ein bedeutender Grundstein von Semantic Plug & Play bildet die MDDM-Architektur, die in Kapitel 6 vorgestellt wird und auf gängigen Softwarearchitekturen aufbaut, die in diesem Abschnitt vorgestellt werden. Der Begriff *Softwarearchitektur* wird in der Literatur unterschiedlich definiert und in vielen Teilgebieten der Softwaretechnik angewendet. Hasselbring definiert den Begriff wie folgt [119]:

> *„Die grundlegende Organisation eines Systems, dargestellt durch dessen Komponenten, deren Beziehungen zueinander und zur Umgebung sowie den Prinzipien, die den Entwurf und die Evolution des Systems bestimmen."*
> *- Wilhelm Hasselbring*

Dabei werden Entwurfsmuster der Gang of Four [131] als Mikroarchitekturen tituliert und sowohl die Client-Server Architektur, als auch eine klassische Schichtenarchitektur als Architekturmuster klassifiziert [119]. Eine genauere Unterteilung der Architekturstile wird von Starke et al. anhand unterschiedlicher Kategorien vorgestellt [279]. So wird die Client-Server Architektur unter *Architekturstile verteilter Systeme* kategorisiert und die Schichtenarchitektur unter *Hierarchische Architekturstile.* Die Model-View-Controller Architektur hingegen wird unter *Interaktionsorientierte Systeme* eingeordnet. Dieser Kategorisierung folgend werden die für die vorliegende Dissertation relevanten Architekturmuster und -stile vorgestellt und mit Semantic Plug & Play assoziiert.

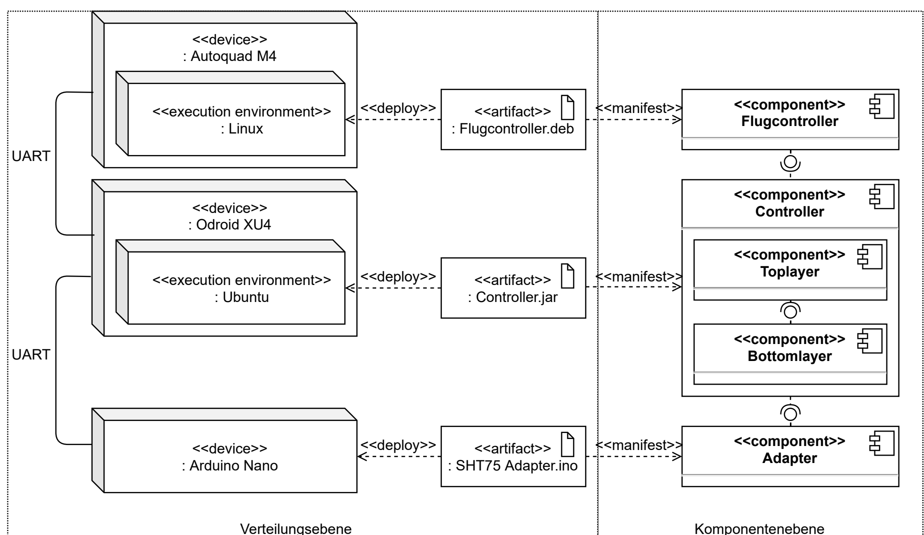

Abbildung 4.14. Kombination eines UML-Verteilungsdiagramms und eines UML-Komponentendiagramms mit Abhängigkeitsbeziehungen. Das verwendete Beispiel korreliert mit dem Aufbau aus Abbildung 4.12. Die konkreten Artefakte sind in exemplarische Komponenten aufgeteilt. Die Komponenten des Controllerartefaktes veranschaulichen eine einfache Schichtenarchitektur. (Eigenwerk)

Architekturstile verteilter Systeme

In Abbildung 4.14 wird illustriert, wie die in dem Experiment benötigten Softwareartefakte anhand einer **Client-Server Architektur** verteilt werden, mit einem zentralen Steuerelement (Odroid XU4), der als Server fungiert und zwei über UART angeschlossene Clients. Eine Alternative ohne zentrales Steuerelement bildet die **Peer-to-Peer** Architektur, in der gleichberechtigte Knoten miteinander kommunizieren. Da ein Peer-to-Peer Netz in der Regel nicht vollvermascht ist, sind Synchronisierungsmechanismen für eine konsistente Datenbasis nötig. Anwendungen für die Peer-to-Peer Architektur bietet Bram Cohen, der sowohl Mitbegründer des BitTorrent Netzwerkes ist [47], ein Netzwerk für verteiltes Filesharing, als auch die Crypto Währung *Chia* entwickelt [48], eine Blockchain basierend auf einem *Proof of Time and Space* Konsensus.

Hierarchische Architekturstile

Die **Schichtenarchitektur** strukturiert das System in mehrere Schichten, wobei höhere Schichten auf die unterliegenden Zugreifen. Dedizierte Schnittstellen, wie in Abbildung 4.14 in der Komponente *Controller* skizziert, sind zwar nicht zwingend vorgegeben, sollten jedoch verwendet werden. Ein klassischer Vertreter ist die 3-Schichten-Referenzarchitektur, in der *Datenhaltungs-*, *Geschäftslogik-* und *Präsentationsschicht* getrennt werden, wie in dem ERP System SAP R3 angewendet [119]. Einen Fokus auf Schnittstellen bietet die **Ports und Adapter Architektur** [131]. Die Komponenten können nur über Ports angesprochen werden während ein Adapter, im Sinne des Gang of Four Entwurfsmusters, die Kompatibilität herstellt.

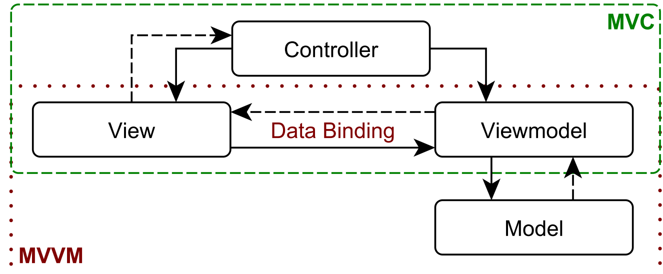

Abbildung 4.15. Kombination des Model View Controller Musters mit dem Model View Viewmodel Muster. Der Zugriff auf das Modell erfolgt über das Viewmodel und die Assoziationen zwischen View und Viewmodel werden mit Data Bindings realisiert. (Eigenwerk)

Interaktionsorientierte Systeme

Das **Model View Controller (MVC)** Architekturmuster findet in zahlreichen Varianten Verwendung, da bspw. die Geschäftslogik nicht eindeutig einem der drei Elemente zugeordnet wird. Nach der Definition von Deacon et al. [172] enthält das *Model* die Daten, die im View dargestellt werden sollen. Üblicherweise kann auch innerhalb des Models eine Schichtenarchitektur verwendet werden, um die Persistenz vom Datenmodell zu trennen. Die *View* kann aus unterschiedlichen Endgeräten bestehen, die beispielsweise eine Grafische- oder Kommandozeilenschnittstelle anbieten. Dadurch bedingt wird der *Controller* entkoppelt, der die View steuert und das Model bei Änderungsbedarf anpasst. In Kombination mit dem **Model View Viewmodel (MVVM)** Architekturmuster kann das MVC Muster ergänzt werden, wie in Abbildung 4.15 skizziert. *Model* und *View* sind identisch zum MVC. Das *Viewmodel* basiert auf dem Prinzip, dass einzelne Darstellungen innerhalb der View nicht auf das gesamte Model zugreifen müssen und spezielle Anpassungen gekapselt werden sollten. Diese Kapselung erfolgt innerhalb eines konkreten Viewmodels, wobei in vielen Realisierungen eine eins zu eins Beziehung zwischen spezifischer View und einem konkreten Viewmodel gilt [277]. Der Zugriff wird wenn möglich mit Data Bindings realisiert, also einer direkten Anbindung eines Steuerelements der Nutzeroberfläche zu einem Objekt des Viewmodels.

Außerhalb der von Starke et al. aufgestellten Kategorien klassischer Softwarearchitekturen [279] ist die nicht normierte **Serviceorientierte Architektur (SOA)** im Kontext der vorliegenden Dissertation zu erwähnen. Eine Definition nach Richter et al. beschreibt SOA als *„ein Architekturmuster, das den Aufbau einer Anwendungslandschaft aus einzelnen fachlichen Anwendungsbausteinen beschreibt, die jeweils eine klar umrissene fachliche Aufgabe wahrnehmen. Die Anwendungsbausteine sind lose miteinander gekoppelt, indem sie einander ihre Funktionalitäten in Form von Services anbieten"* [247]. Eine besondere Herausforderung einer SOA ist die Orchestrierung einzelner Services in konkreten Realisierungen, wie bspw. anhand von Web Services.

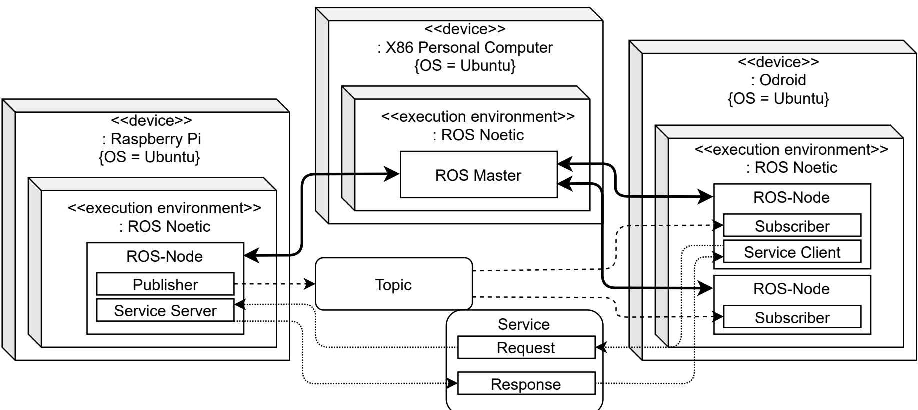

Abbildung 4.16. Exemplarischer Aufbau eines ROS-Systems innerhalb eines UML-Verteilungsdiagramms mit abweichender Syntax. Die auf den Knoten verteilten ROS-Nodes kommunizieren über Topics und Services miteinander, während der ROS-Master die Orchestrierung übernimmt. (Eigenwerk)

In Implementierungen werden unterschiedliche Softwarearchitekturen verwendet, kombiniert und in Teilaspekten neu definiert, wie die verwendeten Beispiele andeuten. So kann innerhalb eines Web Service eine Kombination aus MVC und MVVM verwendet werden und das Model auf einer Schichtenarchitektur aufbauen. Die Orchestrierung der Web Services hingegen findet in einer Server-Client Architektur statt. In Semantic Plug & Play wird eine Architektur vorgestellt, die auf den Prinzipien des MVVM Architekturmusters aufbaut und diese auf Domänen anwendet. Eine stark auf Komponenten aufbauende Architektur mit Service-ähnlichen Konstrukten findet sich in dem Framework ROS wieder, das in Semantic Plug & Play als Erweiterung für die Steuerung von Robotersystemen integriert werden kann.

4.2.4 Modellierung und Simulation im Robot Operating System (ROS)

Das **Robot Operating System (ROS)** ist ein Framework für Roboteranwendungen mit einem starken Fokus auf wiederverwendbare Komponenten. Durch eine große Anzahl an Bibliotheken wie laserbasiertes 2D SLAM, 3D-Punktwolken-basierte Objekterkennung oder die Adaptive Monte Carlo Lokalisierung ist es zu einem Quasi-Standard in der Forschung geworden [157]. Nach Quigley et al. werden von ROS folgende Prinzipen verfolgt [236]:

Peer-to-Peer Architektur

Roboterapplikationen bestehen üblicherweise aus mehreren Komponenten, die sich häufig in einem heterogenen Netzwerk befinden. So gibt es beispielsweise bei großen Servicerobotern [28], für die ROS entwickelt wurde [236], typischerweise mehrere Bordcomputer, die über Ethernet verbunden sind. Dieses Netzwerksegment ist wiederum meist über Wireless LAN mit leistungsstarken Offboard-Rechnern verbunden, die rechenintensive Aufgaben wie die 3D-Punktwolken-basierte Objekterkennung ausfüh-

ren [283]. In der Peer-to-Peer Architektur von ROS werden einzelne Funktionen in sogenannte **Nodes** gekapselt, wobei eine Node nach dem Single Responsibility Prinzip aufgebaut sein sollte. Für die Orchestrierung der Nodes wird das Peer-to-Peer in ROS jedoch durch die Verwendung eines ROS-Master, wie in Abbildung 4.16 abgebildet, gebrochen. Über den **ROS-Master** werden die auf dem TCP Protokoll basierenden Verbindungen hergestellt und verwaltet. Die Kommunikation zwischen den Nodes läuft unter anderem über **Topics**, die das Publish-Subscribe Prinzip verfolgen. Will bspw. eine Node eine Nachricht senden, kann sich diese auf ein Topic, üblicherweise ein einfacher String, als *Publisher* registrieren. Auf diesem Topic können sich hingegen beliebig viele *Subscriber* als Empfänger registrieren. Topics folgen damit einem asynchronen Multicast Prinzip. Für synchrone Nachrichten bietet ROS sogenannte **Services** an, auf die man sich analog registriert und ein Tupel aus *request* und *response* gebildet wird [236]. Ein Beispiel einer auf drei Knoten verteilten ROS-Anwendung mit mehreren ROS-Nodes und der beiden Kommunikationsvarianten ist in Abbildung 4.16 skizziert.

Multilingual

Wie bereits in der Einleitung erwähnt, werden für die Integration von Sensoren und Aktuatoren meist unterschiedliche Programmiersprachen verwendet. Aber auch weitere Kriterien wie persönliche Vorlieben, Laufzeiteffizienz und eine Vielzahl anderer Gründe tragen zur Auswahl einer Programmiersprache bei. Aus diesen Gründen ist ROS so konzipiert, dass es prinzipiell sprachenneutral ist. Eine Basisimplementierung wird aktuell für C++, Python, Octave und LISP angeboten, wobei weitere Sprachportierungen in verschiedenen Stadien verfügbar sind [236].

Open-Source

Der vollständige Quellcode von ROS ist öffentlich zugänglich, damit die Fehlersuche auf allen Ebenen des Software-Stacks erleichtert wird. Wie am Beispiel des proprietären Microsoft Robotic Studio [132] enthalten die meisten bestehenden Robotik Software-Projekte Treiber oder Algorithmen, die außerhalb des Projekts wiederverwendet werden könnten. Leider ist ein Großteil dieses Codes sehr mit der Middleware verstrickt, so dass es schwierig ist, seine Funktionalität zu „extrahieren“ und außerhalb des ursprünglichen Kontextes wiederzuverwenden [236]. In ROS hingegen sollen Treiber- und Algorithmen in eigenständigen Bibliotheken entwickeln werden, die keine Abhängigkeiten zu ROS haben. Das ROS-Build-System führt modulare Builds innerhalb des Quellcode-Baums durch, und die Verwendung von CMake macht es vergleichsweise einfach, dieser schlanken Ideologie zu folgen [236]. Dieser Ideologie folgend ist die eigenständige und umfangreiche Simulationsumgebung **Gazebo** lediglich ein schlanker Baustein unter vielen Bibliotheken. Gazebo verwendet unter anderem die Open Dynamics Engine und Bullet für die Physik in Kombination mit OpenGL für die Visualisierung [146].

In Semantic Plug & Play ist ROS und Gazebo als Erweiterung für die Interaktion und Simulation mit Robotersystemen vorgesehen. Durch den starken Fokus von ROS auf wiederverwendbare Komponenten können Fähigkeiten gekapselt und mit einer Selbstbeschreibung integriert werden. Werkzeuge wie eine grafische Modellierung für ROS-

Systeme oder einer *Domain Specific Language* (DSL) für ROS-Prozesse sind im Rahmen der Dissertation entstanden und vereinfachen die Integration in Semantic Plug & Play.

Den Open-Source-Prinzipien folgend wird der Quellcode von Gazebo, sowie der Großteil an verfügbaren Bibliotheken über die Versionsverwaltung GIT bereitgestellt. Kompilierte Artefakte hingegen werden automatisiert unter Einhaltung bestimmter Regeln aus dem Quellcode erzeugt. In Semantic Plug & Play ist GIT hingegen ein integraler Bestandteil für die automatisierte Verteilung von Fähigkeiten, die auf Quellcodeversionen und kompilierten Artefakten aufbaut.

4.2.5 Versionierung und Verteilung mit GIT

Das Versionskontrollsystem GIT dient der Entwicklung und Pflege eines Repositories mit Inhalten, der Ermöglichung des Zugriffs auf historische Ausgaben der einzelnen Daten und der Aufzeichnung aller Änderungen in einem Log. Als Erfinder und Namensgeber gilt Linus Torvalds, wobei GIT kein Akronym bildet, sondern durch das scherzhafte Zitat: „*Ich bin ein egoistischer Kerl und nenne alle meine Projekte nach mir selbst. Erst Linux, jetzt GIT*“ [181] begründet wird. Eine Besonderheit von GIT im Gegensatz zu klassischen Versionskontrollsystemen wie Subversion ist der dezentrale Ansatz der Repositories. Bei einem *Checkout* wird nicht nur der aktuelle Stand, sondern auch eine Kopie des Repositories verwendet, auf dem gearbeitet werden soll. Das lokale Repository kann nach Änderungen wieder mit dem zentralen Repository zusammengeführt (merge) werden. Für eine ausführliche und praxisorientierte Einführung für die Mechaniken in GIT wird an dieser Stelle auf eine Standardliteratur [181] verwiesen.

GIT wurde vor allem für die Open-Source-Entwicklung des Linux Kernels entwickelt, also unter dem Gesichtspunkt, dass sehr viele Entwickler an einem Projekt arbeiten [181]. Ständige Synchronisation mit einem zentralen Repository bilden dabei einen Entwicklungsengpass. So haben sich in unter GIT unterschiedliche Methodologien entwickelt, wie eine parallele Entwicklung stattfinden kann. In GIT werden Entwicklungsprozesse üblicherweise in einzelne Abzweigungen (Branches) strukturiert, wie in Abbildung 4.17 illustriert [41]. Der *main* Branch (auch bekannt als master) bildet den Hauptzweig und sollte immer eine lauffähige Version des Projektes beinhalten. Er dient als Ankerpunkt für *releases*, welche ebenfalls in einem eigenen Branch ausgelagert werden können. Der *dev* Branch bildet den Entwicklungszweig, von dem einzelne *feature* Branches abgezweigt werden können. Nach der Fertigstellung eines Features können die in Abbildung 4.17 illustrierten Pipeline Strukturen angewendet werden. Buildserver wie bspw. Jenkins [44] kompilieren den Source-Code in einem Branch. Sollten keine Fehler während der Kompilierung auftreten, werden anschließend die *Unit Tests* [326] durchlaufen und abschließend Testszenarien der *Integrationstests* [17] durchgeführt. In größeren Projekten, wie bspw. dem Linux Kernel findet das Zusammenführen erst nach einem Reviewverfahren statt, das auf eine entsprechende Anfrage (pull request) stattfindet [281]. Üblicherweise werden für ein Release weitere Testverfahren auf Produktivsystemen durchgeführt, die unter anderem durch Mechanismen des *Continous Delivery (CD)* unterstützt werden und in Kombination mit der CI als *CI-CD Pipeline* tituliert wird [10].

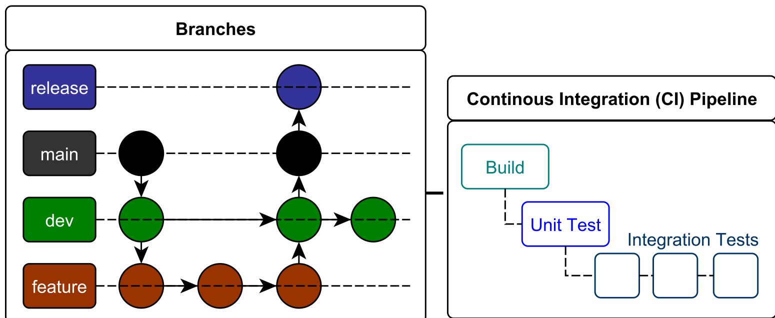

Abbildung 4.17. Skizzierung üblicher Verfahrensweisen für GIT-Branches in Kombination mit einer *Continous Integration (CI) Pipeline* in Anlehnung an die Skizze aus einem Blog von Aleks Pamir [220]. Die vier Linien mit release, main, dev und feature zeigen exemplarische Branches mit einem Versionsverlauf, der durch die jeweiligen Kreise symbolisiert wird. Die CI Pipeline bezieht sich dabei nicht auf einen einzelnen Branch, sondern kann beliebig angewendet werden. (Eigenwerk)

Dieser Methodologie in unterschiedlichen Variationen folgen bereits zahlreiche Open-Source-Projekte, wie ROS, Apache, Python und viele weitere [26], unter anderem auch im wissenschaftlichen Bereich [278]. Die Vorteile finden sich nicht nur in der Codequalität sondern auch in der standardisierten Bereitstellung des Quellcodes sowie der Artefakte. Ein Beispiel liefert das bereits erwähnte Projekt *Chia-Network* von Bram Cohen [48], das die Plattform GitHub verwendet. Die HTTP-URI[2] führt direkt zum Source-Code und den Artefakten, die in einzelne Releases aufgeteilt, versioniert und jeweils durch ein Changelog beschrieben sind. Die CI-CD liefert dabei die Installationsdateien, die auf den Kompilaten des Buildservers des *dev* und *main* Branch aufbaut und für drei Betriebssysteme bereitgestellt werden. Jedes Artefakt erhält dabei wieder eine HTTP-URI und über den Tag *latest* kann die aktuellste referenziert werden.

In Semantic Plug & Play werden diese URIs für die Artefakte einzelner Fähigkeiten verwendet, die zur Laufzeit heruntergeladen und ausgeführt werden können. Die Ausführung der Artefakte ist jedoch anhängig vom eingesetzten Betriebssystem und oftmals bestehen Abhängigkeiten zu anderen Artefakten. So werden für das Kompilieren, Testen und Ausführen des Quellcodes, bzw. der Artefakte, häufig Containervirtualisierungen verwendet. So ist beispielsweise die bekannteste Containervirtualisierung *Docker* direkt in der Pipelinestruktur von Jenkins vorhanden [296].

4.2.6 Virtualisierung und Container

Für das Bereitstellen und Testen von Softwareartefakten müssen insbesondere unterschiedliche Betriebssysteme und Abhängigkeiten zu anderen Artefakten untersucht werden. Eine Möglichkeit, dies auf einer dedizierten Hardware zu realisieren, ist die Verwendung sogenannter *Virtualisierungsplattformen (VMM)*. In Abbildung 4.18 ist der Aufbau einer Virtualisierungsplattform skizziert. Dabei sind zwei unterschiedliche

[2]https://github.com/Chia-Network/chia-blockchain (Stand: 17.05.22)

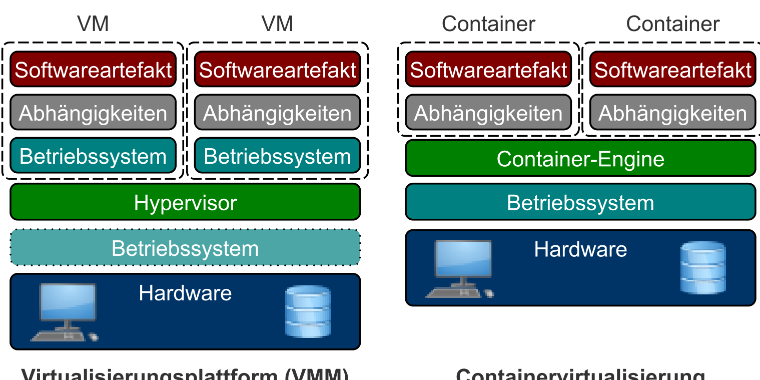

Abbildung 4.18. Skizze der Kernkomponenten einer Virtualisierung über Virtuelle Maschinen (links) und einer Containervirtualisierung (rechts). (Eigenwerk in Anlehnung an Potdar et al. [231])

VMMs zu unterscheiden. Die erste Variante wird innerhalb eines Betriebssystems installiert und ausgeführt, wie bpsw. VirtualBox [297]. Die zweite Variante benötigt kein Betriebssystem und setzt direkt auf der Hardware auf, kann aber als Basis auf einem Betriebssystem basieren, wie bspw. Proxmox [158]. Der *Hypervisor* bildet die abstrakte Zwischenschicht der Hardware und den verschiedenen Gastsystemen, auch *Virtuelle Maschine (VM)* genannt. Virtuelle Maschinen basieren auf einem Abbild (Image), bspw. ein (angepasstes) Betriebssystem. Auf dem Betriebssystem innerhalb der VM können wiederum beliebige Applikationen sowie deren Abhängigkeiten installiert werden. Da die installierten Betriebssysteme eine eigene Ressourcenverwaltung haben, müssen bspw. Arbeitsspeicher und virtuelle CPU-Kerne vor dem Start eine VM zugeteilt werden. Durch die Kapselung, Snapshotmechanismen und den Fokus auf Sicherheit (Security) liegt der Hauptanwendungsbereich einer Virtualisierungsplattform im Serverbereich [292].

Die Containervirtualisierung wie bspw. Docker fokussiert hingegen auf die Softwareartefakte und deren Abhängigkeiten [6]. Eine Abhängigkeit sind andere Bibliotheken, Programme oder Treiber, die in einer bestimmten Version für das Ausführen des Softwareartefaktes benötigt werden. So ist eine ROS-Version beispielsweise für eine bestimmte Version von Ubuntu verfügbar und hat zudem Abhängigkeiten zu bestimmten Versionen von Python [252]. Betriebssysteme nutzen bspw. Installer oder Skripte für das Auflösen dieser Abhängigkeiten, bzw. haben eigene Paketverwaltungen. Die Containervirtualisierung hingegen verwendet Container, in der die Abhängigkeiten bereits aufgelöst sind und ein funktionaler Zustand erreicht ist. Die Container-Engine verwaltet analog zum Hypervisor die Container und den Zugang zur Hardware über das Betriebssystem. Der größte Vorteil gegenüber einer Virtuellen Maschine ist vor allem die deutlich bessere Performance [178, 231] und die Ressourcenverwaltung durch das unterliegende Betriebssystem. Dem Vergleich ist allerdings hinzuzufügen, dass er sich in etwa verhält wie der Vergleich zwischen ARM und X86 Prozessoren und der Einsatzzweck differiert. Ein vollwertiges Betriebssystem, wie es bei einer VM zum Einsatz kommt, ist bei

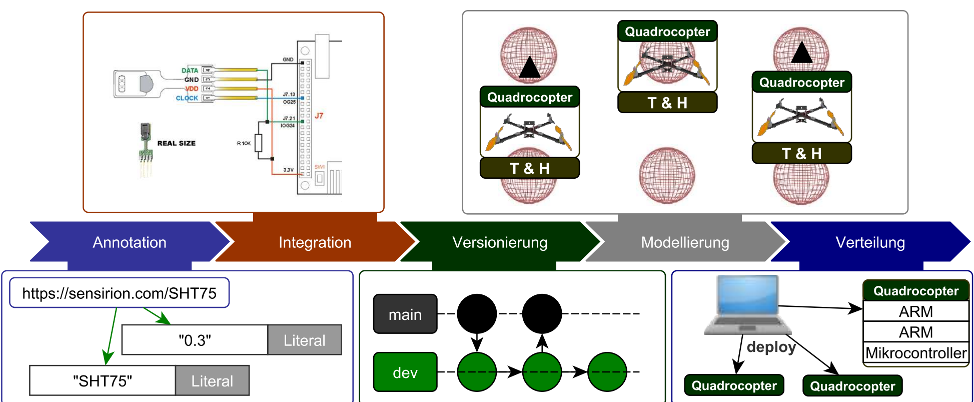

Abbildung 4.19. Zusammenfassung der behandelten Themengebiete und Einordnung in die einzelnen Disziplinen. Die Struktur ist nicht als strikte Abfolge der Themengebiete zu betrachten, da sich bspw. die Versionierung über alle Themengebiete erstreckt und sowohl die Annotation als auch die Verteilung auf Versionierungsmechanismen aufbauen. (Eigenwerk)

Containern nicht gegeben. Dennoch sind damit einfache und effiziente Verteilungen von Softwareartefakten auf Hardware möglich, die durch Skripte (Docker Compose) zusammengefasst und auf unterschiedliche Knoten (Docker Swarm) verteilt werden können [290].

Vor allem für komplexe Systeme bestehend aus mehreren Hardwareelementen, wie im Experiment der virtuellen Messtürme, ist eine einfache Verteilung essentiell. Jeder der drei Aufbauten eines Quadrocopters besteht wiederum aus drei Knoten (2x ARM, Mikrocontroller), wie in Abbildung 4.19 illustriert. Durch das Testen in der Simulation gelingt es zwar, den Ablauf zu konstruieren. Die tatsächlich verwendeten Versionen sowie deren Abhängigkeiten für eine verteilte Ausführung bedingen jedoch eine Strategie, die mit der Unterstützung von Containervirtualisierungen ermöglicht wird. Doch Container können nicht für alle Hardwareelemente als Lösung der Verteilung betrachtet werden. So sind Mikrocontroller durch die begrenzte Leistung und Speicher nicht für Virtualisierungen geeignet, ebenso wie die Annotation der Sensoren, die ebenfalls für Mikrocontroller konzipiert ist. Das durchgängige Konzept der Selbstbeschreibung wird im nächsten Kapitel unter Verwendung der Grundlagen vorgestellt.

Zusammenfassung. Eine Selbstbeschreibung unterliegt der Kombination von Eigenschaften und ausführbaren Fähigkeiten von Hardwareelementen. Das Konzept für diesen Zusammenschluss umfasst neben einer Ontologie auch die konkrete Einbindung lauffähigen Codes in das System. Werden nicht nur einzelne rekonfigurierbare Systeme betrachtet, sondern Ensembles, müssen koordinative Prozesse berücksichtigt werden, wie bspw. für synchrone Bewegungen. Die Lücke zwischen Simulation und Realität wächst hingegen mit der Modularität und der Größe des Ensembles. In diesem Kapitel werden dafür die konzeptionellen Grundlagen vorgestellt.

Konzept der Selbstbeschreibung

5.1 Die Ontologie der Selbstbeschreibung **64**
5.1.1 Experiment: Ausbreitungsmodellierung 65
5.1.2 Eigenschaften und Fähigkeiten 68
5.1.3 Wissensbasis aufbauend auf Ontologien 75
5.1.4 Integration ausführbaren Codes 77
5.2 Fähigkeiten modellieren **79**
5.2.1 Experiment: Synchrone Glasfasermessung 80
5.2.2 Prozesslogik 82
5.2.3 Verteilung 88
5.3 Modulare Simulation und Visualisierung **91**
5.3.1 MR-Bausteine 92
5.3.2 Simulation 94
5.4 Methodologie von Semantic Plug & Play **95**

Semantic Plug & Play umfasst eine Selbstbeschreibung von modularen Hardwareelementen, die zur Laufzeit durch Plug & Play Mechanismen ausgetauscht werden kann. Die Selbstbeschreibung unterteilt Eigenschaften und Fähigkeiten, die einerseits in maschinenlesbarer Form vorliegen und andererseits mit konkreten Implementierungen verknüpft werden.

Die *Annotation* und *Integration* wird im Kapitel *Ontologie der Selbstbeschreibung* vorgestellt, mit einer Ontologie, die auf Basis der vorgestellten Semantic Web Technologien aufgebaut ist und mit GIT-Repositories, bzw. Docker Containern verknüpft wird, um die Verwaltung und Bereitstellung von ausführbaren Fähigkeiten zu ermöglichen. Als leitendes Beispiel dient dabei das Experiment der *Ausbreitungsmodellierung*, mit lediglich einem konfigurierbaren Hardwaresystem.

Die verteilte Prozesslogik hingegen wird an dem Experiment der *synchronen Glasfasermessung* illustriert, in dem mehrere Systeme zum Einsatz kommen. Für die *Modellierung*, wie in Abbildung 5.1 illustriert, wird eine *Block Definition Language* für die Prozessdefinition vorgestellt, die unter anderem auf das Robot Operating System (ROS) aufbaut.

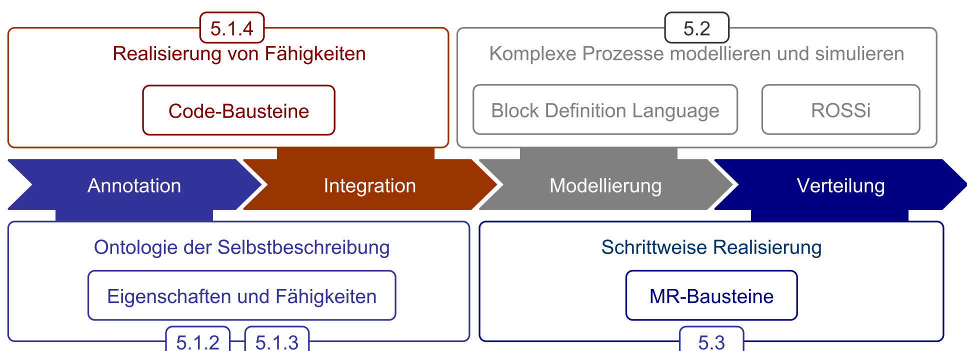

Abbildung 5.1. Übersicht der vier Schwerpunkte in diesem Kapitel. Beginnend mit der Definition von Eigenschaften und Fähigkeiten, die mit Code Bausteinen assoziiert werden, über die Modellierung von Prozessen, bis zu einer modularen Simulation mit MR-Bausteinen. (Eigenwerk)

Abgeschlossen wird das Kapitel mit der Vorstellung einer *modularen Simulation*, die auf dem Konzept der Selbstbeschreibung aufbaut und der komplexen Verteilung mit sogenannten Mixed Reality (MR)-Bausteinen entgegentritt.

5.1 Die Ontologie der Selbstbeschreibung

Das Thema *selbstbeschreibende Hardware* scheint bislang ein recht wenig betrachteter Forschungsbereich zu sein, zumindest wird dies dadurch untermauert, dass unter dem englischen Suchbegriff (self descriptive Hardware) auf Google vorrangig die eigenen Publikationen [84, 153, 309, 311] des Autors zu finden sind. Der Standpunkt, dass das Konzept gänzlich neue Wege beschreitet, stimmt allerdings nur teilweise. So wurden bereits in Kapitel 2 Technologien vorgestellt, die ein ähnliches Ziel verfolgen, sich jedoch in den Anforderungen und der Umsetzung unterscheiden. Für die Selbstbeschreibung von Semantic Plug & Play werden konkret folgende Anforderungen definiert:
Die Selbstbeschreibung ...

... abstrahiert Eigenschaften und Fähigkeiten,

... unterstützt die Ausführung von Fähigkeiten,

... kann maschinenlesbar erweitert werden,

... unterstützt komplexe Systeme (bestehend aus mehreren Subsystemen) und

... befindet sich direkt in oder physikalisch an der Hardware.

So bieten die bereits vorgestellten TEDs ein Schema für die Speicherung in der Hardware, haben allerdings kein Konzept für die maschinenlesbare Erweiterung [232]. Das auf einer Ontologie basierende Konzept von Dibley et al. verwenden reale Hardware, allerdings mit zentralem Server, der die Beschreibungen verwaltet [61]. Diesem Konzept folgen auch konkrete Projekte mit eigenen Ontologien, wie etwa Knowrob [283] oder Projekte aus der Meteorologie [15, 210], die auf bereits existierenden Ontologien, wie SSN [300], aufbauen. Der Begriff *Cyberphysische Systeme (CPS)* nach der Definition von Broy [36]

kapselt Hardware durch eingebettete Systeme, fokussiert jedoch die Schnittstellen zwischen den Systemen. Eigenschaften hingegen spielen für die Schnittstellenbeschreibung eine untergeordnete Rolle, wie in dem Konzept der Verwaltungsschale, das nach Ye et al. [325] eine CPS Kategorie darstellt. Eine Ausprägung eines CPS ist das in Kapitel 1 aufgeführte *Internet der Dinge*, das auch als *Systems of Systems* tituliert wird, solange die globale Vernetzung der Systeme einem übergeordneten Zweck dient, wie bspw. der Datenanalyse [36]. Diese relativ abstrakt definierten Konzepte führen wiederum zu den unterschiedlichen Realisierungen der in Kapitel 2 vorgestellten Arbeiten im Rahmen der Verwaltungsschale. Dabei sei anzumerken, dass im Rahmen der Initiative *Industrie 4.0* bereits sehr viele unterschiedliche Standards existieren und ein Verzicht auf die verwendeten Techniken wie AutomationML oder OPC-UA nicht gewünscht ist, sondern vielmehr das Gegenteil, der Einbezug. Versuche, die Standards zu kombinieren, führen dabei unweigerlich zu einer Verteilung der semantischen Beschreibung, wie bspw. in OpenAAS [255]. Ein radikaler Neuanfang, wie bei der BBC Mediendatenbank, die sehr erfolgreich mit Semantic Web Strategien restrukturiert und vernetzt wurde [145], ist demnach praktisch nicht umsetzbar.

Die Abstraktion von Fähigkeiten in Kombination mit Eigenschaften wird in keinem der oben benannten Projekte realisiert, sondern durch konkrete Fähigkeiten, wie etwa *Messen*, gelöst. Die konkrete Fähigkeit *Messen* wird auch im Experiment der **Ausbreitungsmodellierung** verwendet, das als leitendes Beispiel in diesem Abschnitt fungiert. Die konkreten Fähigkeiten des Experiments dienen dabei als Grundlage für die Definition der abstrakten **Eigenschaften und Fähigkeiten**, die in eine Ontologie überführt werden. Besonders aktuatorische Fähigkeiten wie etwa *Fliege zu Position* erfordern eine Kombination aus mehreren verteilten **ausführbaren Code Bausteinen**, wie in Kapitel 4 gezeigt. Durch die Kombination der Messwerte mit den GPS-Positionen müssen komplexe Systeme wie der Quadrocopter in Subsysteme unterteilt werden, um Fähigkeiten einzelner Sensoren zu erhalten. Da bereits Ontologien für GPS-Positionen oder Sensormesswerte existieren, muss dafür keine Neuentwicklung bestehen, sondern vielmehr eine **Erweiterbarkeit** geschaffen werden, wie in den aufeinander aufbauenden Ontologien nach Rehbein et al. [241].

5.1.1 Experiment: Ausbreitungsmodellierung

Das Konzept des Experiments der Ausbreitungsmodellierung besteht einerseits aus den **Algorithmen** für die **Suche** nach der Gasquelle und einer **Vorhersage** für die Ausbreitung auf der Grundlage verschiedener in situ Messungen, und andererseits aus dem Konzept eines **virtuellen Anemometers**, das durch die Eigenschaften und Fähigkeiten eines komplexen Systems, dem Quadrocopter, abgeleitet werden kann. Die Abbildung 5.2 zeigt dabei die visualisierte Gaswolke aus Abschnitt 3.3.3 in der Vogelperspektive mit einer Rasterung, sowie den strukturellen Aufbau eines Quadrocopters mit Sensoren und den entsprechenden Eigenschaften und Fähigkeiten, die für die Algorithmen nötig sind.

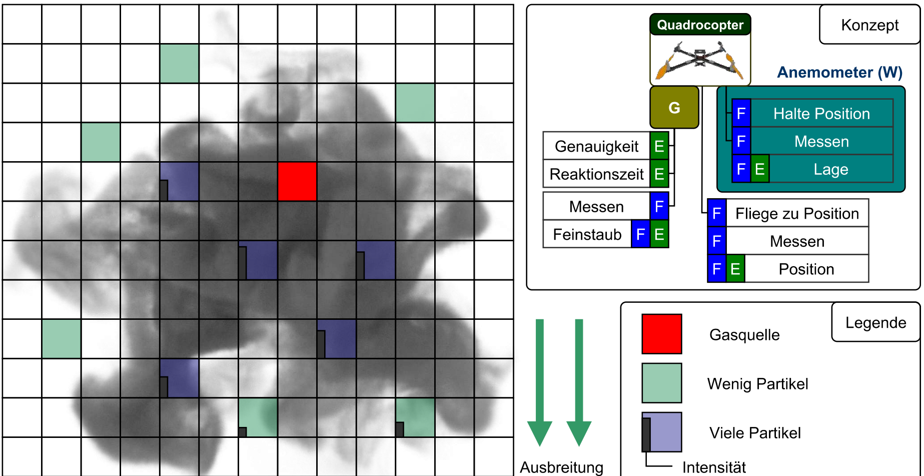

Abbildung 5.2. Konzeptionierung des Experiments: *Ausbreitungsmodellierung* mit einer Unterteilung des Gebietes in ein Raster für Messpunkte, die zeitabhängig sind. Die ausgefüllten Rasterelemente sind bereits gemessene Positionen mit einer Angabe der Intensität (links). Das Konzept umfasst die Unterteilung des Quadrocopters und des Sensors in Eigenschaften (E), Fähigkeiten (F) und Eigenschaften von Fähigkeiten (FE). (Eigenwerk)

Virtuelles Anemometer

Ein Quadrocopter besitzt in der Regel einen Positions- und Lageregler für die absolute Position und die Neigung des Quadrocopters, wie in Abschnitt 4.2 beschrieben. Dadurch kann ein Quadrocopter seine Position selbstständig halten, obwohl externe Kräfte auf ihn einwirken. Die Hypothese ist, dass die Kombination der Fähigkeiten *Halte Position* und *Messen der Lage* Aufschluss über die Windrichtung und -stärke gibt, wie in Abbildung 5.2 illustriert. Der Ansatz des virtuellen Anemometers wurde bereits in mehreren Publikationen [219, 322] untersucht und realisiert, mitunter während des Fluges, wenn die erwartete Neigung für die Beschleunigung miteinbezogen wird. Im Rahmen der Selbstbeschreibung jedoch bildet es ein illustratives Beispiel für nicht direkt ersichtliche Fähigkeiten, die aus Subsystemen eines komplexen Systems entstehen.

Algorithmen für die Suche und Vorhersage

Es existieren zahlreiche Algorithmen für die Suche oder Vorhersage, die üblicherweise anhand der Anforderungen ausgewählt werden, zum Beispiel ob mehrere Quellen berücksichtigt werden sollen oder ein definierter Suchraum vollständig abgesucht wird. Deswegen werden an die Algorithmen für die vorliegende Fallstudie folgende Anforderungen und Einschränkungen gestellt:

- Der Suchraum kann aufgrund der lokalen Begrenzung eines Gasunfalls eingeschränkt werden.
- Es wird von nur einer Quelle ausgegangen.

- Die Messpunkte sind nicht vorgegeben.
- Der Suchraum kann nicht vollständig abgesucht werden.
- Lokale Wetterverhältnisse sind auch temporal einzubeziehen.
- Unterstützung für Ensembles (mehrere Quadrocopter)

Für den **Suchalgorithmus** werden von Lochmatter et al. [179] unterschiedliche Ansätze vorgestellt und anhand unterschiedlicher Kriterien bewertet. Der probabilistische Ansatz der *Sequentiellen Monte-Carlo-Methode* wird zwar üblicherweise für die Selbstfindung von Robotern verwendet [286], kann aber auch auf die Quellenlokalisierung übertragen werden [175]. Auf der Basis einer Unterteilung des Suchraumes in einzelne Raster werden initial Partikel gleichverteilt, wobei jeder Partikel potentiell der Ort einer Quelle sein kann. Je in situ Messung oder festgelegter Iteration werden die Wahrscheinlichkeiten für die Partikel neu berechnet und es wird gegebenenfalls ein Resampling eingeleitet, um zeitliche Abhängigkeiten zu berücksichtigen. Dadurch lässt sich der Bereich einer potentiellen Quelle probabilistisch über die Zeit einschränken. Durch die Unabhängigkeit der Messpunkte können ferner mehrere Quadrocopter eingesetzt werden. Die Rasterelemente werden durch die Fähigkeit *Fliege zu Position* angeflogen und eine Messung wird durch die *Halte Position* und die Sensorfähigkeit *Messen* initiiert. Eine Abschätzung der Dauer einer Messung wird anhand der Eigenschaft *Messzeit* getroffen.

Die **Vorhersage** der Ausbreitung wird durch den Kernel DM Algorithmus nach Reggente et al. [240] konzipiert. Das Raster als Ergebnis des *Suchalgorithmus* dient ebenfalls als Grundlage, wobei die Konzentration der Aerosole aller Rasterelemente mithilfe einer Gauß-Verteilung geschätzt wird. Durch einzelne Messungen wird das Modell über die Zeit verfeinert, wobei eine Hypothese von Lochmatter et al. [179] es ermöglicht, die Messwerte der Nachbarrasterelemente miteinzubeziehen. Die Hypothese lautet, dass bei einer Gas- oder Aerosolwolke die lokale Nähe entnommener Messungen zu geglätteten Übergängen benachbarter Positionen führt und dadurch Aussagen über naheliegende Räumlichkeiten getroffen werden können. Also wenn der Sensor an einer Position einen hohen Messwert erfasst, dann wird in einer naheliegenden Position ein ähnlicher Messwert zu erwarten sein. Diese Hypothese wird durch Reggente et al. [240] mit dem Zusatz der Windmessung ergänzt, wonach sich Gase und Aerosole in Windrichtung stärker ausbreiten. Dies führt zu einem feineren Modell und der Verwendung des *Virtuellen Anemometers*. Somit sind für die *Vorhersage* der Ausbreitung alle in Abbildung 5.2 aufgeführten Eigenschaften und Fähigkeiten notwendig, wobei die Fähigkeit der *Messen* der Position des Quadrocopters relevant ist, um die Fähigkeit *Fliege zu Position* abzuschließen.

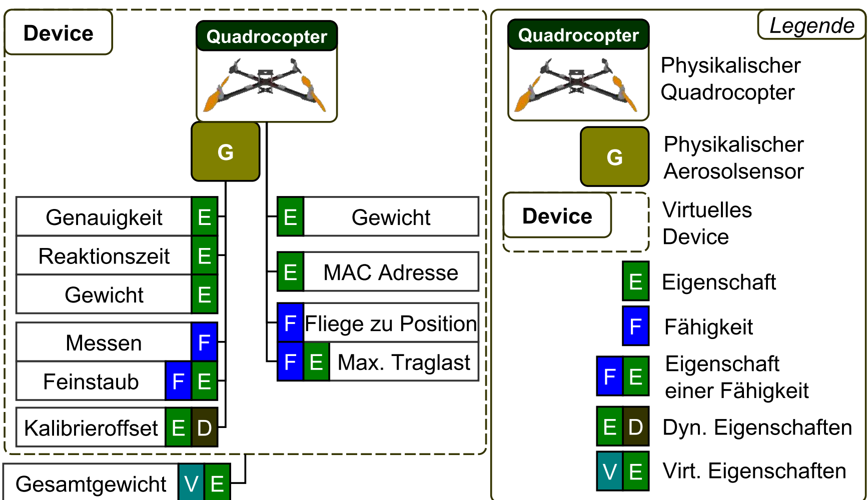

Abbildung 5.3. Darstellung der Eigenschaften und Fähigkeiten im Experiment der Ausbreitungsmodellierung um das Gewicht zu definieren, sowie dynamischen Eigenschaften, in denen das Kalibrieroffset und die MAC Adresse des Quadrocopters persistiert sind. Der Zusammenschluss aus einem Quadrocopter und einem Sensor wird in ein virtuelles Device gekapselt. (Eigenwerk)

5.1.2 Eigenschaften und Fähigkeiten

Die Begriffe *Eigenschaft* und *Fähigkeit* werden in der Literatur für unterschiedliche Beschreibungen angewendet. In der Domäne von Roboterschwärmen werden algorithmische Eigenschaften wie etwa die Robustheit verteilter Systeme beschrieben [31, 65], in der Robotik für geometrische Eigenschaften [182, 283] und in der Modellierung von Prozessen werden Eigenschaften und Fähigkeiten häufig synonym betrachtet [203], wie bspw. das Messen als ausführbare Eigenschaft. Im Rahmen des Semantic Web werden hingegen RDF-Prädikate als Eigenschaft tituliert [63]. In diesem Abschnitt werden die beiden Begriffe detailliert untersucht und abschließend in eine Ontologie überführt.

Eigenschaften

Eine Eigenschaft beschreibt laut Duden ein *„zum Wesen einer Person oder Sache gehörendes Merkmal; charakteristische [Teil]beschaffenheit oder [persönliche, charakterliche] Eigentümlichkeit“* [73]. Im Beispiel des Experiments der Ausbreitungsmodellierung wurden bereits einige Eigenschaften gekennzeichnet, wie etwa die *Genauigkeit* als Beschaffenheit eines Sensors oder eine Präzisierung der Fähigkeit *Messen*. Für eine bessere Klassifizierung sind in Abbildung 5.3 zusätzliche Eigenschaften skizziert, um beispielsweise eine MAC Adresse einem Quadrocopter zuzuweisen, Kalibrierparameter zu speichern oder das Gewicht als Merkmal des konkreten Quadrocopters zu beschreiben. Eine in der Literatur übliche Möglichkeit, Eigenschaften zu Kategorisieren, ist das zu beschreibende Element zu spezifizieren, wie etwa geometrische, haptische oder optische

Eigenschaften [171, 182]. In der Domäne der selbstbeschreibenden Hardware hingegen werden folgende Kategorien unterschieden:

Klassen- und Instanzeigenschaften
Klasseneigenschaften beziehen sich auf generelle Aussagen, die nicht einem Individuum zugeschrieben werden. Ein Beispiel sind die in Abbildung 5.3 abgebildeten Eigenschaften (E), die sich nicht auf den konkreten Sensor oder Quadrocopter beziehen, sondern auf die Serie. Diese Eigenschaften finden sich auch in den Dokumentationen der Hardware wieder und ändern sich in der Regel nicht.

Instanzeigenschaften hingegen bilden konkrete Eigenschaften des Individuums ab, wie bspw. die MAC Adresse, die als eindeutiger Identifikator für einen Ethernetadapter verwendet wird (Ausnahmen im Mikrocontrollerbereich ausgenommen). Bei physischen Geräten ist die Unterteilung in Instanz- und Klasseneigenschaften noch gut nachvollziehbar, in den Bereichen der Objektorientierten Programmierung oder dem Semantic Web ist die Trennung durch die Betrachtungsweise der Domäne abhängig. So kann das Gewicht aus dem Beispiel als Attribut einer Klasse oder als Klasse mit dem Attribut SI Einheit und Wert abgebildet werden. Im Semantic Web hingegen werden RDF-Graphen verwendet, in denen Literale die Blätter der Graphen und Prädikate die Kanten bilden. Adaptiert man das Beispiel, können folgende Tripel vereinfacht gebildet werden: (Sensor, hat, Gewicht) oder (Sensor, Gewicht, „10“). Das letzte Tripel kann nicht mehr mit der SI Einheit verknüpft werden, da Gewicht als Kante auf ein Literal mit dem Wert 10 verweist.

Statische- und dynamische Eigenschaften
Statische Eigenschaften sind nicht lokal änderbar und können sowohl Instanzen als auch Klassen angehören, wie bspw. das Gewicht und die MAC Adresse in Abbildung 5.3. Diese Eigenschaften müssen nicht zwingend auf der Hardware bereitgestellt werden, sondern können durch Referenzen auf URIs verweisen. Dieses Vorgehen ist vor allem für statische Klasseneigenschaften geeignet, die von vielen Instanzen referenziert werden. Dadurch sinkt auch die Komplexität für Erweiterungen und Änderungen bei einer zentralen Referenz.

Dynamische Eigenschaften hingegen sollen zur Laufzeit änderbar sein und sind üblicherweise Instanzspezifisch. Als Beispiel wird in Abbildung 5.3 das Kalibrieroffset skizziert. Sensoren werden üblicherweise anhand einer Referenz kalibriert. Dafür kann bei Temperatursensoren beispielsweise kochendes oder Eiswasser verwendet werden[130], oder bei Gassensoren Kalibriergase [16]. Ausgehend von einem linearen Sensorverhalten können die Messpunkte bspw. durch eine lineare Regression [316] in die Parameter *Steigung* und *Offset* der Korrektur überführt werden. Gerade bei elektrochemischen Sensoren ist eine regelmäßige Kalibrierung durch die Materialermüdung nötig [176]. Durch die dynamische Eigenschaft *Kalibrieroffset* kann die Korrektur direkt am Sensor gespeichert und von Fähigkeiten verwendet werden. Dadurch wird eine manuelle Korrektur in der Nachbearbeitung der Messwerte des Experimentes vermieden und Reaktionen, bspw. durch einen sensorbasierten Flug anhand kalibrierter Messwerte ermöglicht. Durch die enge Bindung auf Instanzebene sollten dynamische Eigenschaften auf der Hardware selbst gespeichert werden.

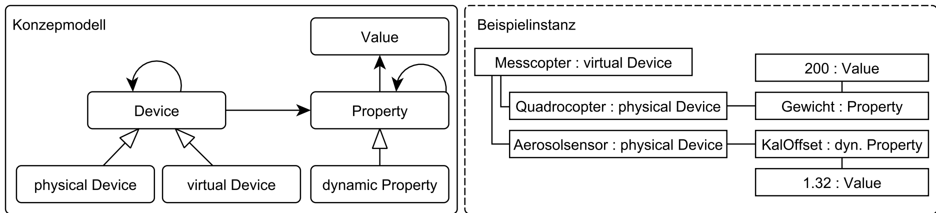

Abbildung 5.4. Das Konzeptmodell (links) illustriert die Abhängigkeiten zwischen Device und Property. Die Multiplizitäten der Assoziationen sind als 1 zu * zu interpretieren. Eine Beispielinstanz (rechts) zeigt den Quadrocopter mit Sensor aus Abbildung 5.3. (Eigenwerk)

Physische- und Virtuelle Eigenschaften
Physische Eigenschaften beziehen sich auf existierende Elemente, wie bspw. das Gewicht eines Quadrocopters. Der Zusammenschluss mehrerer realer Elemente wie ein an einem Quadrocopter befestigter Aerosolsensor führt zu einem kombinierten Gerät, das als virtuelles Device definiert werden kann.

Virtuelle Eigenschaften beschreiben die Eigenschaften eben solcher virtuellen Devices. So ist das Gesamtgewicht des virtuellen Devices die Summe der Gewichte aller physischen Elemente. Virtuelle Eigenschaften können analog zu dynamischen Eigenschaften aus Berechnungen hervorgehen und sind ebenfalls zur Laufzeit änderbar. So wird das Gesamtgewicht bei einer Rekonfiguration neu berechnet und über die virtuelle Eigenschaft bereitgestellt.

Die unterschiedlichen Einordnungen sind nicht als disjunkt zu betrachten. So ist bspw. das Gewicht des Sensors eine physische und statische Klasseneigenschaft, die auf der Hardware referenziert werden sollte und nicht geändert wird.

Abbildung 5.4 illustriert das Konzeptmodell (Domänenmodell) zwischen Geräten (Device) und Eigenschaften (Property). Zwar würde sich für die Selbstreferenz der Devices das Kompositum Entwurfsmuster [131] eignen, um die physischen Devices als Blätter zu definieren, allerdings wäre hier der Fall von Subsystemen ausgeschlossen. So ist der Quadrocopter ein physisches Device, das aus anderen physischen Devices, wie dem GPS-Modul besteht. Eine Klassifizierung zwischen Klassen und Instanzeigenschaften wird in dem Konzept noch nicht berücksichtigt, da eine potentielle Verteilung erst in der Architektur eine Rolle spielt. Die physischen oder virtuellen Eigenschaften hingegen beziehen sich auf den Typ des Devices, der über einer 1 zu * Assoziation verbunden ist. Die bereits in Abbildung 5.3 aufgezeigten Eigenschaften von Fähigkeiten, wie die Eigenschaft *Feinstaub* einer Fähigkeit *Messen*, erfolgt nach den oben aufgestellten Kriterien und wird im nächsten Abschnitt behandelt.

Fähigkeiten

Fähigkeiten werden in der Literatur häufig als Capability [32, 61] oder Skill [3, 129] synonymisch verwendet. Eine Fähigkeit (capability) bedingt nicht zwingend Erfahrung oder Vorkenntnis, die allerdings bei einer Fertigkeit (skill) vorausgesetzt sind. So verwendet Scholtz et al. [265] Fähigkeiten nur im Kontext von Robotersystemen und Fertigkeiten

nur in Bezug auf Menschen. Komplexe Greifstrategien basierend auf geometrischen Modellen und Erfahrungswerten vorangegangener Greifversuche werden in Knowrob [283] hingegen als Fertigkeit tituliert. Da sich die Fähigkeiten in der vorliegenden Dissertation allerdings auf ausführbare Codefragmente beziehen und aktuell nur Ergebnisse aus dem Bereich des Maschinellen Lernens verwendet werden, wie bspw. die Parametrisierung der PID Controller (siehe betreute Abschlussarbeiten), und nicht das Verfahren selbst, werden lediglich Fähigkeiten konzeptuell betrachtet, Fertigkeiten allerdings nicht gänzlich ausgeschlossen. Fähigkeiten können über unterschiedliche Wege implementiert und konzeptuell erfasst werden. In ROS können bspw. die Aufrufe über Topics als Fähigkeit interpretiert werden, wie bspw. das Greifen [283] oder eine einfache Point to Point Bewegung [204]. In von ROS unabhängigen Projekten hingegen befinden sich Aufrufe in Methoden, grafischen Benutzeroberflächen [238] oder anderen Schnittstellen wieder. Die Gemeinsamkeit aller Aufrufe ist die *Parametrisierung*, *Konfiguration* und die *Ablaufvariante*. Parameter bilden dabei ein kleines variables Set an Variablen, das üblicherweise bei jedem Aufruf geändert werden muss, wie etwas die Zielposition eines Quadrocopter [238]. Konfigurationen umfassen ein sich in der Regel nicht häufig änderbares Set an Parametern, wie etwa die Launchfile eines ROS-Projektes [157]. Die Grenzen und Mächtigkeiten hingegen bestimmen die Entwickler, so ist bspw. eine Parametrisierung eines Docker Containers entweder direkt über Parameter, oder in einer Konfigurationsdatei mit unterschiedlicher Syntax (Docker Compose) möglich [243]. Die Ablaufvariante bestimmt, wie eine Methode abläuft und was zu erwarten ist. In ROS finden sich bspw. drei Konzepte für die Realisierung. Publish Subscribe verhält sich analog zum Verhaltensmuster der Softwaretechnik, also eine asynchrone Registrierung auf ein Event, wie bspw. einem Sensorstream. Services werden für Abläufe mit kalkulierbarem Ende verwendet unter Einhaltung der Reihenfolge zwischen dem Senden und Empfangen der Nachricht. Actions erweitern Services um ein Feedback während des Ablaufes und werden vor allem für Abläufe verwendet, deren Fortschritt kritisch ist und überwacht werden sollte, wie bpsw. bei der Fähigkeit *Fliege zu Position* [157]. Eine äquivalente Realisierung in Methoden bildet dazu bspw. der Backgroundworker in .NET (Nativ, noch nicht in Mono) [320], der über einen asynchronen Aufruf gestartet wird und über Publish Subscribe-Mechanismen sowohl den Fortschritt, als auch die Antwortvarianten bereitstellt. In Semantic Plug & Play werden diese Konzepte für Fähigkeiten adaptiert und folgend zusammengefasst:

Fähigkeiten werden asynchron aufgerufen
Der initiale Aufruf einer Fähigkeit erfolgt asynchron analog zum Starten des Backgroundworkers in .NET, der einen neuen Thread für die Bearbeitung des zu definierenden Prozesses startet.

Publish-Subscribe dienen der Typspezifikation
Fähigkeiten bieten unterschiedliche Kategorien von Publishern an. Für kontinuierliche Messungen wird bspw. die Kategorie *Stream* verwendet, für indeterministische Abläufe kann die Kategorie *Process* gewählt und bei deterministischen Abläufen die Kategorie *Finish* verwendet werden.

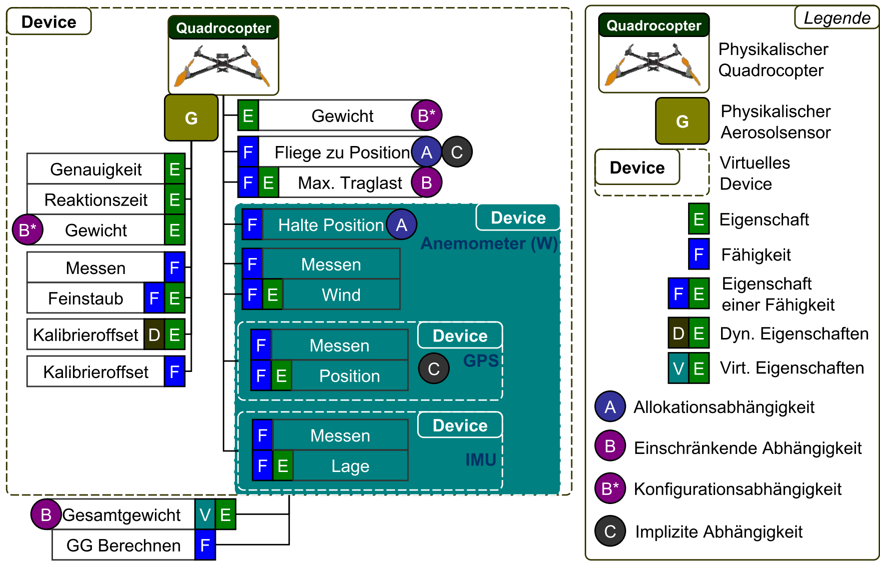

Abbildung 5.5. Erweiterung des internen Aufbaus aus Abbildung 5.3. Eigenschaften ohne Relevanz sind ausgelassen, um den Fokus auf die zusätzlichen Fähigkeiten zu legen. Das Prinzip der Kapselung in Devices ist auf das virtuelle Anemometer erweitert und Abhängigkeiten (A,B,B*,C) sind zwischen einzelnen Fähigkeiten hinzugefügt worden. (Eigenwerk)

Fähigkeiten können synchronisiert werden
Für eine einfache Verwendung synchroner Aufrufe wird die Asynchronität in Wrapper gekapselt und auf das Event der Kategorie *Finish* der Fähigkeit gewartet.

Mit dieser Kombination können sowohl Publish-Subscribe Implementierungen, als auch Services und Actions in ROS abgebildet werden sowie die Funktionalität des Backgroundworkers in .NET. Durch die freie Verwendung der Kombination bietet sich jedoch auch flexiblere Möglichkeiten in der Implementierung. Als Beispiel dient die *Fliege zu Position* Fähigkeit eines Quadrocopters. Diese kann über einen Wrapper synchron gestartet werden und erst bei dem Erreichen der parametrisierten Zielposition wird der Programmfluss fortgesetzt. Eine alternative bietet die asynchrone Verwendung, in der über die Publisher Kategorie *Process* die aktuelle Position übermittelt wird und ein neues Ziel kurz vor Erreichen der Zielpositionen definiert werden kann, um die negative Beschleunigung des Zielanfluges zu vermeiden. Dieses Prinzip der annäherungsweisen Bahnplanung ist in der Robotik als *Überschleifen* [8] bekannt. Welche Kategorien für eine Ausführung der Fähigkeit benutzt werden können wird anhand von Eigenschaften ausgedrückt. Als leitendes Beispiel für das Konzept der Eigenschaften und Fähigkeiten wird das Experiment der Ausbreitungsmodellierung in Abbildung 5.5 ergänzt und es werden unterschiedliche Abhängigkeiten zwischen den Fähigkeiten dargestellt.

- **Eigenschaften von Fähigkeiten**
  Die naheliegendste Ressource für die Beschreibung von programmierten Fähigkeiten ist die Orientierung an der Quellcodedokumentation von Methoden. So bieten

unterschiedliche Werkzeuge wie bspw. Javadoc ein Schema für die Beschreibung der Parameter, Funktionalität und Abhängigkeiten [184]. Die Methodenbeschreibungen anhand des Schemata werden wiederum in natürlicher Sprache und nicht maschinenlesbar formuliert. Als Teil der Selbstbeschreibung werden Parameter in eine maschinenlesbare und erweiterbare Form gebracht, um bspw. Fehler in der Datensemantik zu identifizieren oder die Typspezifikation einer Fähigkeit automatisiert mit den Subscribern zu verknüpfen.

Dynamische Eigenschaften benötigen eine Fähigkeit zur Änderung ihres Inhaltes, wie in Abbildung 5.5 das Kalibrieroffset. Analog verhalten sich virtuelle Eigenschaften, die erst durch die Kombination virtueller Elemente entstehen und üblicherweise auf einen Algorithmus aufbauen, wie im Beispiel des Gesamtgewichts.

- **Abhängigkeiten**
  Fähigkeiten werden in unterschiedlichen Granularitäten der Abstraktion verwendet. Typische Beispiele sind das Greifen eines Gegenstandes als einzelne Fähigkeit [283], oder primitive Fähigkeiten, wie eine Point to Point (PTP) Bewegung eines Roboters [129]. Anhand dieses Beispiels wird bereits eine Abhängigkeit zwischen Fähigkeiten offenbart, da das Greifen eines Gegenstandes unter anderem auf PTP-Bewegungen aufbaut. Im Beispiel aus Abbildung 5.5 benötigt die Fähigkeit *Fliege zu Position* das *Messen* des GPS-Devices, um die Fähigkeit zu synchronisieren, wenn das Ziel erreicht ist. Wird in einer Fähigkeit wie dem *Messen* des Windes im Anemometer hingegen die konkrete Position des Quadrocopters benötigt, so kann die implizite Fähigkeit, also das *Messen* der Position nachträglich separiert und angeboten werden. Dem Prinzip der Kombination unterschiedlicher Fähigkeiten um neue Fähigkeiten zu generieren, wie bspw. das Anemometer (W), stehen wiederum *einschränkende Abhängigkeiten* gegenüber, wie etwa das Gesamtgewicht, das bei Überschreitung der Eigenschaft *maximale Traglast* die Fähigkeit *Fliege zu Position* einschränkt (B).

- **Ressourcenallokation und Systemgrenzen**
  Eine besondere Abhängigkeit bildet die *Allokationsabhängigkeit* (A). So können die Fähigkeiten *Halte Position* und *Fliege zu Position* zumeist nicht parallel ausgeführt werden. Üblicherweise treten die Probleme der Allokationsabhängigkeit eher bei Fähigkeiten auf, die mit Aktuatorik in Verbindung stehen, können aber je nach Implementierung auch bei Sensoren zutreffen, bspw. wenn statt dem Publish-Subscribe Prinzip ein synchroner Request verwendet wird. Durch die freie Struktur der Device-Hierarchie könnten konzeptuell auch mehrere Quadrocopter zu einem virtuellen Device zusammengefasst werden, um beispielsweise kooperative Bewegungen zu realisieren. Dadurch steigt wiederum das Ressourcenallokationsproblem und eine neue Problemstellung durch Abhängigkeiten zwischen physikalisch getrennten Systemen entsteht. Für das Konzept der Abhängigkeiten zwischen Fähigkeiten werden deswegen nur physikalisch verbundene Devices betrachtet, eine Koordination und Prozessdefinition mehrerer Systeme hingegen ist Bestandteil der Multipotenten Systeme aus Abschnitt 3.2.

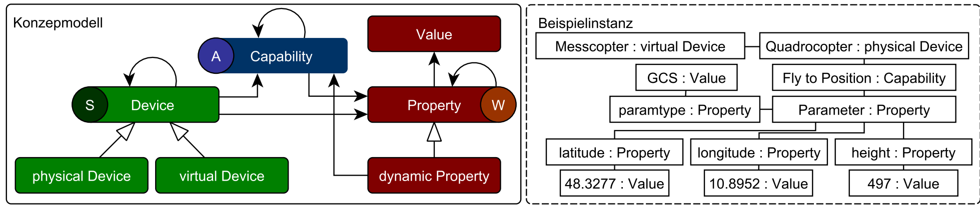

Abbildung 5.6. Konzept der Fähigkeiten in Kombination mit Eigenschaften (links). Auf der rechten Seite ist eine Beispielinstanz der *Fliege zu Position* Fähigkeit skizziert. Die Parameter werden über Eigenschaften ausgedrückt. Die Unterteilungen (W,S,A) stehen für die Zuordnung der Konzepte zur Struktur (S), Ausführung (A) und der Wissensbasis (W). (Eigenwerk)

Abbildung 5.6 zeigt, wie Fähigkeiten (Capability) im Konzeptmodell eingegliedert sind. Sowohl bei Devices, Capabilities als auch bei Properties sind durch diese Selbstreferenz Schleifenbildungen möglich. Das Ziel ist eine sehr offene Struktur, um den Aufwand für externe Migrationen aus dem Semantic Web zu erleichtern. So können die in Abbildung 5.6 skizzierten Parameter der *Fliege zu Position* Fähigkeit über die *Parameter* mit *paramtype* instanziiert werden, oder direkt an den einzelnen Eigenschaften, wie bspw. *latitude*. Neben der Struktur ist auch der Umfang der Informationen relevant. Werden die RDF-Graphenstrukturen aus den Kapitel 4 betrachtet, so können riesige Graphen wie die LOD-Cloud entstehen. Die Wissensbasis (W) wird im Konzept aus Abbildung 5.6 über die Eigenschaften hergestellt, indem die Selbstreferenz der Eigenschaften (Property) für Prädikate verwendet wird und ein *Value* ein Literal abbildet. Eine grundlegende Hypothese der vorliegenden Dissertation für die Wissensbasis ist:

> **Nur die domänenspezifische Anwendung kann entscheiden, welche Informationen benötigt werden. Die externe Struktur der Informationen ist für die Interpretation relevant und sollte durch einfache Mechanismen zugänglich sein.**

Fähigkeiten müssen mit ausführbaren Codefragmenten in Beziehung gebracht werden. Eine automatisierte Vorschrift über die Abhängigkeiten kann nicht getroffen werden, da die Zusammenstellung der Hardware und die domänenspezifische Interpretation ausschlaggebend sind. Als Beispielannahme sei die Fähigkeit *Fliege zu Position* synchron und ohne Publisher für die Position implementiert. So könnten potentielle Kollisionsvermeidungsstrategien, die einen Einfluss auf die Trajektorie haben, nicht realisiert werden. Fähigkeiten sind immer an eine Ausführung gekoppelt, die als Artefakte, also aus kompilierten Codefragmenten und deren Abhängigkeiten, über eine Referenz in der Wissensbasis assoziiert sind. Für eine Eingrenzung werden zwei Arten betrachtet, einerseits Interpretersprachen, die zur Laufzeit interpretiert werden können, und andererseits Container mit beliebigem Inhalt, wie in Kapitel 4 vorgestellt. Daraus leitet sich die zweite Hypothese des Konzepts ab:

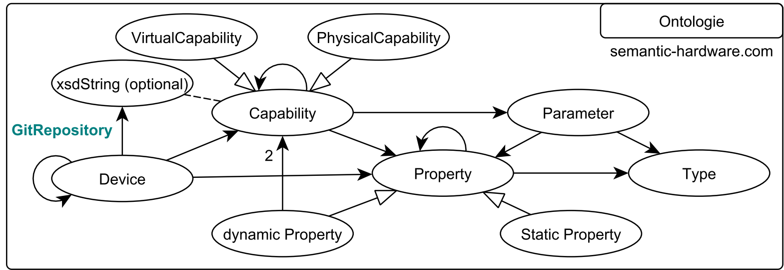

Abbildung 5.7. Ontologie des Semantic Hardware Web (SHW) mit der Grundstruktur des Konzeptes und zusätzlichen Konzepten für die Einschränkungen der Verwendung. Das Strukturelement der Devices wird ebenfalls verwendet, um auf GIT-Repositories zu verweisen. Die komplette Ontologie ist online verfügbar. (Eigenwerk)

> **Die Bereitstellung von Artefakten ist ein Teilprozess solider Softwareentwicklung. Die Verteilung und Verwendung kann durch die Integration in eine gemeinsame Wissensbasis automatisiert werden. Eine maschinenlesbare Aufbereitung der Quellcodedokumentation bietet eine Grundlage für den Abgleich domänenspezifischer Anforderungen und der verfügbaren Fähigkeiten.**

Das Konzept der Devices bietet eine einfache Strukturkomponente innerhalb der Implementierung sowie für die Bündelung von Fähigkeiten eines Artefaktes.

5.1.3 Wissensbasis aufbauend auf Ontologien

Die Struktur der Wissensbasis wird durch eine Ontologie (Auszüge in Abbildung 5.7) abgebildet und baut auf der Grundstruktur des Konzeptmodells auf. Die komplette Ontologie ist im Rahmen der Dissertation entstanden und steht online[1] zur Verfügung. Die Grundidee ist die Trennung der erweiterbaren Wissensbasis durch Semantic Web Technologien von der Domänenrepräsentation innerhalb objektorientierter Programmiersprachen. Das Konzeptmodell dient dabei als gemeinsame Grundlage, die bereits durch die Konzepte der Parameter und Typen in der Ontologie erweitert werden. Auf der objektorientierten Seite hingegen wird in Kapitel 6 ein Design-Klassendiagramm, sowie eine Schichtenarchitektur für die Anbindung der Hardware vorgestellt. Das Grundprinzip ist die Abstraktion in eine einfache Klassenstruktur auf der Programmierseite, die mit Anforderungen instanziiert werden kann. Als Beispiel dient ein Quadrocopter mit der Fähigkeit *Fliege zu Position* in Abbildung 5.8. Dieses Modell ist noch nicht an ausführbare Fähigkeiten gebunden und es ist ungewiss, ob die konkrete Instanziierung der Ontologie anhand von Individuals in einem RDF-Graphen (Semantisches Netz) der Hardware die Anforderungen (Requirements) erfüllen kann. Können die Anforderungen erfüllt werden, wird die Instanz mit den nötigen Verknüpfungen sowie den Basiseigenschaften, wie die Parameterbeschreibung, für die Ausführung der Fähigkeit befüllt und steht der objektorientierten Programmiersprache zur Verfügung. Das semantische

[1] semantic-hardware.com

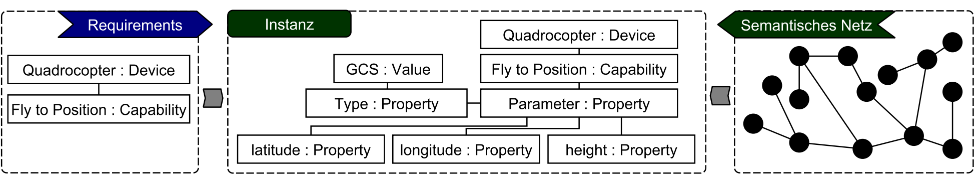

Abbildung 5.8. Trennung des Konzeptmodells in eine objektorientierte Struktur, die anhand instanziierter Klassen Anforderungen (Requirements) definiert, und eines semantischen Netzes, das die Wissensbasis darstellt. Die konkrete Instanz in der Mitte wird anhand der Requirements und den Informationen des semantischen Netzes gebildet. (Eigenwerk)

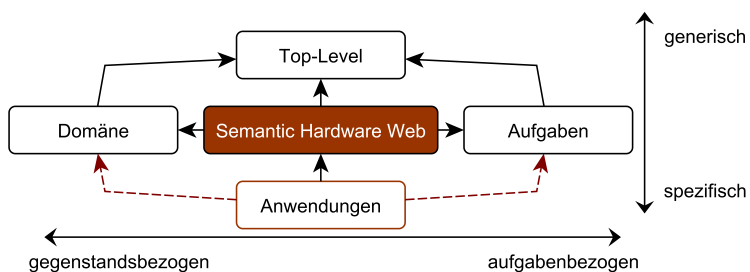

Abbildung 5.9. Verortung der Semantic Hardware Web (SHW) Ontologie in das Schema von Rehbein et al. [241] aus Abbildung 4.9. Durch die Schnittstellen zu den Verknüpfungen zur Domäne sowie der Integration von Aufgaben durch Fähigkeiten befindet sich die SHW Ontologie im Zentrum. Über die SHW und die Konzepte von Semantic Plug & Play müssen Anwendungen keine eigene Ontologie besitzen. (Eigenwerk in Anlehnung an Rehbein et al. [241])

Netz basiert auf den Linked Data Prinzipien und wird anhand der Ontologie aufgebaut. So ist jeder Knoten ein Subjekt mit eindeutiger HTTP-URI und im Spezialfall der Devices kann diese HTTP Uri eine Referenz auf ein GIT-Repository sein. Das Prinzip ist die Verwendung des Strukturelements der Devices als Kapselung für Fähigkeiten. Die Ontologie wird jedoch nicht unterteilt in physische und virtuelle Devices, da es für die Bereitstellung der Codebasis irrelevant ist. Das semantische Netz muss für die Interpretation auf der Semantic Hardware Web Ontologie (SHW) aufbauen, damit eine Interpretation möglich ist. Dennoch soll sie durch konkrete Ontologien, wie bpsw. die Sensor Network Ontologie [300], erweitert werden können. In Abbildung 5.9 wird die Ontologie in das Schema von Rehbein et al. [241] eingeordnet. So können bspw. die SI Einheiten durch die UnitDim Ontologie [249] in der Domäne angewendet werden oder die Aufgaben-zentrierten Messwerte mit der Sensor Network Ontology (SSN) [300]. In der Instanz werden externe Ontologien unter den Eigenschaften aufgeführt, die über Mappingmechanismen mit Suchfunktion zugeordnet werden. Das Mapping zwischen den Anforderungen und der freien Wissensbasis ist eine Herausforderung, der sich das Kapitel 6 widmet. Abhängigkeiten, wie sich gegenseitig ausschließende Fähigkeiten oder subtraktive Fähigkeiten, werden der Anwendung zugeteilt und in der SHW

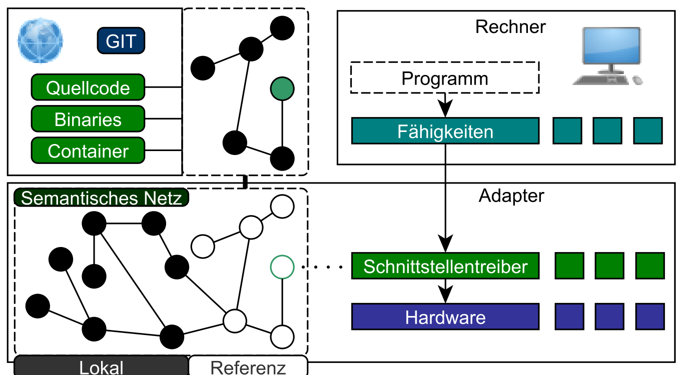

Abbildung 5.10. Zugriffshierarchie über adaptierte Hardware (rechts). Treiber bilden die Basis für Fähigkeiten, die im Programmcode verwendet werden können. Teile des semantischen Netzes können auf dem Adapter bereitgestellt und durch Referenzen im Internet komplettiert werden (links). (Eigenwerk)

nicht beachtet. Abhängigkeiten zwischen simulierten und realen Objekten werden im Rahmen der modularen Simulation in Abschnitt 5.3.2 vorgestellt. Die Referenz auf ein GIT-Repository für Fähigkeiten über Devices ist nur ein kleiner Teil der Codeintegration, die im folgenden Kapitel detailliert vorgestellt wird.

5.1.4 Integration ausführbaren Codes

Das Konzept der Abstraktion von Fähigkeiten ist, dass im Programm nicht mehr gegen eine API oder gegen Schnittstellen implementiert werden muss, sondern dass sich diese selbst anhand der Wissensbasis beschreiben, wie in Abbildung 5.10 illustriert. Fähigkeiten werden über Devices gekapselt und innerhalb der Wissensbasis durch Artefakte mit GIT-Repositories assoziiert. Dabei werden zwei Fälle betrachtet. Die Schnittstellentreiber befinden sich bereits auf dem Adapter und können direkt angesprochen werden, oder sie sind nicht vorhanden und eine Referenz verweist auf ein GIT-Repository, so müssen die Referenzen aufgelöst und heruntergeladen werden. GIT-Repositories sind hauptsächlich für die Versionierung und Bereitstellung von *Quellcode* bekannt. Dieser muss auf dem Adapter üblicherweise kompiliert und Abhängigkeiten müssen installiert werden. Konzeptuell können bei der Quellcodevariante nur interpretierte, höhere Programmiersprachen berücksichtigt werden, die wie Python eine eigene Paketverwaltung (pip) für die Installation von Abhängigkeiten verfügt. Für hardwarenahe Implementierungen hingegen werden spezielle APIs in Kapitel 6 für die Integration bereitgestellt.

Für *Binaries* variieren bei unterschiedlichen Betriebssystemen auch die Installationen. So werden bspw. in Debian-Pakete Abhängigkeiten durch einen zentralen Paketmananger aufgelöst. In Windows hingegen müssen Binaries oft über eigene Installationsroutinen die nötigen Abhängigkeiten bereitstellen. Die Betriebssystemabhängigkeit besteht bei Binaries allerdings in den meisten Fällen. Die Bereitstellung von Binaries wird in GIT über CI CD Pipelines, wie in Abschnitt 4.2.5 beschrieben, üblicherweise für unterschied-

liche Betriebssysteme unterstützt. Da es jedoch in GIT kein vorgefertigtes Schema für Betriebssysteme gibt und zahlreiche Varianten möglich sind, wird auf Binaries und eigene Installationsroutinen in der vorliegenden Dissertation verzichtet. Eine alternative, Betriebssystem unabhängige Lösung für die Bereitstellung von Binaries bieten hingegen *Docker Container*, die über eine Parametrisierung gestartet werden und eine Kommunikation über Ports erlauben.

Die Eigenschaften einer Fähigkeit referenzieren auf den entsprechenden Docker Container oder den Quellcode. Dabei sind potentielle Abhängigkeiten zu anderen Fähigkeiten ebenfalls in den Eigenschaften referenziert und eine Auflösung der Abhängigkeiten führt nicht zum Herunterladen der Artefakte. So können analog zum Template Method Entwurfsmuster der Gang of Four [131] „abstrakte" Fähigkeiten in der Prozessstruktur ohne konkrete Implementierung, wie bpsw. das Anemometer aus Abbildung 5.5 verwendet werden. Für die Messung der Windrichtung und -stärke werden die Fähigkeiten *Halte Position* und *Messen* mit der Eigenschaft *Lage* kombiniert. Die Fähigkeit *Halte Position* hingegen benötigt wiederum die Fähigkeit *Messen* mit der Eigenschaft *Position*. Erst bei einer erfolgreichen Auflösung werden die entsprechenden Artefakte zur Laufzeit heruntergeladen und gestartet. Zusammengefasst ergibt sich folgende Hypothese des Konzeptes zu Fähigkeiten:

> **Fähigkeiten können aufeinander aufbauen, um redundanten Code zu vermeiden. Durch eine maschinenlesbare Dokumentation können auch Änderungen zur Laufzeit berücksichtigt werden, wie im Beispiel einer Rekonfiguration des Systems. Die Verwendung des De-facto Standards GIT in Kombination mit Testverfahren und ggf. Docker Container ist die Grundbedingung für die Bereitstellung der Fähigkeiten.**

Innerhalb von Fähigkeiten wird eine Prozesslogik implementiert, die auf andere Fähigkeiten zurückgreifen kann. Dadurch wird die Trennung des domänenspezifischen Programms und einer Fähigkeit unscharf, da auch der im Programm definierte Prozess als Fähigkeit interpretiert werden kann. So kann die Implementierung des virtuellen Anemometers sowohl als Prozess, aber auch als Fähigkeit definiert werden. Als Richtlinie für die Kapselung von Fähigkeiten ist die Wiederverwendbarkeit anzuführen. Für die Trennung und eine Verwendung externer Mechaniken wird die Modellierung von Fähigkeiten auf Konzeptebene im Detail betrachtet.

Das Konzept der Eigenschaften und der Trennung zwischen Wissensbasis und ausführbaren Codeelementen ist in seiner ersten Iteration in zwei Publikationen [84, 309] vorgestellt. Die aktuelle Ausprägung des Semantic Hardware Web findet sich konzeptuell in einem Übersichtspapier in Kombination mit Roboterschwärmen [153], sowie der konkreten Realisierung in einer Hausautomatisierung [311].

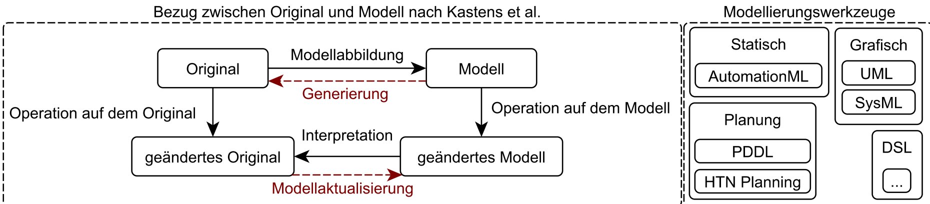

Abbildung 5.11. Abhängigkeiten zwischen Modell und Original mit Änderungen nach Kastens et al. [138], sowie exemplarische Modellierungswerkzeuge. Die roten, gestrichelten Linien sind Ergänzungen für die modellgetriebene Entwicklung. (Eigenwerk in Anlehnung an Kastens et al. [138])

5.2 Fähigkeiten modellieren

Der Begriff *Modell* wird auch außerhalb der Informatik als Abstraktionsmittel verwendet, um Sachverhalte einer Domäne präziser zu beschreiben. Als Beispiel sei eine Bedienungsanleitung für den Zusammenbau von Möbeln eines beliebten Möbelhauses aufgeführt. Textuelle Elemente beschreiben die Prozessreihenfolge und abstrakte Modelle der Originalteile dienen dem Verständnis für geometrische Abhängigkeiten. Dieses Zusammenspiel unterschiedlicher Modellierungen zur Beschreibung eines großen Ganzen wird ebenfalls im Fachgebiet der Informatik praktiziert. Eine Definition des Begriffs *Modell* in der Domäne der Informatik wird nach Kastens et al. wie folgt definiert:

> *„Die Modelle in der Informatik bezeichnen im Allgemeinen abstrakte Abbilder oder Vorbilder zu konkreten oder abstrakten Originalen“ [138].*

Wie in Abbildung 5.11 illustriert, führen Änderungen auf der Modell- und Originalseite auch zur gegenseitigen Aktualisierung, bzw. Interpretation. In Ergänzung zu Kastens et al. werden auch modellgetriebene Ansätze verfolgt, die den Begriff des Originals bspw. als Codebasis betrachten. Modellierungssprachen wie die Unified Modeling Language (UML) oder deren Erweiterung, die Systems Modeling Language (SysML), unterteilen die Modellierung in dynamische (bspw. Aktivitätsdiagramme) und statische Aspekte (bspw. Verteilungsdiagramme) der Betrachtung. Verwendet werden nicht nur grafische Modellierungen, sondern auch textuelle, wie bspw. mit Schemata für Freitexte in Kontrakten oder mit spezifizierter Syntax und Semantik durch die Object Constraint Language (OCL). Die Modelle der UML dienen auch als Vorlage für unterschiedliche, domänenspezifische Realisierungen, wie etwa bei der Modellierung von Roboterskills über statische Komponentenmodelle [234] oder dynamische Aktivitätsdiagramme [122]. Hybride Modellierungsformen wie die Erweiterung statischer Modelle durch dynamische Aspekte zur Laufzeit finden sich bspw. in Matlab Simulink [229]. Falls jedoch eine Modellierungssprache den domänenspezifischen Anforderungen nicht gerecht oder eine zusätzliche Abstraktion benötigt wird, werden üblicherweise domänenspezifische Sprachen (DSL) eingeführt [213].

Ein exemplarischer Ansatz für die Modellierung von Skills mit unterschiedlichen Modellierungsebenen findet sich in der Arbeit von Pfrommer et al. [228], die in Abbil-

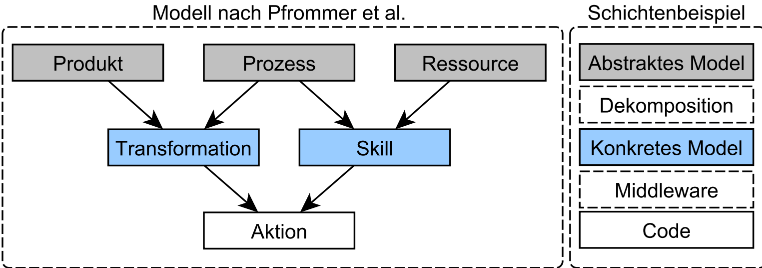

Abbildung 5.12. Links ist das Modell der Abhängigkeiten von Skills nach Pfrommer et al. [228] dargestellt. Rechts findet sich eine exemplarische Zuordnung der Modelleinordnung. (Eigenwerk in Anlehnung an Pfrommer et al. [228])

dung 5.12 auf der linken Seite dargestellt ist. Für die statische Modellierung von Skills wird AutomationML [69] verwendet, während für die Ausführung eine Art rudimentäre Zustandsmaschine für die Orchestrierung einer Servicestruktur vorgestellt wird. Die konkrete Umsetzung der Skills über die Middleware der Servicestruktur hingegen bleibt auf abstraktem Niveau, ebenso wie eine potentielle Rekonfiguration der Maschinen, die sowohl statische und dynamische Aspekte umfasst. Zusammengefasst wird für die Modellierung folgende These aufgestellt:

> **Die Komplexität großer Anwendungen erfordert abstrakte und zusammenhängende Modelle, die bis zum ausführbaren Code reichen. Die Trennung zwischen Domäne und Fähigkeiten ist fließend. Das Programmierparadigma der anforderungsbasierten Entwicklung für rekonfigurierbare Hardware hingegen erfordert auch Anpassungen der eingesetzten Middleware und der darauf aufbauenden Modelle.**

Als leitendes Beispiel wird das Experiment der synchronen Glasfasermessung verwendet, bei dem ein heterogenes Ensemble zum Einsatz kommt. Auf Basis des Beispiels werden die Herausforderungen rekonfigurierbarer Systeme untersucht und es wird eine Programmierung mit domänenspezifischen Anforderungen vorgestellt. Danach wird die Middleware ROS für Rekonfigurationen durch eine DSL erweitert und ein Prinzip für die Prozessmodellierung innerhalb von ROS vorgestellt. Abschließend wird ein Konzept für die Verteilung der Fähigkeiten und Eigenschaften auf reale Hardware vorgestellt.

5.2.1 Experiment: Synchrone Glasfasermessung

Das Experiment der synchronen Glasfasermessung dient der Aufnahme horizontaler Temperaturprofile durch ein Lichtwellenleiterkabel (LWL-Kabel), welches an einer Auswertungseinheit (DTS) angebracht ist, wie in Abschnitt 3.3.2 beschrieben. Für den Aufbau werden zusätzlich Temperatursensoren direkt am Quadrocopter zur Referenzmessung angebracht, wie in Abbildung 5.13 skizziert. Für eine Ausfallsicherheit oder zur Vermeidung von Messungenauigkeiten werden zwei Sensoren mit vorheriger Kalibrierung verwendet. Der Quadrocopter ist analog zu dem in Abschnitt 4.2.2 vorgestellten

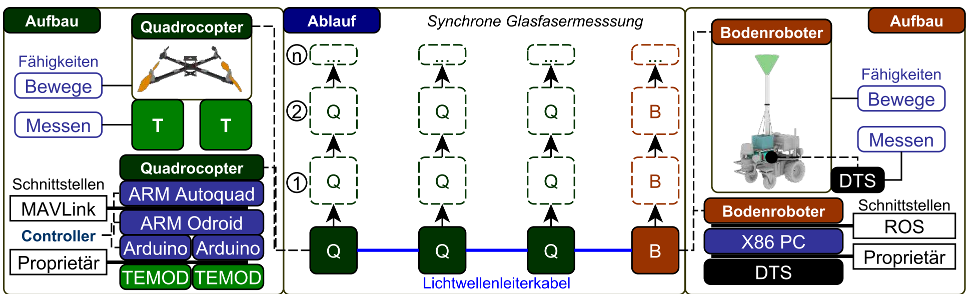

Abbildung 5.13. Aufbau und Ablauf des Experiments der synchronen Glasfasermessung. Der Ablauf wird aus der Vogelperspektive betrachtet und die Bewegung in einzelne Interpolationspunkte unterteilt (1,2,n). Die Fähigkeiten werden weiter abstrahiert in *Bewege* statt *Fliege zu Position*, damit die Fähigkeit auch für den Bodenroboter eingesetzt werden kann. Der Hardwareaufbau dient einer Verortung der Softwareartefakte. (Eigenwerk)

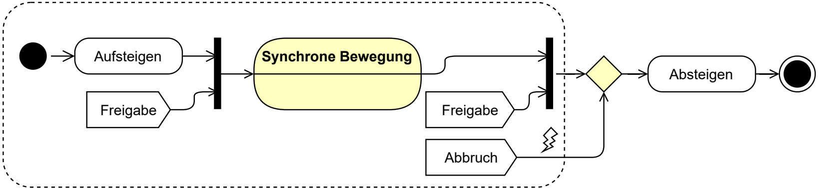

Abbildung 5.14. UML-Aktivitätsdiagramm für den Kontrollfluss eines Quadrocopters im Experiment auf abstrakter Ebene. Der Unterbrechungsbereich (gestrichelt) kann durch das Ereignis *Abbruch* extern verlassen werden. Ereignisse sind globale Aktionen und können von einer Kontrollstation an alle Quadrocopter gesendet werden. Das *Aufsteigen* und *Absteigen* ist bewusst als Aktion modelliert, da ein detaillierteres Modell nicht vorgesehen ist. (Eigenwerk)

aufgebaut, lediglich der Temperatursensor ist durch ein PT1000 an einer TEMOD Auswertungseinheit mit I2C Schnittstelle ersetzt worden. Das MAVLink Protokoll [156] zwischen dem ARM Autoquad des Quadrocopters und dem ARM Odroid ist ein Open-Source-Protokoll für die Kommunikation mit Quadrocoptern. Auf der rechten Seite der Abbildung 5.13 wir der Aufbau des Bodenroboters illustriert. Auf dem Bodenroboter selbst befindet sich ein X86 Personal Computer, der über die Middleware ROS eine Ansteuerung der Sensoren und Aktuatoren des Bodenroboters ermöglicht. Die Auswertungseinheit (DTS) hingegen verwendet eine geschlossene, proprietäre Schnittstelle mit eigener Software. Das Interessante an dem Aufbau ist die Diversität der Hardware sowie die nötige Vereinheitlichung für den Ablauf der Messung.

Der Ablauf der Mission erfolgt auf mehreren Ebenen und wird in einer sehr abstrakten Variante in Abbildung 5.14 skizziert. Bedingt durch das fragile LWL-Kabel zwischen den Quadrocoptern und dem Bodenroboter ist ein automatisierter Auf- und Abstieg nicht praktikabel, da zu viele externe Einflüsse zu beachten sind. So muss darauf geachtet werden, dass sich das Kabel nicht im Gras verfängt oder die Abwicklung auf dem Bodenroboter keine Schleifen im Kabel bildet. Eine Freigabe der Piloten erfolgt, wenn eine stabile Position des Ensembles gefunden wurde, bzw. wenn die synchrone Bewegung

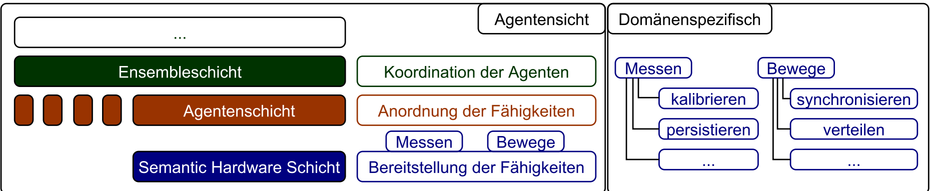

Abbildung 5.15. Die Schichtenarchitektur aus Abschnitt 3.2 angewendet auf das Experiment, mit einfacher Schnittstelle der für Agenten relevanten Fähigkeiten. Die internen Abläufe mit separaten Fähigkeiten sind im rechten Teil des Diagramms angedeutet. (Eigenwerk)

abgeschlossen ist. Durch einen externen Abbruch kann bei Fehlverhalten die Kontrolle an die Piloten übergeben und auf die Freigabe verzichtet werden. Die Koordination der einzelnen Schwarmelemente des Ensembles übernimmt die Ensembleschicht der Architektur Multipotenter Systeme aus Abschnitt 3.2, wie in Abbildung 5.15 skizziert. Die Agenten der Agentenschicht hingegen ordnen die bereitgestellten Fähigkeiten, in diesem Fall die abstrakte Fähigkeit *Bewege* und das initiale *Messen*, an. Durch die Unterschiede der Schnittstellen ist vor allem die Fähigkeit *Bewege* eine Herausforderung und erfolgt über eine Interpolation der Strecke in mehrere Zielpunkte, wie in Abbildung 5.13 illustriert. Die konkreten Abläufe, wie auch die Persistierung der Messwerte zur Laufzeit oder die vorherige Kalibrierung der Sensoren ist für die Agentenschicht nicht relevant, dennoch domänenspezifisch. Für diese Fähigkeiten und den inneren Ablauf werden im Folgenden Modelle vorgestellt, die nicht nur direkt mit Code assoziiert sind, sondern auch ROS für rekonfigurierbare Hardware erweitert und eine textuelle sowie grafische Modellierung in ROS erlaubt.

5.2.2 Prozesslogik

Die Trennung der Prozesslogik von den Fähigkeiten erfolgt nach Pfrommer [228] durch die Wiederverwendbarkeit einer Fähigkeit und der domänenspezifischen Implementierung eines Prozesses. Diese Trennung ist jedoch nicht scharf, wie die vorgestellten Experimente illustrieren. Da bis zur synchronen Glasfasermessung lediglich homogene Quadrocopter-Ensembles betrachtet wurden, ist die Fähigkeit mit impliziter Semantik *Fliege zu Position* wiederverwendbar. Bedingt durch den Bodenroboter wird eine weitere Abstraktionsstufe eingeführt, die Fähigkeit *Bewege* sowie eine Verlagerung der Semantik in die Parametrisierung. Dadurch werden jedoch nicht die Probleme der synchronen Bewegung gelöst. So ist bspw. die Fähigkeit *Bewege* für den Bodenroboter mittels Trajektorienplanung in ROS gelöst. ROS bietet die Möglichkeit nicht nur das Ziel und die Maximalgeschwindigkeit anzugeben, sondern auch den Weg und die Dauer einzuschätzen, was auch von der Orientierung zur Startzeit des Bodenroboters abhängt, da der Bodenroboter sich nicht omnidirektional bewegen kann. Der Quadrocopter hingegen kann sich omnidirektional bewegen und basiert auf einer reglerbasierten Implementierung mit Ziel- und Geschwindigkeitsvorgabe ohne Trajektorienplanung. Die Art der Synchronisierung erfolgt über die Anforderungen aus der Domäne. Eine präzise Synchronisierung erfordert mitunter die Erweiterung der Drohnensteuerung, wie bspw.

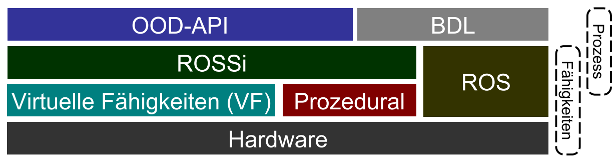

Abbildung 5.16. Technologiestack für die Implementierung und Modellierung von Fähigkeiten. Prozesse können sowohl in objektorientierten Programmiersprachen, als auch mit ROS oder den Erweiterungen ROSSi und der Block Definition Language (BDL) implementiert werden. Der Stack mit seinen optionalen Schichten ist von oben nach unten zu lesen, so kann ein durchgängiger Ansatz bspw. rein objektorientiert erfolgen. (Eigenwerk)

durch die Trajektorienplanung der ETH Zürich [121] mit der dazu passenden Hardware. Ungenauere Varianten lassen sich bspw. mit einer Interpolation und Synchronisierungspunkten in Kombination mit Anpassungen der Beschleunigungs- und Geschwindigkeitsparameter realisieren. Das Experiment verwendet die ungenauere Variante, da durch das LWL-Kabel Positionsabweichungen möglich und durch die unpräzise GPS Steuerung der Quadrocopter auch unvermeidbar sind. So ist auch bei Pfrommer [228] das *Drilling* als Fertigkeit (skill) definiert. Sowohl die eingesetzte Hardware als auch die Domäne bestimmen jedoch den intrinsischen Prozess und die Parametrisierung der Fähigkeit.

Eines der Ziele von Semantic Plug & Play ist eine Modellierung von Fähigkeiten mit domänenspezifischen Anpassungen und Integrationsmöglichkeiten für vorhandene APIs oder Middleware wie das Robot Operating System (ROS) . Der Technologiestack aus Abbildung 5.16 illustriert, welche Varianten möglich sind, um Fähigkeiten und Prozesse zu modellieren bzw. zu implementieren. Die Schicht (OOD-API) bildet eine Prozessdefinition in unterschiedlichen objektorientierten Programmiersprachen mit Rekonfigurationsmöglichkeiten durch die Verwendung von Anforderungen. ROSSi und die Block Definition Language (BDL) sind zwei Erweiterungen für die Modellierung von Prozessen und Fähigkeiten für ROS, die im Rahmen der vorliegenden Dissertation entstanden sind. Die prozedurale Entwicklung bezieht sich auf eine Schnittstellen-API für die Integration beliebiger Sensoren.

Objektorientierte Prozesse

Die Prozessmodellierung findet auf der Ebene der Agenten, wie in Abbildung 5.15 dargestellt, statt. Durch die Verwendung von Anforderungen und der Abstraktion von Fähigkeiten können einfache Codestrukturen verwendet werden. Folgender Pseudocode dient der Veranschaulichung des Konzeptes der objektorientierten Prozesse der OOD-API auf einem Quadrocopter.

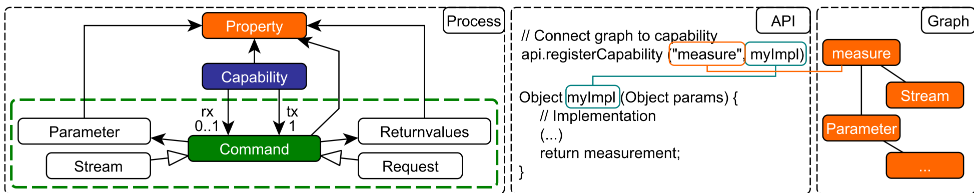

Abbildung 5.17. Erweiterung des Konzeptmodells aus Abbildung 5.6 um Commands für die Ausführung von Fähigkeiten (links). In der Mitte ist Pseudocode für die Implementierung eines Sensors auf der Hardwareseite und die dazu assoziierten Graphen aus Eigenschaften (rechts) skizziert. (Eigenwerk)

```
// Define Requirements
Device myCopter = new Device(new Cap("moveTo", "fly"),
          new Cap("measure", "temperature"));

// Wait till Device is ready or configured
while (!myCopter.instanciated)
  Sleep(500);

// Command Position to one Meter height
myCopter.getCap("moveTo").trigger(new Parameter[]{
    new Parameter(0), new Parameter(0), new Parameter(1)});
```

Die Instanz des Devices wird anhand der zwei Fähigkeiten (*moveTo*, *measure*) gesucht, die durch die Eigenschaften (*fly*, *temperature*) näher beschrieben sind. Über die Schleife in Zeile 6 wird bspw. auf eine Rekonfiguration gewartet. Die Ausführung der Fähigkeiten wird mit den Beispielparametern (0,0,1) gestartet. Dies bildet die Schnittstelle für die Agenten, deren abstrakte Fähigkeiten durch die Definition von Eigenschaften bestimmt werden können. Die Auflösung der natürlichen Sprache mit dem semantischen Netz der Eigenschaften ist Bestandteil der Architektur und wird in Kapitel 6 detailliert beschrieben.

Fähigkeiten auf Codeebene

Das Gegenstück für die Definition des Prozesses ist die hardwarenahe Implementierung der Fähigkeiten bspw. auf einem Mikrocontroller oder einem Einplatinencomputer. Diese Ebene wird in Abbildung 5.16 als prozedural, sowie virtuelle Fähigkeiten für Sprachen mit Objektorientierung skizziert. So ist es möglich, mit einer Kombination aus objektorientierten Prozessdefinitionen und prozedural implementierten Fähigkeiten zu arbeiten. Der erste Schritt für eine generische Schnittstelle für Fähigkeiten ist die Erweiterung des Konzeptes aus Abbildung 5.6 um eine Befehlsschnittstelle (Command) mit Assoziationen für das Senden- (tx) und Empfangen (rx) von Nachrichten. In Anlehnung an die Mechanismen aus ROS [157] werden Commands unterteilt in Anfragen (Requests) und Datenstrom (Stream), wie in Abbildung 5.17 abgebildet. Eine Antwort

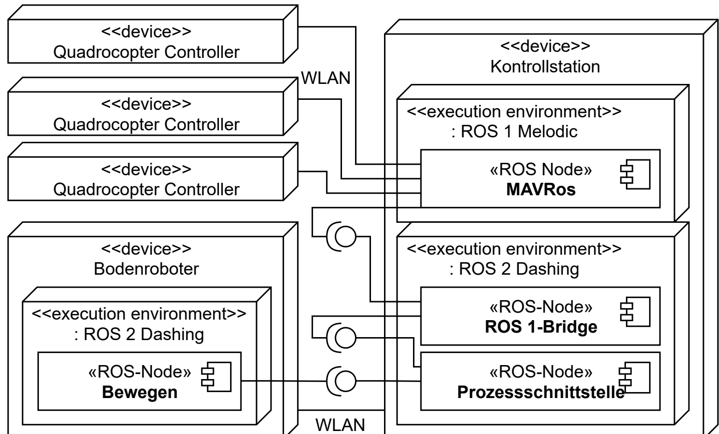

Abbildung 5.18. UML-Deploymentdiagramm in Kombination mit einem Komponentendiagramm für die Verteilung der ROS-Nodes. Der Umfang beschränkt sich auf die Realisierung der Fähigkeit *Bewegen*, das über MAVRos an den Flugcontroller der Quadrocopter geschickt wird. (Eigenwerk)

über rx ist sowohl bei Requests als auch bei Streams optional. Die Parameter und potentielle Rückgaben (Returnvalues) können wie auch die Fähigkeit selbst über Eigenschaften beschrieben werden. Eine exemplarische Implementierung einer Fähigkeit innerhalb der Schnittstellen-API ist in der Mitte von Abbildung 5.17 skizziert, in der die Commands in Methoden verarbeitet werden und eine Kopplung zwischen den Fähigkeiten und Eigenschaften erfolgt. Das Grundprinzip einer Auslagerung der Eigenschaften in ein semantisches Netz (Graph) wird nicht verletzt, da lediglich der Methodenname mit dem Namen der Fähigkeit assoziiert wird. Die Art der Methode, also ob ein Stream oder ein Request benötigt wird, erfolgt auf der Prozessdefinitionsebene, in der ebenfalls ROS verwendet werden kann.

ROS-Erweiterungen

Das *Robot Operating System* (ROS) befindet sich aktuell in einer Übergangsphase aus der Version 1 in die Version 2, mit weitreichenden Umstellungen und einer Abwärtsinkompatibilität. Für *ROS 1* steht eine breite Palette an externen Paketen zur Verfügung und die statischen Launchfiles werden über eine statische XML-Syntax definiert. *ROS 2* hingegen bietet Python für die Launchfiles sowie aktualisierte Abhängigkeiten (bspw. Python 3 statt 2) und die Unterstützung harter Echtzeit. Beide Varianten können jedoch über eine *ROS 1-Bridge* miteinander kommunizieren, wie in Abbildung 5.18 abgebildet. In diesem Beispiel wird das MAVLink Protokoll der *MAVRos-Node* für die Kommandierung der Quadrocopter verwendet, das in ROS 1 zur Verfügung steht. Die Launchfiles hingegen bestimmen die Infrastruktur der eingesetzten Knoten und müssen zur Designzeit definiert werden. Eine Rekonfiguration zur Laufzeit mit Änderungen der Struktur ist in keiner der Versionen vorgesehen.

Für Semantic Plug & Play werden die Konzepte der Prozessdefinition, Verpackung, Infrastruktur und Adapter vorgestellt. Die **Prozessmodellierung** findet nicht wie in

ROS üblich, in einem der Nodes statt sondern erfolgt extern in einer Domain Specific Language mit dem Namen *Block Definition Language* (BDL). Prinzipien der Selbstbeschreibung, wie die Bereitstellung über ein GIT-Artefakt oder die Kapselung in Container, gelten auch für ROS-Nodes, die durch eine Referenz in der Selbstbeschreibung **verpackt** werden können. Durch die Vorgabe der Infrastruktur vor der Ausführung sind Rekonfigurationen zur Laufzeit innerhalb von ROS nicht ohne Neustart möglich. So dient ROS der Verwaltung komplexer Systeme, die sich zur Laufzeit nicht ändern, wie etwa des Bodenroboters aus dem Beispiel der synchronen Glasfasermessung. Damit ROS und die OOD-API ineinandergreifen, ist ebenfalls eine Erweiterung der Infrastruktur nötig. Für die Generierung von Launchfiles innerhalb der BDL muss die **Infrastruktur** der ROS-Nodes modelliert werden. Dafür wird ein Modellierungstool mit dem Namen ROSSi vorgestellt, das eine statische Modellierung ermöglicht und diese mit dynamischen Aspekten zur Laufzeitüberwachung ergänzt. Damit auch ROS mit der OOD-API interagieren kann, um beispielsweise Fähigkeiten zu synchronisieren, werden **Adapter** vorgestellt, die wie im Beispiel der ROS 1-Bridge aus Abbildung 5.18 mit der OOD-API kommunizieren.

Mit diesen Konzepten soll sowohl eine Verzahnung mit ROS möglich sein als auch die Unabhängigkeit der OOD-API gewahrt werden. So ist der in Abbildung 5.16 aufgeführte Stack auch über beliebige Wege von oben nach unten innerhalb einer Anwendung nutzbar. Dadurch soll ROS für bestimmte Aufgaben verwendet werden, wie bspw. in Abbildung 5.18 zur Koordination eines nicht zur Laufzeit rekonfigurierbaren Schwarms. Andere Fähigkeiten, wie das *Messen* oder *Kalibrieren*, können ohne ROS integriert werden. Das bietet den Vorteil, auch Systeme mit unterschiedlichen Schnittstellen und Anforderungen kombinieren zu können, wie im Beispiel der synchronen Glasfasermessung.

Block Definition Language - Prozessmodellierung

Die **Block Definition Language** (BDL) ist eine Domain Specific Language (DSL) für die Anbindung an das Robot Operating System (ROS), die auf Funktionsblöcken aufbaut und Konzepte der Kontrollflüsse aus UML-Aktivitätsdiagrammen verwendet. Erweiterungen mit UML-Modellierungen für ROS finden sich auch in der Literatur [285], die auf „UML-Statemachines“ aufbauen und interne Prozesse einer Node fokussieren. BDL ist hingegen unabhängig von ROS und dient der Modellierung von Prozessen, die allerdings mit ROS verknüpfbar sind. Für die Beschreibung von BDL wird das Beispiel eines Quadrocopters mit einem Sensor verwendet, dessen Bewegungssteuerung über ROS erfolgt und die Sensoren über die OOD-API anspricht. Eine exemplarische Modellierung ist in Abbildung 5.19 illustriert. Die Semantik der BDL orientiert sich an der Kontrollflusssemantik von UML-Aktivitätsdiagrammen. So dienen Kontrollknoten wie der *Fork* aus Abbildung 5.19 der Parallelisierung des Kontrollflusses und *Joins* der Synchronisierung. Entscheidungen können über *Guards*, wie der Kontrolle des Batteriestandes, den Kontrollfluss steuern und über Merges (nicht im Beispiel) zusammenführen. Erweiterte Semantik, wie bspw. Unterbrechungsbereiche oder Objektflüsse werden hingegen nicht unterstützt. Für die Aktionen werden zwei Kategorien aufgestellt. **Funktionsblöcke** dienen der Interaktion mit ROS, wie z.B. der Initialisierung von Nodes oder der Aus-

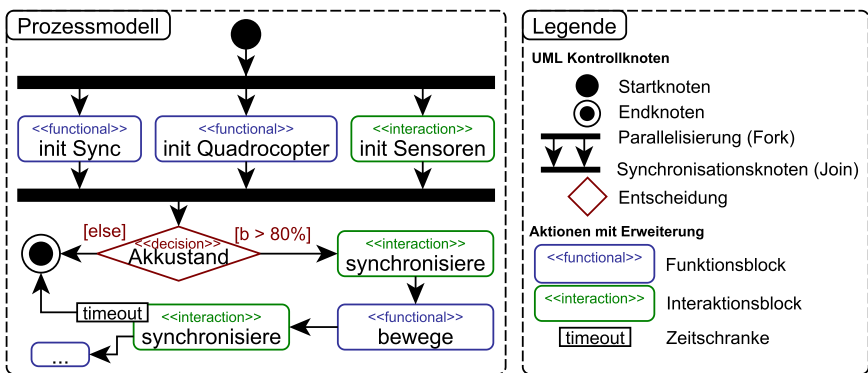

Abbildung 5.19. Prozessmodellierung eines Quadrocopters mit Sensor. Dabei wird ein UML-Aktivitätsdiagramm durch Stereotypen erweitert, um die unterschiedlichen Aktionen zu verdeutlichen. Die Zeitschranke ist für das Verständnis zusammengefasst, kann jedoch auch mit der UML-Syntax modelliert werden. (Eigenwerk)

führung von Aktionen. Im Beispiel wird die Node für die Synchronisierung und die Node des Quadrocopters gestartet und initialisiert, die *Bewege* Funktion hingegen wird auf der Quadrocopter -Node ausgeführt. **Interaktionsblöcke** dienen der Integration von Codeblöcken außerhalb der ROS-Domäne, wie im Beispiel die Initialisierung der Sensoren und die Funktion *synchronisiere*, die zwar über die ROS-Node *init Sync* ausgeführt wird, aber Schnittstellen zu externen Systeme bereitstellt, wie die *ROS 1-Bridge* in Abbildung 5.18. Eine Besonderheit bietet die Erweiterung des timeout, das sowohl als Zeitschranke als auch für Fehlerbehandlungen verwendet werden kann. Die Mächtigkeit von Aktivitätsdiagrammen würde eine explizite Modellierung erlauben, etwa über Decisions und Merges oder alternativ mit Unterbrechungsbereichen. Für ein einfacheres Verständnis wird die implizite Modellierung außerhalb der UML-Syntax verwendet. Das Resultat der Modellierung ist sowohl eine Prozessdefinition, die Befehlsstrukturen für Nodes als auch Erweiterungen, wie die OOD-API oder Agentensysteme, berücksichtigt. Das Konzept sowie die Realisierung mit zusätzlicher DSL für die Missionsplanung und Verteilung anhand der *Dynamic Mission Definition and Execution (DMDE)* werden in der Publikation [268] vorgestellt.

ROSSi - Infrastruktur

Während die Block Definition Language (BDL) Prozesse und damit dynamische Abläufe betrachtet, wird in ROS Simple (ROSSi) die statische Modellierung fokussiert. Das Projekt RoBMEX [167] unterteilt ebenfalls eine statische (ROSModL) und dynamische Modellierung (ROSMiLan) für Prozesse, Launchfiles und den Code. Das noch junge Projekt (2021) hat bereits Metamodelle für die Modellierung vorgestellt und anhand eines Eclipse-Plugins auf Klassenebene evaluiert. Eine eigene Realisierung hingegen ist noch ausstehend, aber angedeutet. ROSSi unterteilt die statische Modellierung in einen Launchfile-Editor und einen Node-Editor, wie in Abbildung 5.20 illustriert. Im Launchfile-Editor werden ROS-Nodes mittels grafischer Oberfläche miteinander assozi-

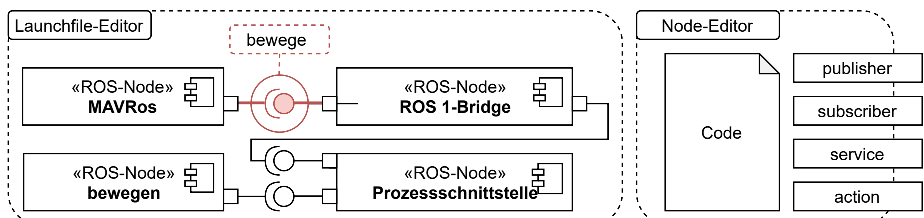

Abbildung 5.20. Der Launchfile-Editor (links) bietet neben der Infrastruktur für ROS-Nodes auch eine Anzeige aktueller Werte der Topics zur Laufzeit, das durch den *Bewege* Befehl angedeutet ist. Der Node-Editor hingegen stellt Codefragmente für die Werkzeuge von ROS bereit, wie bspw. eine Action. (Eigenwerk)

iert, um eine Launchfile zu erstellen. Als Vorlage werden wie in Abbildung 5.20 UML-Komponentendiagramme verwendet. Ähnlich zu Blueprints der Unrealengine [294] soll das statische Modell zur Laufzeit dynamische Einsichten erlauben, z.B. durch das Hervorheben von aktuell benutzten Kanten und der Anzeige des Inhalts der Nachricht. Der Node-Editor dient der konkreten Implementierung und Klassifizierung von Nodes, die wiederum in der BDL für Prozesse verwendet werden können. Da ROSSi auf der Infrastruktur von ROS aufbaut, ist eine Nutzung ohne ROS nicht vorgesehen. Vielmehr wird eine Möglichkeit geschaffen, die teils komplexe Infrastruktur übersichtlicher zu gestalten und die Fehlersuche zu vereinfachen. Das Konzept und die Evaluation von ROSSi ist publiziert [312] und der Quellcode unter der MIT Lizenz verfügbar[2]. Die Vorteile von ROSSi werden vor allem bei verteilten Systemen ersichtlich, wenn mehrere ROS-Nodes verwaltet werden müssen.

5.2.3 Verteilung

Die Verteilung additiver Informationen, wie z.B. der Selbstbeschreibung, ist bei Sensoren und Aktuatoren nur sehr selten möglich. So bietet bspw. ein einfacher Temperatursensor, wie der SHT75, keinen persistenten Speicher, oder komplexe Systeme wie die Microsoft Kinect erlaubt keinen Zugriff darauf [288]. In der Realisierung eines Adapters mit Selbstbeschreibung wird in OpenAAS [255], für die Realisierung einer Verwaltungsschale, ein Einplatinencomputer verwendet. In den Realisierungen für eine Beschreibung meteorologischer Messdaten werden übliche PCs verwendet [61]. Das Schema von TEDs ist für Mikrocontroller und FPGAs konzipiert [276]. Semantic Plug & Play hat das Ziel, unterschiedliche Plattformen zu unterstützten und die Verteilung auf den Plattformen zu vereinfachen. Dafür werden zwei Softwarebausteine vorgestellt, über die eine Selbstbeschreibung und die Verwendung von Fähigkeiten bereitgestellt werden. Der **Semantic Adapter** bildet das Instrument der Selbstbeschreibung für Hardwareelemente, die der **Semantic Controller** sammelt und über Schnittstellen dem Nutzer bereitstellt.

[2]https://github.com/isse-augsburg/ROSSi

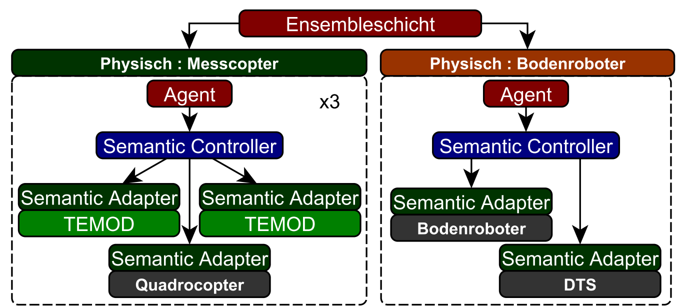

Abbildung 5.21. Verteilung der Softwarebausteine zwischen den physisch zusammenhängenden Geräten. Die Ensembleschicht aus Abbildung 3.3 steuert die Agenten, welche über den Semantic Controller die Fähigkeiten der Semantic Adapter kommandieren. (Eigenwerk)

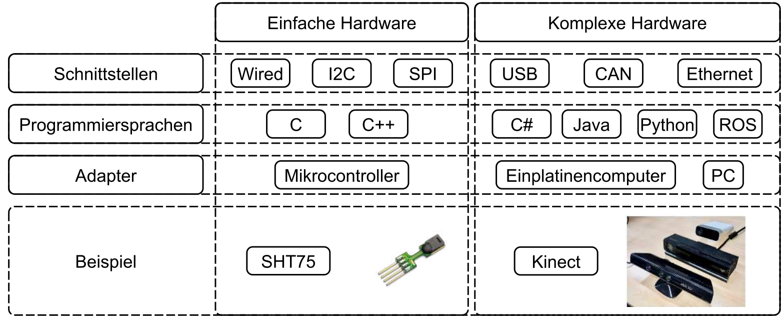

Abbildung 5.22. Unterteilung komplexer und einfacher Hardware anhand der verwendeten Adapter und exemplarische Schnittstellen und Programmiersprachen. Als Beispiel für einfache Hardware wird der SHT75 [271] und für komplexe Hardware die Microsoft Kinect [288] verwendet. (Eigenwerk)

Semantic Adapter

Der Semantic Adapter ist ein Softwareartefakt des Semantic Plug & Play und steht in zwei Varianten zur Verfügung, die in Abbildung 5.22 skizziert sind. Für die Integration einfacher Hardware, die direkt verkabelt wird, wie bspw. der SHT75 oder die im Beispiel verwendete TEMOD Auswertungseinheit, ist die Schnittstellen-API zuständig. Die Schnittstellen-API umfasst eine einfache Verwendung des internen Speichers für dynamische Eigenschaften und Methodenvorlagen, die der Commandstruktur entsprechen, sowie eine Speichermöglichkeit für semantische Netze. Der mit der Schnittstellen-API programmierte Code kann nicht automatisch verteilt werden, da Mikrocontroller in der Regel nicht über eine Webschnittstelle verfügen oder nicht immer über eine Programmierschnittstelle angeschlossen sind. Somit kann zwar in den Eigenschaften ein GIT-Repository referenziert sein, der gegen die Schnittstellen-API programmierte Code muss jedoch selbst auf den Mikrocontroller aufgespielt werden.

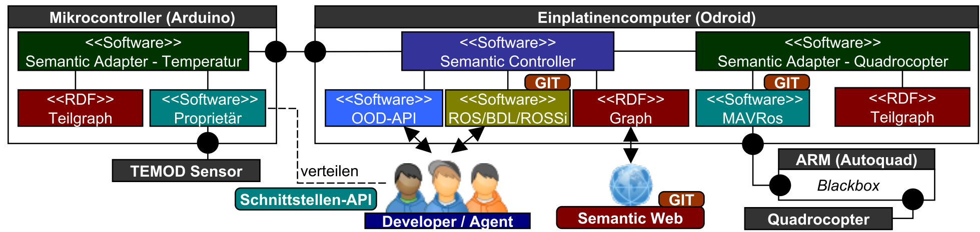

Abbildung 5.23. Aufbau des physischen Messcopters aus Abbildung 5.21 mit nur einem Temperatursensor und den Softwareartefakten *Semantic Controller* und *Semantic Adapter*. Die schwarzen Punkte symbolisieren den Übergang eine logischen in eine physische Schnittstelle und vice versa. Die mit GIT gekennzeichneten Artefakte werden in den Teilgraphen referenziert. (Eigenwerk)

Der zweite Ansatz des Semantic Adapters ist für komplexere Hardware, wie eine Microsoft Kinect, die über Treiber und ein eigenes SDK angesprochen werden muss [288]. Dafür steht eine eigene API zur Verfügung, die auch eine automatische Verteilung des Codes erlaubt. So kann der Entwickler eines Semantic Plug & Play-Systems in den Eigenschaften ein GIT-Repository referenzieren, das zur Laufzeit im *Semantic Controller* aufgelöst, dem *Semantic Adapter* zur Verfügung gestellt, heruntergeladen und ausgeführt wird. Neben den nativen Implementierungen in den Programmiersprachen Java, C# und Python stehen dabei auch ROS-Nodes zur Verfügung, die durch den Mechanismus der Selbstbeschreibung einfach verteilt werden können, wie in Abbildung 5.23 durch die Node *MAVRos* illustriert. Bedingt durch Abhängigkeiten der Leistung und der eingesetzten Sprachen steht der Semantic Adapter für komplexe Hardware nur für Einplatinencomputer und PCs zur Verfügung.

Semantic Controller

Der Semantic Controller ist die Schnittstelle für die OOD-API und externe Systeme, wie ROS. Im Sinne einer Server-Client Architektur stellt der Semantic Controller den Server für mehrere Semantic Adapter und verwaltet diese. Der Semantic Adapter und Semantic Controller muss nicht physikalisch getrennt werden, wie in Abbildung 5.23 illustriert. Wird ein physikalische getrennter Semantic Adapter an den Semantic Controller angeschlossen, so wird als erstes der Teilgraph ausgelesen, um ihn über das Semantic Web zu vervollständigen und nötige Abhängigkeiten aufzulösen. Im Falle des Mikrocontrollers aus Abbildung 5.23 ist die Software bereits aufgespielt und der Teilgraph besteht aus den Beschreibungen der implementierten Fähigkeiten sowie zusätzlichen Eigenschaften. Der Teilgraph des Quadrocopters hingegen besitzt eine Referenz auf ein GIT-Repository des MAVRos-Artefaktes, bspw. ein Docker-Container. Dieser wird anschließend auf dem *Semantic Adapter* in der aktuellen Version heruntergeladen und ausgeführt. Die daraus resultierenden Fähigkeiten stehen der OOD-API als Instanz zur Verfügung und können in der Prozessstruktur verwendet werden. Falls jedoch ein System aus mehreren Teilnehmern besteht, die lediglich ROS nutzen, so ist die automatische Verteilung vorteilhaft und die BDL unterstützt zusätzlich eine automatisierte Verteilung der Subprozesse über

GIT. Diese Funktion ist in einem Ensemble mit Agentenschicht jedoch überflüssig, da die Ensembleschicht die Dekomposition und Koordination der Agenten übernimmt.

Die Abstraktion von Prozessen findet über mehrere Schichten statt, die wie in Abbildung 5.16 unterschiedlich kombiniert werden können. Eine Verwendung der abstrakten Fähigkeiten in Ensembles wurde in den Publikationen [152–155] anhand von Simulationen und in den Publikationen [84, 309, 311] mit echter Hardware vorgestellt. Die Integration mit ROS durch die Block Definition Language [268] erfolgte nicht nur durch echte Hardware, sondern auch mit der Vorstellung smarter Sensoren, die über ein LCD-Display Statusinformationen bereitstellen. Das Modellierungstool ROSSi [312] basiert auf einem Hardwaredemonstrator mit dem Namen *Quadrocopterdirigent*, der in der vorliegenden Dissertation nicht näher vorgestellt wird. Eine weitere erwähnenswerte Publikation ist über einen Abbruch eines Prozesses und anschließender Übernahme als Sicherheitsstrategie für Ensembles [269], der in Abbildung 5.14 angedeutet ist und für die synchrone Glasfasermessung konzipiert wurde. Die automatisierte Verteilung ist konzeptuell in der Publikation [153, 311] vorgestellt. Für das Testen der Hardwareverteilung vor den Experimenten sind aus Ermangelung einer modularen Simulation im Rahmen der Dissertation mehrere Prototypen entstanden, wie etwa ein kleines Cluster mit mehreren Einplatinencomputern. Eine Simulation, die eine Verteilung durch Selbstbeschreibung unterstützt und eine dynamische Visualisierung bietet, ist Bestandteil des nächsten Kapitels.

5.3 Modulare Simulation und Visualisierung

Die Simulation in der Domäne der Robotik hilft, komplexe Prozesse vor dem Schritt in die Realität zu testen und Funktionen zu prüfen. Dabei wird eine virtuelle Repräsentation bestimmter Gegebenheiten, wie etwa die Umwelt, geschaffen, die auf domänenspezifische Anforderungen abzielen. Dabei ist eine Visualisierung der Gegebenheiten nicht zwingend erforderlich, wie etwa bei der Dynamikberechnung inverser Kinematiken [307], die numerisch dargestellt werden. Die Entwicklungen der Hardware sowie Fortschritte in der Spieleentwicklung führten zu einer engen Bindung zwischen Simulation und Visualisierung [239]. Obwohl der Anspruch einer Gameengine nicht die korrekte Simulation der realen Welt innehält, werden sie häufig als Basis für spezifische Simulationen verwendet, wie bspw. USARSim, eine Simulation für mobile Such- und Rettungsroboter mit einem Editor für das Terrain [52]. Die Hardware für die Visualisierung einer Simulation entwickelte sich ebenfalls. So werden nicht mehr Bildschirme für die Visualisierung verwendet, sondern Tablets oder spezielle Brillen, die entweder eine komplette virtuelle Umgebung darstellen oder die reale Umgebung nutzen, um virtuelle Objekte zu projizieren. Die Begriffsdefinition wird von Milgram et. al bereits 1994 unter dem Begriff **Mixed Reality** (MR) zusammengefasst und publiziert [202], wie in Abbildung 5.24 illustriert. Moderne Hardware, wie die Microsoft Hololens, ermöglichen die Realisierung der Vision durch Hologramme in der Realität [187], die auch in Simulation für die Robotik eingesetzt werden, wie bspw. für die Visualisierung von Trajektorien im Kartesischem Raum [217] aufbauend auf der inversen Kinematik.

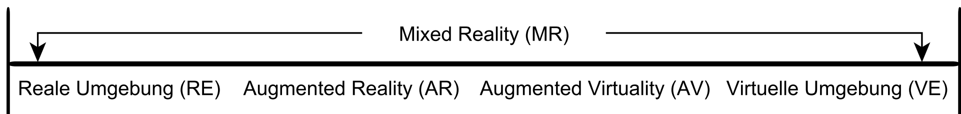

Abbildung 5.24. Definition der Mixed Reality nach Milgram et al. [202] durch das *Virtuality Continuum*. Der bekannte Begriff Augmented Reality im Zusammenhang mit 3D Brillen wird durch den Begriff Mixed Reality subsumiert. (Eigenwerk in Anlehnung an Milgram et al. [202])

Für eine Näherung zwischen der simulierten und der realen Welt werden auch *Hardware in the Loop* (HIL) Simulationen eingesetzt. Eine HIL Simulation ist die Integration eines Hardwareelements in eine Simulation, um ein korrektes Verhalten auf der Hardware sicherzustellen, wie bspw. ein Steuerelement für die Trajektorienplanung eines simulierten Knickarmroboters [191]. Das Gegenstück ist eine reine *Software in the Loop* Simulation, wie etwa in MORSE [75] mit modularen Komponenten realisiert, die auch in ROS verwendet werden können.

Eine weitere Möglichkeit für ein Bindung an die Realität sind Interaktionsmöglichkeiten innerhalb der Simulation. Neben den üblichen Interaktionen über die grafische Benutzeroberfläche und Peripheriegeräte, ist die Kombination von **Tangible User Interfaces** (TUI) in Mixed Reality Anwendungen ein interessanter Forschungsschwerpunkt. TUIs sind anfassbare Benutzer, die eine Interaktion anhand physischer Objekte erlauben und bspw. durch den Metadesk des MIT Media Lab bekannt wurden [291]. Neben der Positionsmanipulation von erkannten Objekten können bspw. auch Roboter als physische Objekte verwendet werden, indem über die Kraftmomentsensoren eine externe Manipulation der Achsstellung ermöglicht wird [298].

Das in diesem Kapitel vorgestellte Konzept der **modularen Simulation** für modulare Hardware umfasst eine zur Laufzeit erweiterbare Simulation, die eine Mixed Reality Visualisierung unterstützt, echte Hardware integriert und durch physische Objekte manipulierbar ist. Dadurch sollen Multipotente Systeme mit vielen Schwarmteilnehmern aus der simulativen Entwicklung schrittweise in die Realität überführt werden können.

5.3.1 MR-Bausteine

Ein Semantic Adapter dient der Kapselung einer Hardware sowie der Selbstbeschreibung, die in einem Semantic Controller gesammelt und bereitgestellt wird. Der Semantic Controller wird auf einem Einplatinencomputer installiert, auf dem sich auch, wie in Abbildung 5.23 illustriert, Semantic Adapter befinden können. Er ist das zentrale Element eines physikalisch zusammenhängenden Systems. Das Konzept des **MR-Bausteines** ist es, diesen Umstand als Basis für eine modulare Simulation zu verwenden. Der Einplatinencomputer gilt dabei als Repräsentant und physisches Interaktionsmittel für einen Digitalen Zwilling. Der **Digitale Zwilling** in der Domäne von Semantic Plug & Play umfasst eine Erweiterung der Selbstbeschreibung um 3D Modelle sowie simulierte Fähigkeiten, wie in Abbildung 5.25 illustriert. Der Einplatinencomputer mit digitalem Zwilling wird als MR-Baustein tituliert. Neben den zu realisierenden Objekten kann

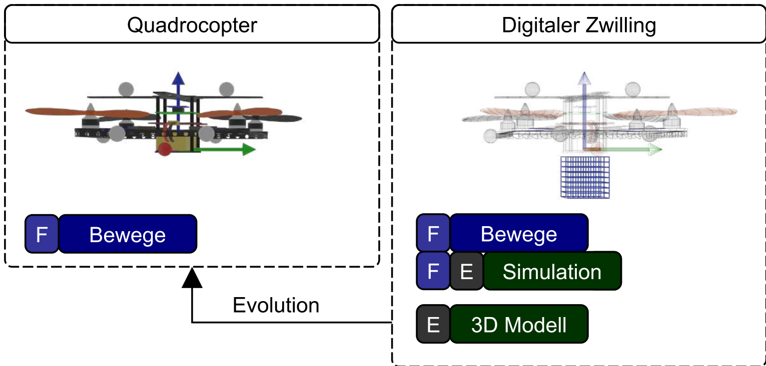

Abbildung 5.25. Digitaler Zwilling eines Quadrocopters mit einem 3D Modell das als Eigenschaft bereitgestellt wird, sowie der Fähigkeit *Bewege* für die Simulation. (Eigenwerk)

ein MR-Baustein auch Objekte der Simulationsumgebung repräsentieren, wie bspw. das Infrastrukturelement *Haus* in Abbildung 5.26. Diese können frei im Raum positioniert werden und stellen damit eine haptische Interaktionsmöglichkeit (TUI) für den Nutzer dar. Die Prozessalgorithmik kann sowohl digitale Zwillinge über simulierte Fähigkeiten als auch echte Hardware steuern, indem ein gemeinsames Bezugskoordinatensystem verwendet wird. Der Nutzer verwendet eine 3D-Brille, um die Visualisierung als Überlagerung der Realität in Mixed Reality wahrzunehmen.

Dieses Konzept bietet folgende Vorteile:

Schrittweise Überführung in die Realität
Durch das Zusammenspiel der virtuellen und realen Elemente ist eine schrittweise Überführung der virtuellen in die reale Welt möglich.

Echte Verteilung in der Simulation
Die auf GIT aufbauende Verteilungsstrategie kann auf den Einplatinencomputern der MR-Bausteine getestet werden, um bereits in einem frühen Stadium Fehlverhalten diagnostizieren zu können.

Rekonfigurationen zur Laufzeit
Durch die Struktur des Semantic Controllers und der Adapter können hybride Varianten zur Laufzeit realisiert werden, wie bspw. ein echter Quadrocopter mit einem digitalen Zwilling eines Sensors.

Integration von Hardware
Die Verwendung echter Hardware innerhalb der Simulation kann sowohl über simulierte Fähigkeiten auf einem Gerät (HIL) als auch außerhalb durch das Zusammenspiel mit der (virtuellen) Umgebung realisiert werden.

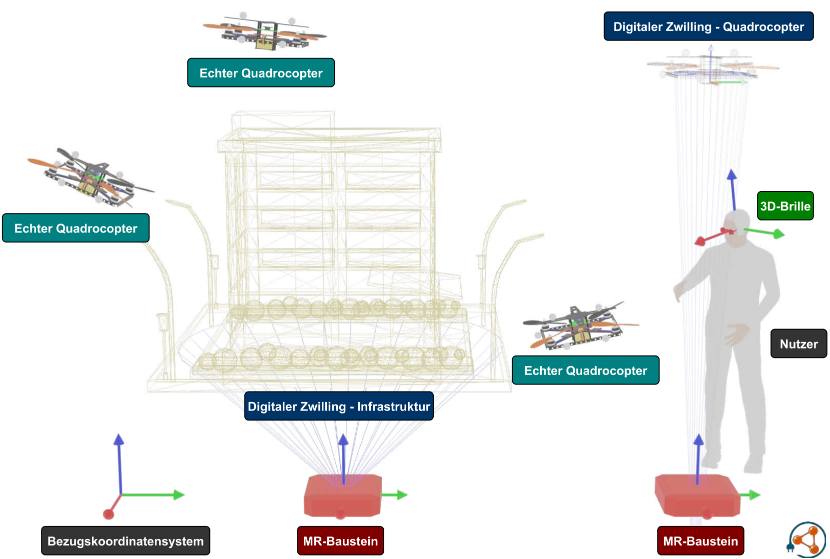

Abbildung 5.26. Konzept der MR-Bausteine in Kombination mit echten Quadrocoptern und dem Nutzer. Der Nutzer trägt eine 3D-Brille, um die Hologramme der digitalen Zwillinge zu sehen. Das Bezugskoordinatensystem gilt für alle physischen und visualisierten Objekte. (Eigenwerk)

Interaktion durch die Umwelt
Neben der Erfassung realer Komponenten ist der MR-Baustein selbst ein Interaktionsmittel, womit bspw. eine Planung in der Simulationsumgebung ermöglicht wird. So können verschiedene Infrastrukturen auf einzelne MR-Bausteine verteilt werden, um die virtuelle Umgebung zu gestalten.

Schwärme plausibel evaluieren
Gerade im Bereich der Roboterschwärme sind Hardwarerealisierungen aufgrund der Anzahl der Schwarmteilnehmer sehr zeit- und arbeitsintensiv. Die Simulationen sind hingegen meist realitätsfern. Durch das Zusammenspiel realer und simulierter Hardware können Aspekte des Verhaltens mit echter Hardware dargestellt und die Reaktion des Roboterschwarms visualisiert werden.

Aus diesem Konzept ergeben sich allerdings einige Herausforderungen an die unterliegende Simulation, die mit Laufzeitänderungen und Abhängigkeiten zwischen der Realität und den digitalen Zwillingen konfrontiert wird.

5.3.2 Simulation

Die Simulation muss Kenntnis über alle verfügbaren Semantic Controller haben und reiht sich damit auf die Ebene der Ensembleschicht ein, wie in Abbildung 5.27 illustriert. Eine Assoziation zur Ensembleschicht hingegen ist nicht vorgesehen, so dass die Agenten den Unterschied zwischen Simulation und realer Hardware nicht nachvollziehen können. Die Simulation hat die Aufgabe, entsprechende Fähigkeiten virtuell oder real

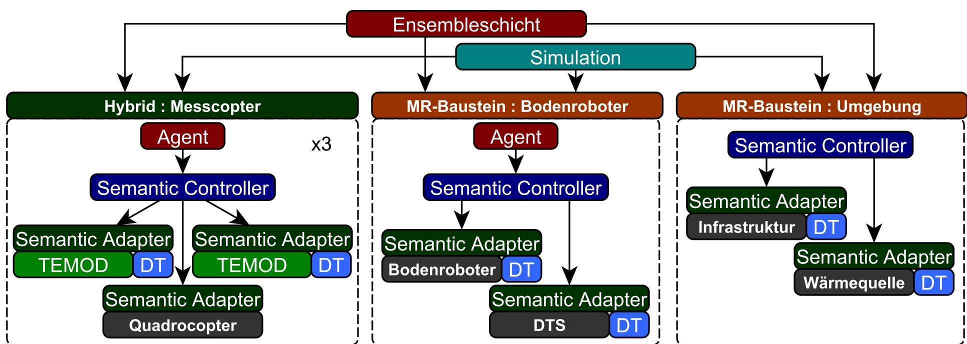

Abbildung 5.27. Exemplarischer Aufbau einer gemischten Simulation, bestehend aus einem Messcopter mit realem Quadrocopter und zwei simulierten TEMOD Sensoren. Das Postfix DT steht für Digitaler Zwilling. (Eigenwerk)

auszuführen. Diese Orchestrierung ist allerdings nur eingeschränkt möglich. So kann der *Messcopter* aus Abbildung 5.27 zwar einen virtuellen *TEMOD* Sensor tragen, aber nicht vice versa. Ebenso wenig kann ein echter Sensor eine virtuelle Wärmequelle messen. Diese Abhängigkeiten werden in der Simulation modelliert, um die Zwischenschritte für die schrittweise Überführung zu definieren. Zwar können über den Semantic Controller die Eigenschaften und Fähigkeiten zur Laufzeit bereitgestellt werden, die Abhängigkeiten hingegen müssen zur Designzeit definiert sein. Um dennoch Rekonfigurationen zur Laufzeit in der Simulation zu ermöglichen, ist eine Sammlung der potentiell genutzten Hardwareklassen erforderlich. Doch auch wenn die Klassen zur Designzeit bekannt sind, so bieten übliche Simulationen, wie MORSE [75] oder Gazebo [146], als auch Visualisierungen wie Unity [114] oder Unreal [294], keine Möglichkeit, zur Laufzeit 3D-Modelle hinzuzufügen. Dieser Tatsache schuldend wird im Rahmen der vorliegenden Dissertation die Selbstbeschreibung verwendet, jedoch wird in der Simulation der Laufzeitaspekt gemindert durch eine Kenntnis der konkreten Hardwareinstanzen zur Designzeit.

5.4 Methodologie von Semantic Plug & Play

Zusammengefasst ergibt sich aus dem Kapitel der Techniken für die Selbstbeschreibung die Methodologie des Semantic Plug & Play in der Reihenfolge aus Abbildung 5.28. In der **Annotation** werden Eigenschaften und Fähigkeiten in einem semantischen Netz unter Einhaltung der Prinzipien von Linked Data und unter Verwendung des Semantic Hardware Webs beschrieben. Innerhalb des semantischen Netzes können GIT-Repositories referenziert werden, die Artefakte bereitstellen. Die **Integration** dieser Artefakte findet entweder in der Objektorientierten API (OP), in der Schnittstellen-API oder im Robot Operating System (ROS) statt. Dabei können die semantischen Netze im Code mit den Fähigkeiten des semantischen Netzes verknüpft werden. Die Modellierung umfasst die Prozesslogik in Fähigkeiten und Prozesse mit unterschiedlichen Varianten. In der OOD-API können im Code Anforderungen ausgedrückt werden, die zur Laufzeit instanziiert

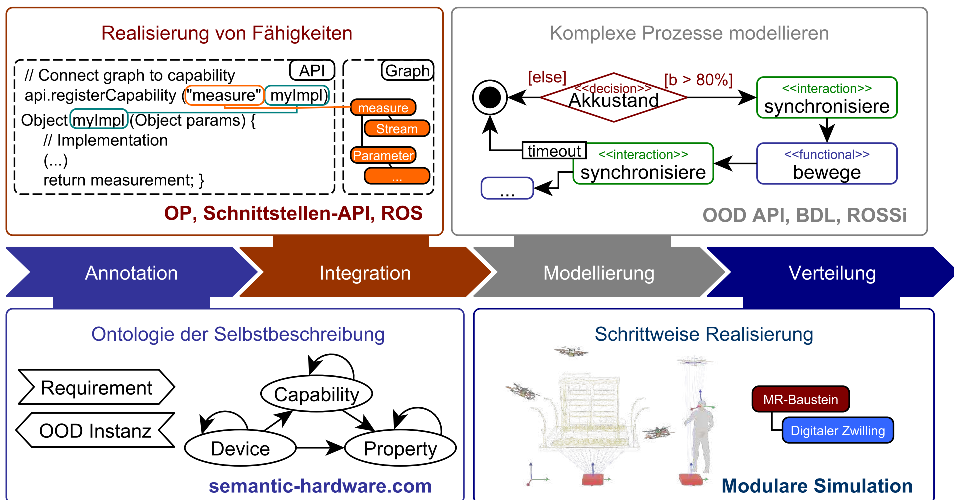

Abbildung 5.28. Methodologie für die Bereitstellung selbstbeschreibender Hardware innerhalb des Semantic Plug & Play beginnend bei der Annotation von Eigenschaften mittels des Semantic Hardware Web über die Integration der Fähigkeiten bis hin zu dedizierten APIs, der Modellierung von Prozessen im Code oder anhand abstrakter Modelle, sowie die schrittweise Realisierung in einer modularen Mixed Reality Simulation. (Eigenwerk)

werden, sofern die aktuelle Hardwarekonfiguration dies zulässt. Speziell für die Integration von ROS bietet ROSSi eine grafische Modellierung der statischen Komponenten mit Laufzeitüberwachungen an und in der Block Definition Language (BDL) können gezielt Prozesse auf dieser Infrastruktur definiert werden. Für eine schrittweise Realisierung dient die modulare Simulation, in der MR-Bausteine mit digitalen Zwillingen eingesetzt werden, um die Lücke zwischen Simulation und Realität zu verkleinern.

Die Methodologie stützt sich auf mehrere Blackbox-APIs, die symbiotisch zwischen der Objektorientierung und semantischen Netzen vermitteln, ausführbaren Code herunterladen und zur Laufzeit in die Prozesse einpflegen. Das nächste Kapitel öffnet diese Blackbox und offenbart die zugrundeliegende Architektur und Mechanik von Semantic Plug & Play.

Zusammenfassung. Die Methodologie von Semantic Plug & Play baut auf unterschiedlichen Werkzeugen und APIs auf, die in diesem Kapitel anhand einer Gesamtarchitektur verortet und beschrieben werden. Beginnend bei den objektorientierten Anforderungen über eine Verknüpfung mit semantischen Netzen bis zur realen Ausführung auf der Hardware spielen die Softwarekomponenten, Austauschprotokolle und Formate der Architektur eine Rolle.

6 Architektur und Realisierung

6.1 Experiment: Pick & Place **98**
6.2 Die Model-Domain-Domainmodel-Architektur **100**
6.3 Domain - Objektorientierte Schnittstelle **101**
6.4 Domainmodel - Mapping der Domäne **105**
6.4.1 Graphensuche 106
6.4.2 Virtuelle Fähigkeiten 107
6.5 Model - Integration der Hardware **111**
6.5.1 Das Self Descriptive Protocol 112
6.5.2 Die Schnittstellen-API 114
6.6 Zusammenfassung und Einordnung **115**

Das Konzept der Selbstbeschreibung sowie die daraus folgende Methodologie erfordert Softwarekomponenten für objektorientierte Programmiersprachen, sowie Bibliotheken prozeduraler Programmiersprachen, die in einer Middleware des Semantic Controllers miteinander in Beziehung gebracht werden, wie in Abbildung 6.1 illustriert. Die Anforderung an die Middleware für die Unterstützung von Erweiterungen wie dem Robot Operating System (ROS), unterschiedlichen Austauschformaten und logischen Schnittstellen, sowie die Anbindung an das Semantic Web erfordern einen erheblichen Programmieraufwand, den es zu strukturieren gilt. Vergleichbare Projekte aus der Meteorologie von Dibley et al. [61] unterteilen die Softwarebestandteile anhand einer sehr groben Schichtenarchitektur, während in den Implementierungen der Verwaltungsschale häufig eine serviceorientierte Architektur aufbauend auf OPC-UA vorgeschlagen wird [262]. In konkreten Projekten werden hingegen Mischformen verwendet [227]. Mischformen sind nicht negativ zu bewerten, sondern spiegeln die unterschiedlichen Facetten einer größeren Codebasis wieder und können auch symbiotisch, wie in Abschnitt 4.2.3 dargestellt, verwendet werden. So können mehrere Architekturen je nach Anforderungen einzelner Softwarekomponenten in einem einzigen Projekt Verwendung finden.

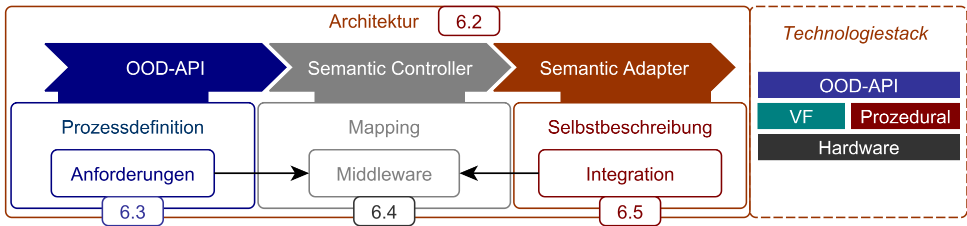

Abbildung 6.1. Kapitelübersicht der Architektur, die sich über den Semantic Adapter und Controller bis zur OOD-API erstreckt. Für eine Einordnung ist im rechten Teil der Abbildung der Stack aus Abbildung 5.16 skizziert. (Eigenwerk)

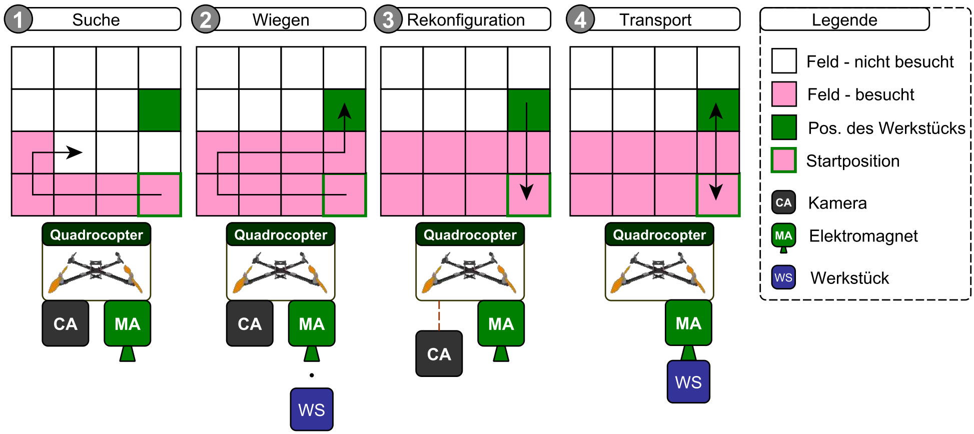

Abbildung 6.2. Das Experiment Pick & Place wird in vier Abschnitte unterteilt, die von links nach rechts aufgeführt sind. Das Areal ist für die Suche (1) in einzelne Felder unterteilt, die mit der Kamera abgesucht werden. Nach dem Wiegen (2) des Werkstückes wird eine Überschreitung der zulässigen Gesamtlast festgestellt und die Kamera muss demontiert werden (3), damit der Transport (4) gelingt. (Eigenwerk)

In diesem Kapitel wird eine symbiotische Gesamtarchitektur vorgestellt, die sich über den Semantic Adapter, Controller und die OOD-API zieht, wie in Abbildung 6.1 skizziert. Aus dem Technologiestack wird lediglich der linke Teil betrachtet, während Erweiterungen im nächsten Kapitel vorgestellt werden. Die Komponenten der Architektur umfassen die objektorientierte Schnittstelle, die Integration der Hardware sowie des Mapping zwischen den beiden. Als leitendes Beispiel wird das Pick & Place Experiment verwendet.

6.1 Experiment: Pick & Place

Der Ablauf des Experiments wird in vier Phasen untergliedert, wie in Abbildung 6.2 illustriert. In der ersten Phase wird anhand einer Rasterkarte ein vordefiniertes Feld mit einer Kamera abgesucht, die Positionen von Farbmarkern erkennt. Die Rasterkarte ist

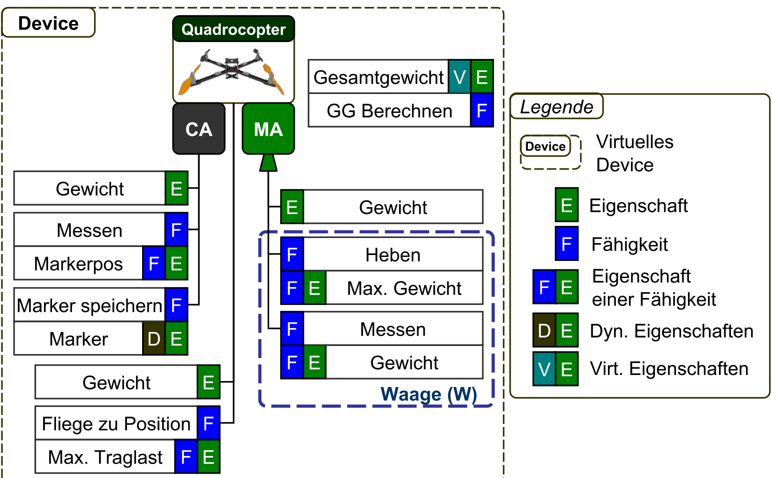

Abbildung 6.3. Das im Experiment verwendete Device besteht aus einem Quadrocopter mit einer Kamera und einem Elektromagneten. Die virtuelle Fähigkeit des Devices besteht aus der Berechnung des Gesamtgewichtes aus der Summe aller Einzelgewichte physischer Elemente sowie dem gemessenem Gewicht der Waage (W), einer kombinierte Fähigkeit. Die Kamera hat eine integrierte Markererkennung, wobei Marker für die Identifikation von Objekten in einer dynamischen Eigenschaft gespeichert sind. (Eigenwerk)

frei von Hindernissen, weshalb der Einsatz eines trivialen Suchalgorithmus ermöglicht wird. Wenn das Werkstück anhand eines Farbmarkers gefunden ist, startet die zweite Phase, das Wiegen, in welcher der Quadrocopter das Werkstück über den Elektromagneten aufnimmt. Durch die Kombination des Anhebens in Verbindung mit dem benötigten Schub der Rotoren kann ein Rückschluss auf das Gewicht des Werkstücks geschlossen werden. Übersteigt das Gesamtgewicht die Traglast der Fähigkeit *Fliege zu Position*, die in Abbildung 6.3 skizziert ist, kann die Fähigkeit nicht ausgeführt werden. Der Fall der Überschreitung wird im Experiment bewusst provoziert, um die dritte Phase, die Rekonfiguration, zu betreten. In dieser wird die Position des Werkstücks in der dynamischen Eigenschaft *Position* gespeichert und der Quadrocopter fliegt zur Startposition zurück. Dort wird die Kamera demontiert, damit eine Gewichtsersparnis erzielt wird, um anschließend das Werkstück in der letzten Phase transportiert werden kann.

Dieses Experiment erfordert ein breites Spektrum an Fähigkeiten und Eigenschaften, die in Abbildung 6.3 aufgeführt sind, sowie eine Rekonfiguration zur Laufzeit durch die Demontage der Kamera. Als Vorbereitung für das Experiment muss das Werkstück farblich markiert und der Farbmarker auf der dynamischen Eigenschaft gespeichert werden. Der Suchalgorithmus verwendet diesen Farbmarker um die Position des Werkstücks auszumachen, indem er die Positionen des Rasters sequentiell mit der Fähigkeit *Fliege zu Position* anfliegt. Die Fähigkeit des *Gewichtsmessens* basiert auf der Fähigkeit *Heben* und internen Sensordaten des Quadrocopters, die aus Platzgründen nicht in Abbildung 6.3 modelliert sind. Da die Kamera zur Laufzeit entfernt wird, stehen dem System auch nicht mehr deren Selbstbeschreibungen zur Verfügung. So wird die domänenspezifische

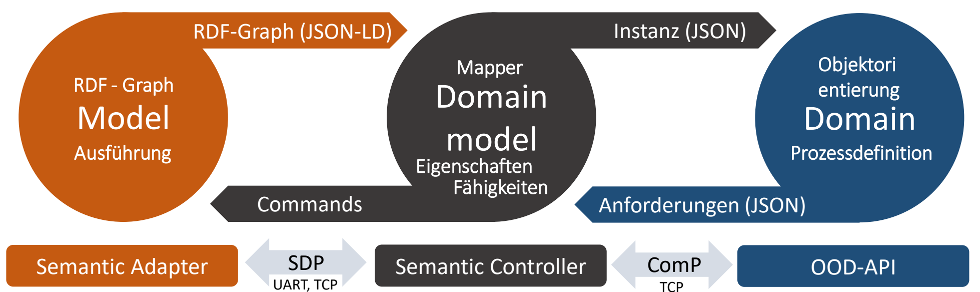

Abbildung 6.4. Die Model-Domain-Domainmodel (MDDM)-Architektur orientiert sich an der MVVM Architektur aus Abschnitt 4.2.3. In der Domain werden die Requirements definiert, die im Domainmodel mit Commands gebunden werden. Als Austauschformat für die RDF-Graphen im Model ist JSON-LD vorgesehen und der Austausch findet über ein eigenes Protokoll statt: das Self Descriptive Protocol (SDP). Die Instanz mit den gebundenen Commands wird in einer JSON Serialisierung über das Commands Protocol (ComP) der Domain zur Verfügung gestellt. (Eigenwerk)

Position in der Prozessdefinition auf dem Semantic Controller gespeichert und für den Transport verwendet.

Dieses technische Experiment dient der Illustration aller Komponenten der Selbstbeschreibung, sowie einer Rekonfiguration als Grundlage für die Vorstellung der Gesamtarchitektur.

6.2 Die Model-Domain-Domainmodel-Architektur

Die **Model-Domain-Domainmodel** (MDDM)-Architektur ist angelehnt an die in Abschnitt 4.2.3 vorgestellte Model-View-Viewmodel-Architektur (MVVM). Das Grundprinzip der MVVM besteht in der Zwischenschicht, das Viewmodel, zwischen dem Model und den einzelnen Views, um einer Aufblähung des Models entgegenzuwirken. So benötigt eine View nur ein Subset des Models sowie individuelle Anpassungen für die Darstellung. Für die dependency inversion zwischen View und Vievmodel werden data bindings statt publish subscribe Mechanismen verwendet [277].

Das Prinzip der Separierung von Anforderungen einer View wird in der MDDM-Architektur adaptiert und auf die Domäne angewendet, wie in Abbildung 6.4 skizziert. So besitzt jede Domänenapplikation eigene Anforderungen bezüglich des Models, die in einem Domainmodel zusammengefasst werden. Diese Anforderungen werden allerdings nicht im Domainmodel ausgedrückt, sondern kommen direkt aus der Domäne anhand einer serialisierten JSON-Datei mit einer Blaupause der benötigten Fähigkeiten und Eigenschaften. Die *Blaupause* besteht aus noch leeren Instanzen für die Struktur und einer textuellen Beschreibung der Anforderungen einzelner Strukturelemente. Das Domainmodel überprüft anhand des verteilten Models einzelner Hardwareelemente, ob die Anforderungen der Blaupause erfüllbar sind und assoziiert die Ausführungslogik geforderter Fähigkeiten in Kombination mit den dazugehörigen Eigenschaften. Als Austauschformat zwischen dem Model und dem Domainmodel wird JSON-LD [142]

eingesetzt, eine schlanke Speichermöglichkeit für semantische Netze im RDF-Format auf JSON-Basis. Das Resultat des Mappings im Domainmodel ist eine Instanz, die auf der Struktur der Blaupause aufbaut, Eigenschaften der Hardware ergänzt und Fähigkeiten mit der Commandstruktur assoziiert. Diese JSON-Instanz überschreibt automatisch zur Laufzeit die Blaupause in der Domain ohne die vorgegebene Struktur zu verletzen, womit ein Prozess definiert werden kann, ohne die konkrete Realisierung der Hardware vorauszusetzen. Dieses Vorgehen ist mit dem Softwarepattern *Template Method* verwandt, in dem abstrakte Methoden innerhalb einer abstrakten Klasse ohne konkrete Implementierung verwendet werden können. Durch die Commandstruktur wird zudem die dependency inversion sichergestellt, so dass ein Zugriff des Domainmodels auf die Domain ausgeschlossen wird.

Im Gegensatz zur MVVM-Architektur ist die MDDM-Architektur strikt durch definierte Austauschformate getrennt. Dies bietet nicht nur den Vorteil einer verteilten Ausführung, sondern auch die Einhaltung der Trennung zwischen den Schichten. Der in vielen WPF-Anwendungen oder in ASP.NET übliche Beipass des Viewmodels durch die Missachtung der Struktur ist damit praktisch ausgeschlossen bzw. erfordert Anpassungen sowohl in den Austauschformaten als auch in der Schnittstellenimplementierung. In Semantic Plug & Play werden für den Austausch zwischen den einzelnen Komponenten spezielle Protokolle verwendet, die für unterschiedliche physikalische Schnittstellen konzipiert sind. Das **Self Descriptive Protocol** (SDP) ist primär für die Verwendung zwischen Einplatinencomputern und Mikrocontrollern entwickelt und eignet sich für den Versand über eine serielle Schnittstelle (UART). Die Commands hingegen werden über das **Commands Protocol** (ComP) ausgetauscht, das auf dem TCP Stack basiert.

Die einzelnen Bestandteile der Architektur und deren Implementierung werden in den nachfolgenden Kapiteln anhand des Pick & Place-Experiments vorgestellt.

6.3 Domain - Objektorientierte Schnittstelle

Die objektorientierte Schnittstelle verwendet JSON als Austauschformat, um die Portierung in beliebige objektorientierte Programmiersprachen zu vereinfachen. Da das in der Fallstudie eingesetzte Multiagentenframework Jadex aus [152–154] auf Java aufbaut, ist die erste Portierung der OOD-API nativ in Java geschrieben und mittels IKVM auch in C# nutzbar. Die Klassenstruktur in Abbildung 6.5 übernimmt weitestgehend das Konzeptmodell aus Abbildung 5.4 mit Erweiterungen für die Definition von Anforderungen und der Klasse *ThinClient* als Schnittstellenimplementierung für das *Commands Protocol*. Die Selbstassoziation der Klasse Device dient als Strukturelement, so kann ein Messcopter aus den Subdevices Quadrocopter, Kamera und Elektromagnet zusammengestellt sein. Für die korrekte Instanziierung ist dies allerdings irrelevant, da auch dann ein Device mit allen Fähigkeiten erstellt werden kann, wenn keine Rekonfigurationen zur Laufzeit stattfinden. Für die Definition der Anforderungen werden Devices mit *DeviceRequirements* und *CapabilityRequirements* erstellt. Das String-Array wird über den Konstruktor befüllt und bietet die Möglichkeit, anhand von Freitexten die Anforderungen an die Eigenschaften zu definieren. Das Domainmodel entscheidet anhand der Freitexte, welche

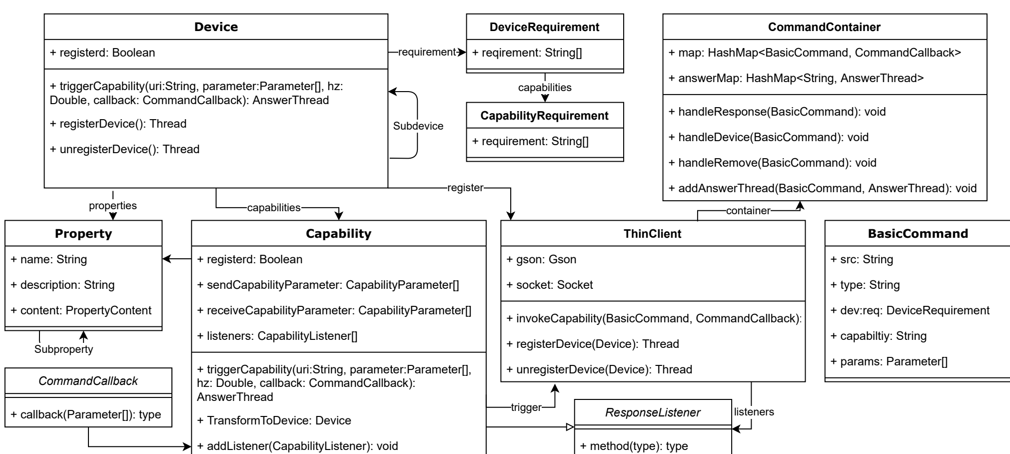

Abbildung 6.5. Auszug aus dem Klassendiagramm der OOD-API mit dem Model, bestehend aus den Strukturelementen Device, Capability und Property sowie der Schnittstellenimplementierung für das *Commands Protocol* in Form des ThinClients. (Eigenwerk)

Eigenschaften der Devices oder welche Fähigkeiten adressiert sind. Als Beispiel werden zwei Varianten für die Instanziierung vorgestellt.

```
1  // Blaupause für ein Device mit der Eigenschaft pixy
2  Device camera = new Device(new DeviceRequirement("pixy")));
3
4  // Blaupause anhand der Fähigkeiten mit den Eigenschaften FlyToPosition,
       GetPosition
5  Device copter = new Device(new DeviceRequirement(
6       new CapabilityRequirement("FlyTo"),
7       new CapabilityRequirement("GetPosition"),
8       new CapabilityRequirement("LiftWeight")));
```

Für jede Fähigkeit und jedes Device existiert eine Namenseigenschaft, die über *Device-Requirements* adressiert werden können, wie bspw. das Device mit dem Namen *camera* durch das Devicerequirement *pixy* in Zeile 2. Mit diesem Aufruf wird das physikalische Device mit der Eigenschaft *pixy* angefordert. Eine Alternative ist die Erstellung eines Devices anhand der Eigenschaften seiner Fähigkeiten, wie im Device *copter* exemplarisch dargestellt. Anders als beim Duck-Typing wird nicht etwa ein passendes Device gesucht, sondern ein virtuelles Device mit den geforderten Fähigkeiten zweier physikalischer Devices, des Quadrocopters und des Elektromagneten, erstellt.

Die Blaupausen der Devices können über den Wrapper *registerDevice()* an das Domainmodel gesendet werden. Wenn davon ausgegangen wird, dass die Hardware verfügbar ist, kann die Methode *.join()* für einen synchronen Aufruf verwendet werden.

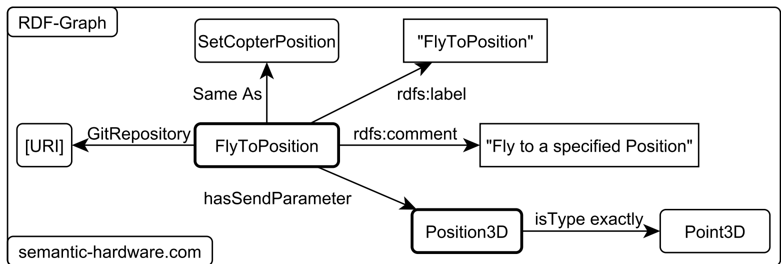

Abbildung 6.6. In der Semantic Hardware Web Ontologie finden sich exemplarische Individuals, die konkrete Hardware beschreiben. Ein Auszug ist in dem Graphen zu sehen, wobei nur die Knoten mit dicker Umrandung näher beschrieben sind. Eckige Boxen stellen Literale dar. (Eigenwerk)

```
1  // Registrierung der Device im Domain-Model und Synchronisation durch .join()
2  copter.registerDevice().join();
3  // Aktuelle Position des Copters, die von der konkreten Instanz des Models
       zurückgegeben wird
4  Object[] pos = copter.triggerCapability("GetPosition").waitForAnswer();
5  // Verwendung der FlyToPosition Fähigkeit
6  copter.triggerCapability("FlyToPosition", new Parameter(pos)).join();
```

Das Messen der Position wird über die Nomenklatur der Instanziierung, also *GetPosition*, ausgeführt und durch die Methode *waitForAnswer* synchronisiert. Wenn jedoch ein Device ohne Fähigkeiten erstellt wird, wie in Zeile 2 das Device *camera*, werden alle Fähigkeiten des physischen Devices assoziiert. Der Aufruf einzelner Fähigkeiten erfolgt in diesem Fall über den konkreten Namen der Fähigkeit, wie er auch im RDF-Graph steht.

Die für die Programmierung benötigte Dokumentation der Fähigkeiten befindet sich vollständig im RDF-Graph und kann auch zur Laufzeit über die Konsole abgefragt werden, um eine Übersicht der aktuellen Hardwarekonfiguration anzuzeigen.

```
1  capability properties:
2  name: FlyToPosition (STRING),
3  description: Fly to a specified Position (STRING),
4  uri: http://semantic-hardware/ontology/individuals/FlyToPosition (STRING),
5
6  Additional Properties for triggering this capability:
7  name: Position3D (STRING),
8  description: unkown (STRING),
9  uri: http://semantic-hardware/ontology/individuals/Position3D (STRING),
10 (...)
```

Der Konsolenoutput zeigt nur ein Subset des RDF-Graphen in Abbildung 6.6, der für die Programmierung benötigt wird. Durch die Referenz der URIs können weitere In-

formationen direkt abgerufen werden. Die Verwendung der Linked-Data-Prinzipien erfordert eine vollständige Onlinebeschreibung, so besitzt jeder Knoten der Individuals (Instanz der Ontologie) eine eindeutige URI, über die auch ein Zugriff ohne vorhandene Hardware erfolgen kann. So können auch Dokumententationen aus den RDF-Graphen generiert werden, wie etwa die Beispielindividuals der Semantic Web Ontology[1], die mit Protégé [208] der Stanford University in das HTML Format konvertiert wurden.

Die Fähigkeiten (Capability) in dem Designklassendiagramm aus Abbildung 6.5 der OOD-API haben im Gegensatz zu dem Konzeptmodell aus Abbildung 5.4 keine Selbstassoziation. Das Konzept aufeinander aufbauender Fähigkeiten ist damit nicht innerhalb der OOD-API realisiert, sondern bewusst durch das Viewmodel obfuskiert. Die Selbstassoziation von Eigenschaften (Property) ist hingegen vorhanden, damit auch präzisere Angaben wie die Position3D aus Abbildung 6.6 gemacht werden können. Der Point3D besteht aus einer x,y,z-Koordinate im Datentyp *double*, womit eine Rotation in der Fähigkeit *FlyToPosition* nicht berücksichtigt wird, da nur Geräte mit omnidirektionaler Bewegung verwendet werden.

Eine geplante Rekonfiguration kann durch ein *unregister* des konkreten Devices erfolgen, um die Provokation von Fehlermeldungen und deren Behandlung zu vermeiden. Wird ein Device zur Laufzeit getrennt, so erzeugen alle Fähigkeiten, die im Code angestoßen werden, einen *DeviceNotAvailable* Fehler. Besonders Devices, die sich der Fähigkeiten mehrerer physikalischer Devices bedienen, wie der *copter* aus dem Beispiel, stellen potentielle Fehlerquellen für Rekonfigurationen dar. So sollten rekonfigurierbare Elemente stets als physische Devices instanziiert werden.

```
1 // Helferklasse für alle Devices, auch die selbst instanziierten
2 devices = DeviceContainer.getInstance().getPhysicalDevices();
3 // Berechne Gesamtgewicht
4 sum = 0;
5 for (Device d : devices)
6   sum += (Double)d.getPropertyByName("Weight").getValue();
7 ...
8 // Entkoppeln
9 camera.unregister().join();
```

Für eine Übersicht aller Eigenschaften existiert eine Helferklasse mit dem Namen *DeviceContainer*, die alle physischen Devices instanziieren kann, aber auch bereits instanziierte Devices registriert. Dadurch kann bspw. das Gesamtgewicht aller physischen Geräte ermittelt werden, das als Auslöser für die Rekonfiguration im Pick & Place-Beispiel verwendet wird.

Die Vorteile der schlanken, wenn auch sehr generischen OOD-API sind die Entkopplung der Hardware von der Prozessdefinition durch Fähigkeiten und deren erweiterbare Beschreibung. Die Maschinenlesbarkeit des RDF-Graphen erlaubt es, Selbstbeschreibungen durch Eigenschaften im Code zu verwenden, wie bspw. bei der Berechnung des Gesamtgewichts. Auch Parameter müssen definiert und mit primitiven Datentypen assoziiert werden. Die sonst übliche Semantik in der Codedokumentation, wie etwa die

[1] www.semantic-hardware.com

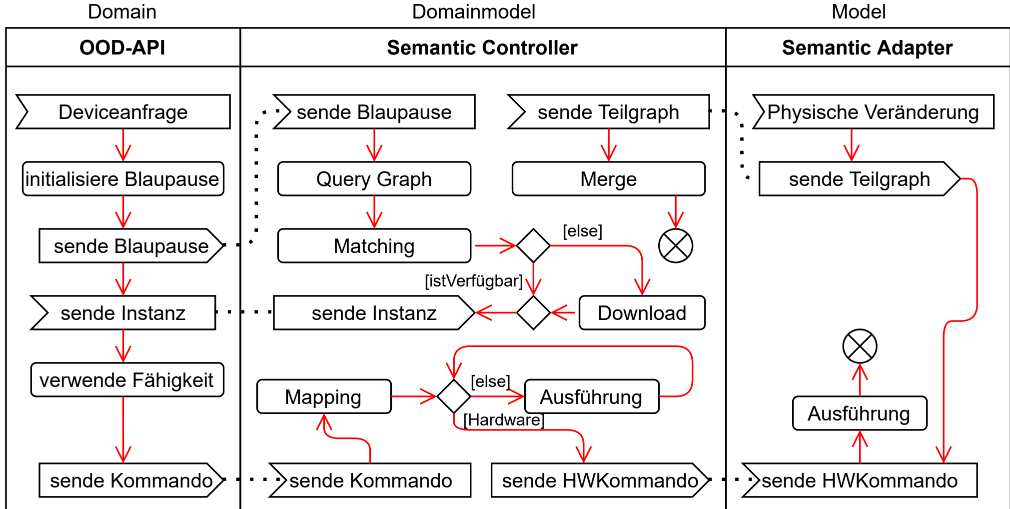

Abbildung 6.7. UML-Aktivitätsdiagramm für die Visualisierung des Kontrollflusses zwischen OOD-API, Semantic Controller und Adapter. Protokollspezifische Nachrichten werden als Signale mit einer gestrichelten Linie zwischen Sende und Empfangssignalen modelliert. Es wird nur ein exemplarischer Ablauf beschrieben, der beim Senden des ersten Kommandos endet. (Eigenwerk)

verwendeten SI-Einheiten, können nachträglich im Graphen ergänzt werden, ohne den Quellcode der Hardware zu ändern. Durch die Bereitstellung der Selbstbeschreibung mittels HTTP-URIs wird der Gedanke des Semantic Webs auf Hardware übertragen, und durch die öffentlich zugängliche Verteilung kann der RDF-Graph mit anderen RDF-Graphen assoziiert und erweitert werden. Die Anforderungen der Domäne können hingegen eine eigene Granularität aufweisen. So ist die alleinige Anforderung der Temperaturmessung mit praktisch jedem Temperatursensor vereinbar und ein Wechsel des Sensors führt nicht zu einer Anpassung im Code der OOD-API. Die Freitexteingabe bei der Deviceinstanziierung, die Kombination von Fähigkeiten und das Protokoll zwischen der Domain und dem Domainmodel wird im folgenden Kapitel näher betrachtet.

6.4 Domainmodel - Mapping der Domäne

Die Aufgaben des Domainmodels als zentrales Organ bestehen aus der Sammlung und dem Merge aller Teilgraphen angeschlossener Hardware, der Auflösung von Blaupausen in Instanzen und der Koordination der Kommandos, wie in Abbildung 6.7 illustriert. Realisiert ist das Domainmodel durch den Semantic Controller, der in Python geschrieben ist. Die Vorteile einer Interpretersprache sowie zahlreiche Projekte für die Verwaltung von RDF-Graphen mit einem SPARQL-Endpoint begründen den Vorzug gegenüber Programmiersprachen wie Java oder C#. Sobald ein Semantic Adapter angeschlossen wird, werden die Teilgraphen an den Semantic Controller gesendet und von ihm dem RDF-Graphen hinzugefügt. Durch den Eingang einer Blaupause wird der RDF-Graph nach den geforderten Eigenschaften durchsucht, wobei eine Metrik auf den Raum der Symbolsequenzen angewendet wird. Das Matching verknüpft die Blaupause mit den

konkreten Fähigkeiten und ergänzt fehlende Eigenschaften wenn nötigt. Sollten Fähigkeiten nicht von der angeschlossenen Hardware zur Verfügung gestellt werden können, werden die assoziierten GIT-Repositories verwendet, um den Code virtueller Fähigkeiten herunterzuladen und auszuführen. Als Beispiel dienen Fähigkeiten, wie die *Waage (W)* Fähigkeit aus Abbildung 6.3 oder das virtuelle Anemometer aus dem Experiment der Ausbreitungsmodellierung. Diese können wiederum selbst (virtuelle) Fähigkeiten in ihrer Prozessstruktur verwenden. Die Auflösung der Abhängigkeiten aufeinander aufbauender Fähigkeiten wird im späteren Verlauf des Kapitels vorgestellt. Wurden alle Anforderungen der Blaupausen erfüllt, können die Fähigkeiten in der OOD-API verwendet werden. Das Mapping ordnet die Fähigkeiten den entsprechenden Adaptern oder virtuellen Fähigkeiten zu und überprüft, auf welchem Adapter die Ausführung stattfinden soll. Folgend werden die Graphensuche und virtuelle Fähigkeiten detailliert vorgestellt.

6.4.1 Graphensuche

Die Graphensuche zur Bestimmung von Fähigkeiten, die in der Blaupause gefordert sind, erfolgt über einen SPARQL-Endpoint auf dem im JSON-LD Format gespeicherten RDF-Graph. Dabei wird das Python-Paket *RDFLIB* verwendet, das sowohl einen Parser für JSON-LD als auch für den Endpoint zur Verfügung stellt [318]. Die Persistenzschicht weicht hierbei von der intendierten Variante des MVVM ab, so wird das Model als RDF-Graph betrachtet und nicht als Klassenkonstrukt mit serialisierter Datenbank. Dieser Wechsel beruht auf der Eigenschaft erweiterbarer RDF-Graphen, deren Struktur nur teilweise vorgegeben ist. Eine Datenbank hingegen würde dem vorgegebenem Schema der Tabellenstrukturen basierend auf der Serialisierungsmethodik oder manuellen Definitionen folgen. Damit ein gemeinsamer Konsens für Abfragen geschaffen wird, dient die Ontologie des Semantic Hardware Web als Einstiegspunkt. So können einfache Hilfsmethoden wie die Helferklasse für die Abfrage aller physischen Devices durch folgende SPARQL-Query im Semantic Controller realisiert werden.

```
1  // Liste aller phyischen Devices
2  self.all_devices_in_Ontology = self.sparql_query("
3  SELECT ?devs
4  WHERE{ ?devs rdf:type commands:Device .}")
```

Für den Bedarfsfall stehen auch Fähigkeiten des Semantic Controllers für die OOD-API bereit, um direkt SPARQL-Queries auszuführen, falls bspw. eine detaillierte Graphensuche erforderlich ist. Im Falle der Instanziierung von Blaupausen werden mehrere SPARQL-Queries verwendet, um die Objekte zu befüllen, sofern die Eigenschaften mit den Literalen des RDF-Graphen übereinstimmen.

Damit auch Devices mit Fähigkeiten und einer ungefähren Beschreibung aufgelöst werden können, wie im Codebeispiel der *copter* mit den Fähigkeiten *FlyToPosition*, *GetPosition* und *LiftWeight*, werden alle Namen der virtuellen und physischen Fähigkeiten im Semantic Controller gesammelt. Innerhalb dieses Namensraums wird über die Levenshtein-Distanz [174] eine Bestimmung der Ähnlichkeit von Zeichenketten be-

stimmt. Dabei wird die minimale Anzahl (lvsd) einfügender, löschender und ersetzender Operationen für jede Permutation der Wortpaare bewertet.

Wort (a)	Wort (b)	lvsd	similarity
FlyPosition	FlyToPosition	2	0,85
GetPostiton	GetPosition	2	0,82
Weight	LiftWeight	4	0,6

In diesem Beispiel werden zwei Schritte für das Löschen der Buchstaben „*To*“ zwischen der geforderten Fähigkeit *FlyToPosition* und der im Graphen benannten *FlyPosition* benötigt. Die Gleichheit von 85% errechnet sich unter Einbezug der Wortlängen durch die Formel:

$$similarity = \frac{len(a) + len(b) - lvsd}{len(a) + len(b)} \tag{6.1}$$

Diese Lösung bietet sich bei minimalen Abweichungen in kleinen Datensätzen an. Sowohl der verwendete Datensatz als auch der Schwellwert können im Semantic Controller parametrisiert werden. Der Standartdatensatz umfasst lediglich das *rdfs:label* der Fähigkeiten, also den Namen, da die Komplexität in Kombination mit der Menge an Permutationen polynomial wächst. Die Toleranz minimaler Abweichungen adressiert eine typische Problemstellung aus dem Semantic Web. Es existieren viele Dictionaries, deren Inhalte nicht disjunkt sind und Literale können komplett frei gewählt werden. So wird in dem Beispiel aus Abbildung 6.3 das *Messen* als Fähigkeit betrachtet, und eine genauere Spezifikation findet sich in den Eigenschaften wieder, wohingegen die Fähigkeit *Fliege zu Position* eine implizite Semantik besitzt. Durch die Parametrisierung des zu durchsuchenden Datensatzes können bspw. alle rdfs:label beliebiger Tiefe anhand einer SPARQL-Query miteinbezogen werden. Dieses Verfahren bietet die Möglichkeit, eine anpassbare, automatisierte Suche mit dem Kriterium der Ähnlichkeit nach Levenshtein zu definieren, mit der Fallback-Strategie, den Graphen durch die Integration in das Semantic Web selbst zu durchsuchen.

6.4.2 Virtuelle Fähigkeiten

Virtuelle Fähigkeiten werden innerhalb eines Betriebssystems ausgeführt und haben genau eine Quelle für den Speicherort. Sie können eine Schnittstelle zu komplexen Systemen bieten, wie bspw. eine Microsoft Kinect (mit eigenem SDK) und sind im Technologiestack aus Abbildung 5.16 als *virtuelle Fähigkeiten* (VF) vertreten. Die Realisierung von virtuellen Fähigkeiten ist in zwei Kategorien unterteilt. Einerseits die Unterstützung für Pythoncode, der zur Laufzeit ausgeführt werden kann und nativ durch die Python-Basis des Semantic Controllers unterstützt wird, und andererseits die Kapselung beliebiger Programmiersprachen durch die Verwendung von Containern in Docker. Für beide Varianten ist ein Verweis auf ein GIT-Repository oder eine lokale Referenz erforderlich, der in Abbildung 6.8 über die Ontologie ausschnittsweise verzeichnet ist. Da für die Integration von Pythoncode keine Artefakte benötigt werden, ist lediglich eine Referenz auf die Datei mit dem GIT-Tag *latest*, bzw. der Pfad auf die entsprechende

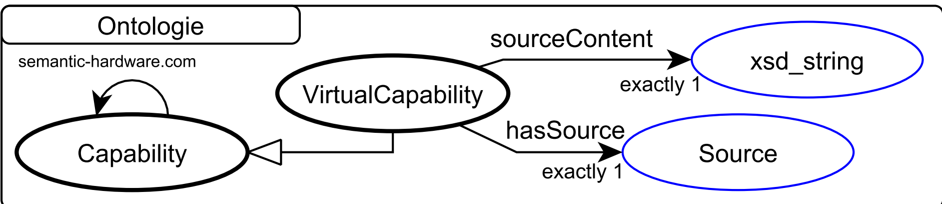

Abbildung 6.8. Ausschnitt aus der Semantic Hardware Web Ontology mit dem Fokus auf virtuelle Fähigkeiten und der Möglichkeit, Fähigkeiten zu kombinieren. Das Prädikat *hasSource* beschreibt den Typ des *xsd_string* über *sourceContent*. (Eigenwerk)

Datei zu definieren. Benötigt der Pythoncode hingegen Abhängigkeiten, die bspw. über das Python-Paketverwaltungsprogramm *PIP* installiert werden müssen, so bietet sich bereits die Containerisierung in Docker an. Für die Erstellung eines *Dockerimages* wird ein *Dockerfile* verwendet, deren Struktur wie nachfolgend beschrieben ist.

```
1  FROM python
2  COPY camera.py /var
3  COPY requirements.txt /var
4  EXPOSE 9999
5  RUN    python -m pip install -r /var/requirements.txt
6  CMD    python /var/camera.py
```

Zeile (1) definiert über *FROM* das Standard-Image von Docker für Python, das für unterschiedliche Plattformen zur Verfügung steht und als Basis für das aufzubauende Image dient. Über die Anweisung *COPY* in Zeile (2,3) werden die Daten für das Docker-Image kopiert, sowohl der Quellcode für die Integration der Kamera, als auch die benötigten Abhängigkeiten für die PIP Paketverwaltung in der *requirements.txt*. *EXPOSE* in Zeile (4) gibt den Kommunikationsport an, über den mit dem Container interagiert werden kann. Über den *RUN* Befehl in Zeile 5 werden die Abhängigkeiten in PIP installiert und es wird sichergestellt, dass der Container über den Befehl in Zeile (6) nach *CMD* ausgeführt werden kann. Für die Integration in den Semantic Controller werden lokal verfügbare und über ein GIT-Repository bereitgestellte Varianten vorgestellt, falls eine Internetfähigkeit auf dem Zielgerät nicht gewährleistet ist.

Lokale Dockerfile

Hat eine virtuelle Fähigkeit die Beziehung *hasSource DockerSource*, geht der Semantic Controller davon aus, dass in der Beziehung *sourceContent* eine in JSON kodierte Nachricht mit folgenden Attributen vorhanden ist:

```
1  port:     Kommunikationsport
2  name:     Name der Virtual Capability (Nicht zwingend die URI)
3  tag:      Version des Artefaktes
4  filepath:  Pfad des Dockerfiles zum Erstellen des Images.
```

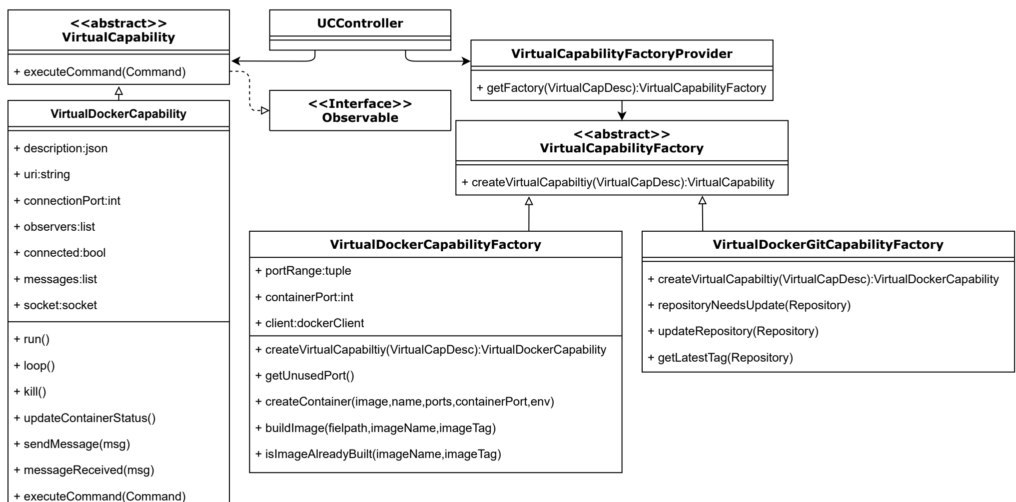

Abbildung 6.9. Designklassendiagramm für abstrakte Fähigkeiten nach dem Erzeugungsmuster *Abstract Factory*. Die Klasse *UCController* symbolisiert den Anschluss an die restlichen Klassen des Gesamtsystems. Vererbte Methoden des Interfaces sind nicht skizziert. (Eigenwerk in Anlehnung an die Ergebnisse der betreuten Abschlussarbeit von Moritz Hofer, Student ISSE.)

Beim Erstellen der Virtual Capability wird das Dockerfile im Ordner *filepath* mit dem zusammengesetzten Namen aus *name:tag* erstellt und die Kommunikation über den internen Containerport *(port)* realisiert. Dies setzt voraus, dass das Dockerfile mit allen benötigten Dateien lokal auf dem Hostrechner vorhanden ist.

Dockerfile via GIT

Wird in dem RDF-Graphen eine Virtual Capability als *hasSource* mit einer *DockerGitSource* referenziert, so wird ein Tupel aus dem Kommunikationsport und der URI des Repositories im JSON Format erwartet. Analog zur lokalen Dockerfile werden *name* und *tag* aus den Informationen des Repositories abgeleitet und der *filepath* wird durch den Ort des Checkouts bestimmt.

Im Semantic Controller wird für beide Varianten das Entwurfsmuster *Abstract Factory* verwendet, wie in Abbildung 6.9 illustriert, um die Art der Erzeugung vom Rest des Systems zu abstrahieren. Die abstrakte Klasse *Virtual Capability* bietet wie das Pendant der normalen Fähigkeit die Möglichkeiten der Interaktion über die *executeCommand* Methode und der Implementierung des *Observable* Interface. Die Klasse *VirtualDockerCapability* erbt von dieser und verwaltet den entsprechend assoziierten Container.

Die abstrakte Klasse *VirtualCapabilityFactory* bildet die abstrakte Fabrik und bietet die Funktion zum Erstellen einer *Virtual Capability* durch die Methode *createVirtualCapability* anhand der Beschreibung aus der Ontologie. Die beiden Klassen *VirtualDockerCapabilityFactory* und *VirtualDockerGitCapabilityFactory* erben von dieser Klasse und implementieren die jeweiligen Varianten.

Die Verwaltungsklasse *VirtualCapabilityFactoryProvider* ist als Singleton realisiert und dient als Schnittstelle für Hilfsfunktionen, wie die Rückgabe der korrekten *VirtualCapabilityFactory* basierend auf der Beschreibung.

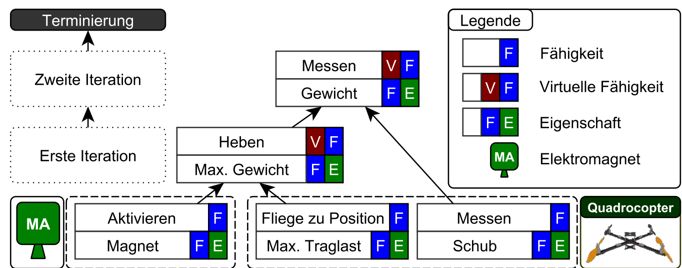

Abbildung 6.10. Beispiel für die Fähigkeit der Waage des Pick & Place-Experiments. Auf der untersten Ebene befinden sich Hardwarefähigkeiten, die über zwei Iterationsschritte zur virtuellen Fähigkeit *Messen* mit der Eigenschaft *Gewicht* zusammengesetzt werden. (Eigenwerk)

In der OOD-API ist nicht ersichtlich, ob es sich um eine virtuelle Fähigkeit oder Hardwarefähigkeit handelt. Die Erstellung und Ausführung wird komplett im Semantic Controller gekapselt. Für die Selbstassoziation der Fähigkeiten in Abbildung 6.8 wird folgend die Implementierung aufeinander aufbauender Fähigkeiten, die sogenannten kombinierten Fähigkeiten, vorgestellt.

Kombinierte Fähigkeiten

In der Ontologie des Semantic Hardware Web können Fähigkeiten auf anderen Fähigkeiten aufbauen. Diese werden in den RDF-Graphen angelegt und miteinander verknüpft. Die Realisierung auf der Implementierungsseite gestaltet sich allerdings nicht trivial. So kann der Graph über die Zeit wachsen und eine Vielzahl an verknüpften Einträgen bieten, von denen lediglich ein **Subset an Fähigkeiten** die vorhandene Hardware unterstützt. Dieses Subset muss aber nicht komplett instanziiert werden, da die Blaupause vorgibt, welche **Fähigkeiten ausgeführt** werden sollen.

Für das **Subset an Fähigkeiten**, dass die Hardwarekonfiguration bereitstellt, wird ein BottomUp-Ansatz verfolgt. Auf Basis der Fähigkeiten, die direkt an der Hardware verfügbar sind, wird der Graph in mehreren Iterationen durchsucht, bis keine weiteren Fähigkeiten mehr gebildet werden können. Im Beispiel aus Abbildung 6.10 wird aus den drei Basisfähigkeiten in der ersten Iteration lediglich die virtuelle Fähigkeit *Heben* mit der Eigenschaft *Max. Gewicht* gebildet. Erst in der zweiten Iteration kann demnach die Fähigkeit *Messen* mit der Eigenschaft *Gewicht* gebildet werden. Sobald keine weiteren Fähigkeiten mehr abbildbar sind, wird der Algorithmus beendet, wie im Beispiel nach der zweiten Iteration. Das Subset wird allerdings nicht automatisch heruntergeladen oder gestartet, sondern es wird auf die Anforderungen der Blaupause gewartet.

Stimmt eine Anforderung aus der Blaupause mit einer der möglichen Fähigkeiten aus dem Subset überein, so soll diese **Fähigkeit ausgeführt** werden. Dafür wird ein TopDown-Ansatz verfolgt, der bei der geforderten Fähigkeit startet und schrittweise alle Abhängigkeiten auflöst. Im Beispiel aus Abbildung 6.10 ist exemplarisch die Fähigkeit *Messen* mit der Eigenschaft *Gewicht* im Blueprint gefordert. Diese Fähigkeit wird gestartet, danach die virtuelle Fähigkeit *Heben* und *Messen* mit der Eigenschaft *Schub* und

zuletzt die Fähigkeiten *Aktivieren* und *Fliege zu Position*. Die Ressourcenallokation wird in der Bildung nicht beachtet, kann aber durch den BottomUp-Ansatz für das Subset durch eine ergänzende Ontologie erweitert werden.

Für die **Kommunikation zwischen den virtuellen Fähigkeiten** wird auf das Commands Protocol (ComP) mit einer minimalen Erweiterung Docker spezifischer Elemente zurückgegriffen, wie eine *isAlive* Nachricht für die Überprüfung, ob der Container gestartet ist oder noch läuft. Die Erweiterungen beziehen sich zu großem Teil auf die potentielle Rekonfiguration zur Laufzeit, die sich ebenfalls auf virtuelle Fähigkeiten auswirkt. Falls bspw. der Magnet entfernt wird, so werden auch die Fähigkeiten *Heben* und *Messen* mit der Eigenschaft *Gewicht* entfernt, nicht jedoch das Repository.

Der Semantic Controller dient als Middleware, in der die Wissensbasis über das Domainmodel aufgebaut und bereitgestellt wird. Durch die Trennung des ausführbaren Codes und der Selbstbeschreibung können **Eigenschaften erweitert** werden, ohne den Code zu ändern. Die **Obfuskation** des Graphen durch die einfache Objektstruktur erlaubt auch textuelle Abfragen, die mit der Levenshtein-Distanz gemapped werden. Falls erforderlich, kann der Graph auch direkt über den **SPARQL-Endpoint** angesprochen werden. Virtuelle Fähigkeiten erlauben einerseits die Kapselung **wiederverwendbarer Prozesse**, andererseits auch die Möglichkeit **aufeinander aufbauende Prozesse** zu definieren. Durch die Integration von GIT im **automatisierten Verteilungsprozess** ist ein Update von Fähigkeiten auch in verteilten Systemen möglich. Docker bietet durch zahlreiche Images eine breite Plattform für **unterschiedliche Betriebssysteme und Programmiersprachen** für die Implementierung virtueller Fähigkeiten. Die Realisierung der Hardwarefähigkeiten ist Bestandteil des verteilten Models sowie der Schnittstellen-API.

6.5 Model - Integration der Hardware

Das Model bildet die verteilte Wissensbasis, die durch den Semantic Controller bereitgestellt und von den Semantic Adaptern gesammelt wird. Der Semantic Adapter ist ein Softwareartefakt, das sowohl in Python als auch in C / C++ zur Verfügung steht. Die Python-Variante kann wie in Abbildung 6.11 neben dem Semantic Controller auf einem Einplatinencomputer ausgeführt werden und verwendet das Commands Protocol (ComP) für die Interaktion mit dem Semantic Controller, das neben der Kommunikation zwischen virtuellen Fähigkeiten auch den Austausch des RDF-Graphen umfasst. Die Pythonimplementierung basiert primär auf der Serialisierung der Objekte, die dadurch entsprechend einfach persistiert werden können. So ist die Implementierung einer dynamischen Eigenschaft für das persistente Speichern von Variablen durch eine serialisierte Klasse realisiert, die bspw. auf dem SD-Speicher abgelegt wird. Komplexer gestaltet sich die Schnittstelle sowie die Implementierung auf der Seite des Mikrocontrollers, da es keinen dedizierten, persistenten Speicher gibt und die Interpretation serialisierter Klassen sowohl die Leistung als auch den Programmierspeicher unnötig belasten. Für die Interaktion mit dem Mikrocontroller werden folgend das *Self Descriptive Protocol* (SDP) und die *Schnittstellen-API* vorgestellt.

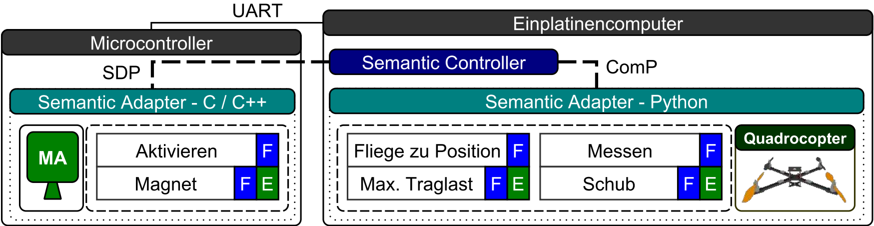

Abbildung 6.11. Verteilung der Softwareartefakte auf einen Mikrocontroller und einen Einplatinencomputer, die über UART kommunizieren. Die jeweils verwendeten Protokolle sind einerseits das Self Descriptive Protocol (SDP) und das Commands Protocol (ComP). Der Elektromagnet (MA) wird von einem Mikrocontroller adaptiert, währen der Quadrocopter mit einem Einplatinencomputer verbunden ist. (Eigenwerk)

6.5.1 Das Self Descriptive Protocol

Das **Self Descriptive Protocol** (SDP) ist speziell für die Verwendung auf Mikrocontrollern entwickelt, die über serielle Schnittstellen kommunizieren, wie bspw. der Arduino oder der ESP32. Die Herausforderung dabei ist, die serialisierte TCP-Kommunikation in einen bytebasierten Austausch zu übersetzen. Da UART eine sehr geringe Bitrate bietet, wie am Beispiel des Arduino Mikrocontrollers sind es zwischen 9600 oder 115000 bit/s (Baudrate ist in diesem Fall die Bitrate) gestaltet sich die Übertragung der Selbstbeschreibung als sehr zeitintensiv. Viele Mikrocontroller bieten zudem oftmals nur einen Hardwareinterrupt an, so dass eine Separierung in einen Daten- und einen Steuerkanal nur mit einem Softwareinterrupt unter Verwendung von zwei Kabelkanälen realisiert werden kann. Die Lösung für das Problem ist die Behandlung von Interrupts innerhalb des SDP. Zu diesem Zweck ist folgend eine kontextfreie Grammatik in der erweiterten Backus-Naur-Form aufgeführt, die exemplarisch die Verwaltung dynamischer Eigenschaften über das SDP repräsentiert.

```
1  Statement = '<',TypeByte,Payload,'>'
2  TypeByte = Dynamic|DynamicRes|DynamicReq|DynamicReqRes|
       DynamicDel|DynamicDelRes|Err|Write|WriteRes|Read|ReadRes
3  Dynamic = 'b',{Number},':'
4  DynamicRes = 'b',{Number},'@',{Number}
5  DynamicReq = 'c',{Number}
6  DynamicReqRes = 'c',{Number},':'
7  DynamicDel = 'd',{Number}
8  Err = 'e',{Number}
9  Payload = (Statement|ASCII_Byte|'<<'|'<>'),Payload|ε
```

Das Statement wird umschlossen von den Flagsymbolen „<,>“, die für den Beginn und das Ende eines Statements stehen. Die Rekursion in Zeile 9 findet anhand der Payload statt, die byteweise als ASCII_Byte übertragen wird, ein weiteres Statement umfasst oder die Flagsymbole escapen kann, indem ein „<“ verwendet wird. Die Selbstreferenz ermöglicht

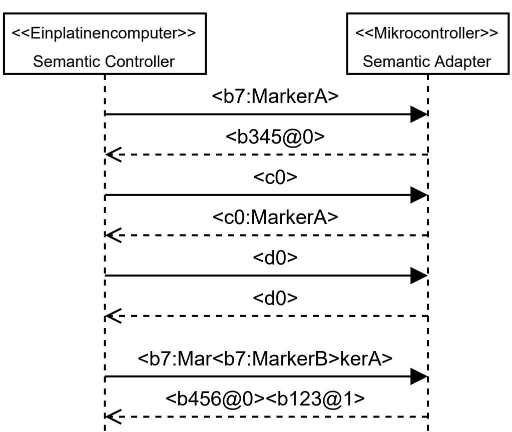

Abbildung 6.12. Beispiel eines Protokollablaufs anhand eines UML-Sequenzdiagramms. Die Softwareartefakte Semantic Controller und Adapter sind auf einen Einplatinencomputer und einen Mikrocontroller verteilt. (Eigenwerk)

eine Schachtelung beliebiger Tiefe. Durch das TypeByte können trotz Schachtelung keine zwei „<“ außer bei einem bewussten escapen auftreten. Das TypeByte in Zeile 2 bestimmt den Nachrichteninhalt und definiert die Rahmenbedingungen für die Payload. Repräsentanten für die Interaktion mit einer dynamischen Eigenschaft sind in den Zeilen 3-8 skizziert. Eine dynamische Eigenschaft kann zur Laufzeit auf dem Gerät angelegt, geändert und persistiert werden.

In dem Beispiel aus Abbildung 6.12 wird der String *MarkerA* mit dem *TypeByte Dynamic* übergeben, der die dynamische Eigenschaft der Kamera aus dem Pick & Place-Experiment repräsentiert. Durch die *Number* wird die zu übergebende Stringlänge definiert, die durch ein *’:’* von der Payload getrennt ist. Als Rückantwort wird das *TypeByte DynamicRes* verwendet, wobei die Nummer vor dem Trennungszeichen *’@’* eine Prüfsumme ist, die sich aus der Summe der ASCII-Werte des Strings modulo der Primzahl „1451“ ergibt. Die Zahl nach dem Trennungszeichen *’@’* bezieht sich auf den Index des Speicherbereichs, der reserviert wird. Das *TypeByte DynamicReq* in der Nachricht *<c0>* fordert das Auslesen dieses Speicherbereichs durch die *Number* an und erhält in der Antwort den String im Schema des *TypeByte DynamicReqRes* zurück. Im Beispiel wird der Index des persistenten Speicherbereichs wieder durch *TypeByte DynamicDel* freigegeben. Die letzte Nachricht des Beispiels zeigt die Verschachtelung von Nachrichten. Während der Übertragung des *TypeByte Dynamic* mit dem Wort „*MarkerA*“ wird ein „‚MarkerB‘“ gesendet und in den Bytestream geschachtelt (Interleaving). Im Semantic Adapter führt der innere Befehl zu einer Art Interrupt, indem nur vollständig erkannte Befehle verarbeitet werden. So bezieht sich die erste Antwort mit dem *TypeByte Dynamic* und die fiktive Prüfsumme „123“ auf den String *MarkerB* und die zweite auf *MarkerA*.

Neben dem Beispiel existieren weitere *TypeBytes* wie bpsw, für Streams, deren Frequenz definiert werden kann, oder Graphenrequests, die den RDF-Teilgraphen übertragen. In dem Beispiel der Verschachtelung wird nur der Request behandelt, bei Streams oder

der Graphenübertragung hingegen können auch Rückantworten verschachtelt werden, um auf einen Interrupt im Protokoll zu reagieren. Die Schnittstellenimplementierung des SDP im Semantic Controller adaptiert die Struktur und bildet die entsprechenden Instanzen des Commands Protocol für die Weitergabe. Im Semantic Adapter hingegen werden keine Klassenstrukturen für die Kapselung der Nachrichten verwendet, sondern es wird eine spezielle Schnittstellen-API angeboten.

6.5.2 Die Schnittstellen-API

Die Schnittstellen-API verbindet den proprietären Code der Hardware mit der semantischen Beschreibung des RDF-Graphen und dem Self Descriptive Protocol (SDP). Zwar ist die API speziell für die Arduino-Plattform entwickelt, sie kann aber auch auf andere Plattformen adaptiert werden. Als leitendes Beispiel wird die Integration der *Pixy* Kamera, sowie die dynamische Eigenschaft des Markers weitergeführt.

```
1  #include <Arduino.h>
2  #include <PythonComm.h> // Schnittstellen API
3  #include <SPI.h> // Sensorspezifisch - SPI
4  #include <Pixy.h> // Sensorspezifisch - Pixy API
5
6  String marker = ""; // Dynamische Eigenschaft
7  Pixy pixy; // Sensorspezifisch - Pixy
8
9  void setup() {
10   //Parameter: Fähigkeiten URI, Funktionszeiger
11   pythonComm.registerCaps("Measure", MeasurePosition);
12
13   pythonComm.registerDevice("Pixy");
14
15   // Parameter: URI, pointer der dyn. Eigenschaft, getter, setter
16   pythonComm.registerDynamix("MarkerURI", &marker, getMarker, setMarker);
17
18   pixy.init(); // Sensorspezifisch - Initialisierung
19 }
20 void loop() {
21   pythonComm.run(); // Regelmäßiger Aufruf
22 }
23 String getMarker(String s){
24   return marker;
25 }
26 String setMarker(String s){
27   marker = s;
28   pythonComm.saveDynamix(marker);
29   pixy.setMarker(marker); // Sensorspezifisch - Marker setzen
30   return marker;
31 }
32 String measurePosition(String s){
33   return pixy.getPos(); // Sensorspezifisch - Markerposition
34 }
```

Über den Header *#include <PythonComm.h>* wird die Schnittstellen-API integriert. Die restlichen Header, wie *SPI* oder *Pixy*, werden für die Implementierung der proprietären Kamera benötigt, die über die SPI-Schnittstelle des Mikrocontrollers angeschlossen ist. Die Methode *registerCaps()* bindet konkrete Fähigkeiten an einen Funktionspointer. Der erste Parameter ist das Label der Fähigkeit im RDF-Graphen. Die referenzierte Methode in Zeile 36 besitzt immer einen String als Parameter und Rückgabewert, der Cast in primitive Datentypen erfolgt manuell. Der Fähigkeitstyp hingegen wird anhand des Graphen beschrieben und über das SDP entsprechend definiert. Da ein Semantic Controller ein Device umfasst, ist die Methode *registerDevice* nicht von den Fähigkeiten abhängig und kann auch danach aufgerufen werden. Der Parameter *„Pixy"* in Zeile 13 referenziert wieder das Label des RDF-Graphen für das entsprechende Device.

Die dynamische Eigenschaft benötigt eine globale Variable, im Beispiel den *String marker* in Zeile 6, sowie die Get und Set Methoden. In Zeile 16 findet die Registrierung über die Methode *registerDynamix* statt. Die *„MarkerURI"* ist wiederum die Referenz auf den RDF-Graphen, die restlichen Parameter sind Pointer. In der Setter-Methode wird in Zeile 31 durch die *saveDynamix()*-Methode explizit die Variable in einen reservierten Speicherbereich des EEPROM persistiert. Der Grund für die explizite Methode ist die Funktion des SDP, dass dynamische Variablen auch nur im flüchtigen Speicher gehalten werden können das Speichern und erst durch ein Kommando geschieht. Im Beispiel hingegen ist das explizite Write-Kommando nicht nötig.

Für das Hochladen des Graphen findet sich ebenfalls ein spezielles Kommando im SDP, das analog zur dynamischen Eigenschaft funktioniert. So müssen im Code keine Selbstbeschreibungen definiert sondern lediglich die Labels referenziert werden.

Das kurze Beispiel illustriert den Umfang und die einfache Verwendung der Schnittstellen-API. Durch die Verknüpfung der Methoden über Funktionspointer wird sowohl die Kommunikation als auch die Interaktionsart gekapselt, wie bspw. ob es sich um einen Request, einen Stream oder einen asynchronen Aufruf handelt. Diese Informationen sind Bestandteil des RDF-Graphen, der im Semantic Controller interpretiert wird und die entsprechende Ansteuerung über das SDP an den Semantic Adapter kommandiert. Die verteilten RDF-Graphen aller Semantic Adapter bilden das Model in der MDDM-Architektur.

6.6 Zusammenfassung und Einordnung

Die MDDM-Architektur folgt der Separierung des Models und den Anforderungen der Domäne durch eine Zwischenschicht, dem Domainmodel. Modulare Hardwareelemente führen zu einer Verteilung des Models in einzelne Teilgraphen, die auch für virtuelle Fähigkeiten eingesetzt werden, wie in Abbildung 6.13 skizziert. Die Domain verwendet Blaupausen, in denen die Anforderungen der Domäne definiert sind. Das Mapping zwischen den Anforderungen und dem verteilten Model ist Bestandteil des Domainmodels, in dem nur die geforderten Fähigkeiten bereitgestellt werden. Die MDDM-Architektur ist in der Publikation [153] vorgestellt und in der Publikation [311] anhand einer Hausautomatisierung evaluiert.

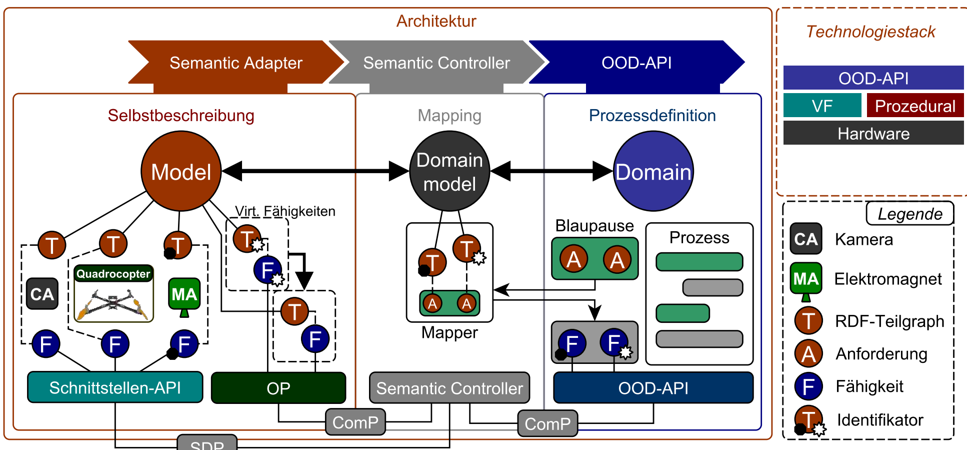

Abbildung 6.13. Übersicht der MDDM-Architektur mit Verortung der APIs für die Integration der Selbstbeschreibung. Das Model besteht aus verteilten Teilgraphen, die im Domainmodel mit den Anforderungen der Domain anhand einer Blaupause gemappt werden. Unten im Bild sind die verwendeten Protokolle für die Kommunikation zwischen den Artefakten skizziert. (Eigenwerk)

Der **Semantic Adapter** ist das Softwareartefakt, das die Selbstbeschreibung und Fähigkeiten verknüpft und bereitstellt. Die Selbstbeschreibung besteht aus einem RDF-Teilgraphen als Teil des verteilten **Models**. Für den Semantic Adapter gibt es zwei Implementierungen, einerseits für Mikrocontroller anhand der Schnittstellen-API und andererseits für objektorientierte Sprachen in Form einer Kapselung für virtuelle Fähigkeiten (VF). Die Variante für den Mikrocontroller implementiert die Schnittstelle des *Self Descriptive Protocol* (SDP) und bietet spezielle Kommandos an, die bspw. den Zugriff auf den EEPROM des Mikrocontrollers erlauben. Die objektorientierte Variante verwendet das Commands Protocol (ComP) mit der Ergänzung des Speicherortes der RDF-Teilgraphen. Die Realisierung des Semantic Adapters als selbstbeschreibendes Element auf einem Mikrocontroller in Kombination mit einer objektorientierten Programmiersprache ist in den Publikationen [84, 309] veröffentlicht. Das SDP mit einer Verlagerung der Funktionsbeschreibung und die Bereitstellung der Schnittstellen-API sind in der Publikation [311] verortet.

Der **Semantic Controller** ist das Bindeglied zwischen der Hardware und den Prozessen der OOD-API. Die Anforderungen innerhalb der Blaupausen werden unter Verwendung der Levenshtein Distanz mit dem Model abgeglichen und darauf aufbauend wird das **Domainmodel** erstellt. Dabei werden nicht nur die Fähigkeiten der Semantic Adapter verknüpft, sondern auch aufeinander aufbauende virtuelle Fähigkeiten aufgelöst. Die Auflösung der Fähigkeiten basiert auf einem BottomUp-Ansatz, wobei die Ausführung einer TopDown-Strategie folgt. Die Verwendung eines SPARQL-Endpoints innerhalb des Semantic Controllers für die Verknüpfung ausführbarer Fähigkeiten mit dem RDF-Graphen ist in der Publikation [309] anhand eines Experiments illustriert. Das Konzept der Blaupause und virtuelle Fähigkeiten sind in den Publikationen [153, 311] vertreten.

Die **OOD-API** offeriert eine Schnittstelle für Prozessdefinitionen, die stark von der eingesetzten **Domäne** abhängt. Anforderungen der Domäne können in der Klassenstruktur als Blaupause ausgedrückt und bei dem Semantic Controller angefragt werden. Diese neue Herangehensweise der *anforderungsbasierten Programmierung* zielt auf die Bedürfnisse der auf Fähigkeiten basierten Agenten der Multipotenten Systeme ab, die in der Publikation [153] veröffentlicht sind. Die Abstraktion erlaubt den einfachen Austausch der Hardware, ohne die Software zu ändern, sowie Rekonfigurationen, aus denen sich neue Fähigkeiten ergeben. Es wird auch eine Java API unterstützt, die auf der Robotics API [8] aufbaut und in den Publikationen [84, 309] vorgestellt ist. Die unabhängige OOD-API ist in der Publikation [311] veröffentlicht. An dieser Stelle sei auf die erfolgreiche Abschlussarbeit von Luca Alfano verwiesen, die im Rahmen der Dissertation betreut wurde und einen Anteil für die Realisierung der Architektur beigetragen hat.

Die einzelnen Artefakte, wie der Semantic Controller, Semantic Adapter, OOD-API, sowie die Schnittstellen-API sind Bestandteile des **Semantic Plug & Play**. So ist der Begriff als Sammlung verschiedener Bausteine zu interpretieren, die wie die Ontologie des *Semantic Hardware Web* ebenfalls als Open-Source-Projekt[2] unter der MIT Lizenz zur Verfügung stehen. Neben der zugrundeliegenden Architektur und den Artefakten des Semantic Plug & Play existieren noch zahlreiche Werkzeuge für die Integration des Robot Operating System (ROS), sowie für die Erstellung der RDF-Graphen, die im nächsten Kapitel vorgestellt werden. Das einleitende Pick & Place-Experiment wird in Abschnitt 8.6 durchgeführt und anhand der Ergebnisse bewertet.

[2]https://github.com/isse-augsburg/SemanticPlugAndPlay

Zusammenfassung. Semantic Plug & Play besteht aus mehreren APIs und Protokollen für die Realisierung der Selbstbeschreibung, die eine fähigkeitsbasierte Steuerung von Hardware ermöglicht. Im Forschungsbereich der Robotik hat sich das Robot Operating System (ROS) für die Ansteuerung der Hardware etabliert. In diesem Kapitel werden Werkzeuge präsentiert, die eine symbiotische Verwendung von ROS und Semantic Plug & Play ermöglichen sowie ROS um Plug & Play-Mechanismen erweitern. Abschließend wird eine modulare Simulation vorgestellt, die eine schrittweise Überführung von der Simulation in die Realität ermöglicht.

Semantic Plug & Play - Werkzeuge

7.1 SemanticSens - Erstellung einer Selbstbeschreibung 120
7.2 Integration von ROS 123
7.2.1 Experiment: Der sensorbasierte Flug 124
7.2.2 ROSSi - Modellierung von Fähigkeiten 126
7.2.3 Die Block Definition Language und das DMDE-Framework 132
7.3 Modulare Simulation 137
7.3.1 Experiment: GoLive 137
7.3.2 Selbstbeschreibung mittels digitalen Zwillings 139
7.3.3 Simulation 141
7.3.4 Verteilte MR-Virtualisierung und MR-Bausteine 143
7.4 Einordnung der Werkzeuge in die Methodologie 146

Mit Kapitel 5 wird ein durchgängiges Konzept für die Integration modularer Hardwareelemente mit Selbstbeschreibungen vorgestellt, die auf den Semantic-Web Technologien aufbaut. Verschiedene APIs und Protokolle unterstützen die in Abschnitt 5.4 vorgestellte Methodologie für die Integration. Die Realisierung der Werkzeuge, APIs und Protokolle wird in diesem Kapitel in der Reihenfolge der Methodologie vorgestellt. Beginnend bei der in Abbildung 7.1 dargestellten Annotation müssen RDF-Graphen erstellt werden.

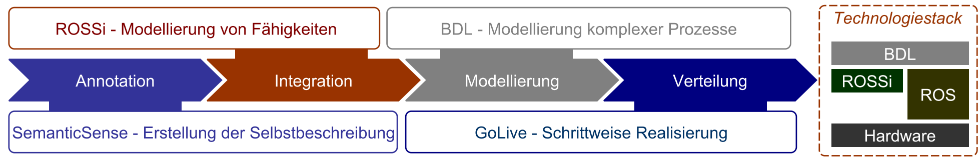

Abbildung 7.1. Methodologie von Semantic Plug & Play mit einer Zuordnung der Werkzeuge. Rechts abgebildet ist ein Ausschnitt des Technologiestacks aus Abbildung 5.16, der in diesem Kapitel eingehender vorgestellt wird. (Eigenwerk)

Für die Erstellung von RDF-Graphen im Semantic Web existieren bereits zahlreiche Werkzeuge [208]. Diese sind jedoch nicht speziell auf die Selbstbeschreibung des Semantic Plug & Play zugeschnitten und erfordern eine hohe Einarbeitungszeit. Daher wurde im Rahmen dieser Arbeit SemanticSens als ein Werkzeug für die Erstellung einer Selbstbeschreibung von Hardware, das im Rahmen der vorliegenden Dissertation entwickelt wurde, vorgestellt.

Speziell im Bereich der Robotik betritt Semantic Plug & Play zumindest kein Greenfield, da viele Projekte mit Ähnlichkeiten zur Fallstudie [89, 107, 261] auf den De-Facto-Standard ROS aufbauen. Eine Rekonfiguration mit Plug & Play und Selbstbeschreibungsmechanismen, wie sie in multipotenten Systemen in Abschnitt 3.2 definiert sind, ist jedoch ohne eine Anpassung von ROS nicht möglich. Das im Rahmen der Dissertation entstandene Projekt ROSSi bietet eine grafische Entwicklungsumgebung für die Programmierung von ROS-Nodes, die als Fähigkeiten in Semantic Plug & Play über den Semantic Adapter integriert werden können. Der grafische Launchfile-Editor ist im Gegensatz zum weit verbreitetem Tool ROSMOD [164] für die Konfiguration des ROS-Masters mit einer Bridge für ROS 1 Anwendungen für die Version ROS 2 ausgelegt.

Eine ebenfalls auf ROS 2 aufbauende Prozessdefinition ist durch die im Rahmen der Dissertation entstandene Block Definition Language (BDL) möglich. Über eine Tokensemantik in Anlehnung an UML-Aktivitätsdiagramme können Funktionsblöcke definiert werden, die eine Schnittstelle für externe Systeme wie bpsw. Multiagentensysteme bieten. Eine eigene DSL mit dem Namen *Dynamic Mission Definition and Execution* (DMDE) erweitert BDL für einfache Missionsabläufe mit mehreren Quadrocoptern, analog zu der Arbeit von Ramirez et al. [238], mit dem Unterschied, dass modulare Sensoren berücksichtigt werden und durch die Funktionsblöcke der BDL auf Umwelteinflüsse reagiert werden kann. Eine Besonderheit der BDL ist die Integration von Mikrocontrollern in das ROS-Universum, die eine Plug & Play-Strategie in ROS zur Laufzeit ermöglichen.

Für die Simulation und Visualisierung der Experimente werden digitale Zwillinge und MR-Bausteine, mit dem Ziel einer schrittweisen Realisierung, eingesetzt. Die digitalen Zwillinge werden sukzessive durch reale Prototypen ersetzt. Diese umfassen neben unterschiedlichen Platinen für die Quadrocopter auch gefräste Rahmenteile, 3D gedruckte Gehäuse und Platinen für die Integration diverser Sensoren und Aktuatoren, die Plug & Play-Mechanismen unterstützen und als Basis für die Experimente dienen. Die konkreten Prototypen der einzelnen Experimente werden hingegen in Kapitel 8 vorgestellt.

7.1 SemanticSens - Erstellung einer Selbstbeschreibung

Durch die semantische Annotation von Informationen im Semantic Web können große Wissensdatenbanken wie die DBPedia [200] als Grundlage für Suchmaschinen dienen, um bspw. assoziierte Informationen in Rich Snippets darzustellen [188]. Rich Snippets sind kleine Inhaltsauszüge von Webseiteninhalten auf den Suchergebnisseiten von Suchmaschinen wie Google. Der größte Knoten der Linked Open Data Cloud [196] (DBPedia) basiert auf automatisierten Annotationen mit der Semistruktur der Wikipedia als Ausgangsbasis. Unternehmen wie Facebook hingegen verwenden eine obfuskierte

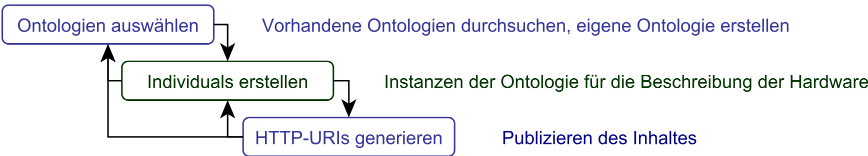

Abbildung 7.2. Iterativer Ablauf für die semantische Annotation in Anlehnung an das Wasserfallmodell [168]. Das Modell ist keine strikte Vorgabe der Phasen, sondern soll als Übersicht für den allgemeinen Ablauf dienen. (Eigenwerk)

Variante für den eigenen social graph [314], indem bspw. durch den Nutzer initiierte Aktionen eine Erweiterung des Graphen anstoßen. Auch wenn Formate wie JSON-LD die Einstiegshürde in die Welt des Semantic Webs senken, so ist die manuelle Annotation nicht gänzlich trivial, vor allem wenn Dinge außerhalb des Internets beschrieben werden sollen, wie etwa Hardware. Als Übersicht der Vorgehensweise für die manuelle Annotation dient das Modell in Abbildung 7.2, das an das Wasserfallmodell [168] angelehnt ist. Folgend wird das Modell speziell in der Domäne des Semantic Plug & Play erörtert, wobei der Ablauf generisch anwendbar ist.

Im ersten Schritt muss mindestens eine Ontologie als Vorlage für die zu beschreibenden Individuals[1] ausgewählt werden. Die *Semantic Hardware Web*[2]-Ontologie bildet dabei die Grundlage für die semantische Annotation der Eigenschaften und Fähigkeiten. Ergänzende Ontologien sollten entsprechend gegenstands- oder aufgabenbezogen selektiert werden, wie in der Einordnung des Semantic Hardware Web in Abbildung 5.9 aus Kapitel 5 skizziert. Als ein Beispiel wären für SI-Einheiten die OM Ontologie [249] zu nennen.

Individuals beschreiben die konkreten Instanzen der Ontologie, wie etwa einen Sensor, und können direkt in JSON-LD ausgedrückt werden. Werkzeuge wie das an der Stanford University entwickelte Protégé [208] bieten für die Instanziierung eine grafische Benutzeroberfläche an.

Das Resultat muss konform den Prinzipien von Tim Berners Lee [21] über HTTP-URIs bereitgestellt werden. Protégé bietet dafür eine Exportfunktion nach HTML an, in der jedes Individual auch eine HTTP-URI erhält. Die HTML-Seite hingegen muss selbstständig auf den entsprechenden Webspace hochgeladen und über einen Uniform Resource Locator (URL) zur Verfügung gestellt werden.

SemanticSens[3] ist im Rahmen der vorliegenden Dissertation entstanden und bietet im Gegensatz zu Protégé eine webbasierte Schnittstelle an, die speziell auf die Annotation von Individuals zugeschnitten ist, welche auf der *Semantic Hardware Web* Ontologie aufbauen. Anwender sollen über diese Plattform auf einfache Weise ihre Hardware semantisch annotieren und zur Verfügung stellen können, um anschließend das Ergebnis zu exportieren und als Selbstbeschreibung in Semantic Plug & Play zu

[1] Individuals sind die konkreten RDF-Graphen, die anhand einer oder mehrerer Ontologien erstellt werden.
[2] www.semantic-hardware.com
[3] www.semanticsens.de

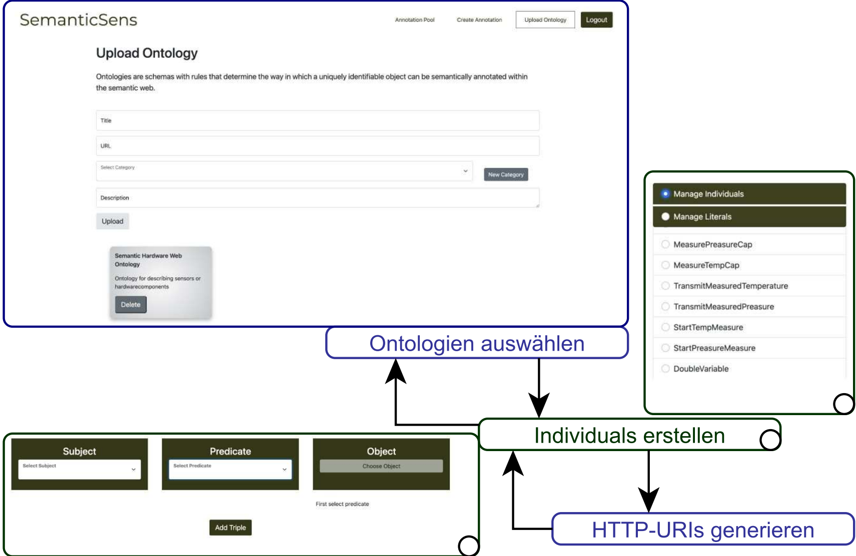

Abbildung 7.3. Abbildung des in SemanticSens umgesetzten iterativen Ablaufes mit Ausschnitten aus dem Frontend der Applikation. (Das Frontend stammt aus der betreuten Abschlussarbeit von Florian Timter, Student am ISSE)

nutzen. Der Vorteil einer Onlinelösung wie in SemanticSens ist, dass die HTTP-URIs automatisiert angelegt werden können, ohne einen eigenen Webspace bereitstellen zu müssen. Eine Nutzerverwaltung bietet die Möglichkeit, eigene Projekte zu erstellen, in denen konkrete Individuals angelegt werden können. SemanticSens ist als klassische 2-Schichten-Architektur umgesetzt, wobei Frontend und Backend über eine REST Schnittstelle [194] kommunizieren. Das Frontend ist in Vue.js programmiert. Vue.js ist ein clientseitiges JavaScript-Webframework für Single-Page-Anwendungen, das auf der MVVM-Architektur aufbaut [165]. Das Backend von SemanticSens ist in Python programmiert und verwendet das *django-rest-framework* für die Bereitstellung der REST Schnittstellen Endpunkte [293]. Die Endpunkte umfassen drei Kategorien:

Nutzerverwaltung
Damit ein einfacher Zugriff auf Projekte gewährleistet wird, können unterschiedliche Nutzer und Rollen angelegt werden. Normale Nutzer können sich registrieren und erhalten Zugang. Ein Endpoint für die Nutzerverwaltung stellt die CRUD-Operationen (Create, Read, Update, Delete), sowie ein Sessionmanagement basierend auf JWT Authentification Tokens [134] zur Verfügung.

Ontologiemanagement
Um neben der *Semantic Hardware Web*-Ontologie weitere Ontologien verwendbar zu machen, bietet ein Endpoint eine Importfunktion für OWL-Ontologien an. Das Webfrontend für den Import ist in Abbildung 7.3 zu sehen. Der Zugriff auf bereits importierte Ontologien für die Erstellung von Individuals wird ebenfalls über denselben Endpoint bereitgestellt.

Individuals
Ein Endpoint mit CRUD-Operationen für Individuals ermöglicht den Zugriff auf den intern verwendeten Quadstore, eine auf Quadrupel basierte Datenbank für RDF-Graphen, wobei zu den RDF-Tripeln eine Referenz zur verwendeten Ontologie gespeichert wird. Das verwendete Owlready2-Paket für den Quadstore verwendet eine Sqlite3-Datenbank für die Persistierung der Quadrupel [133]. Die Struktur der RDF-Syntax wird auch im Frontend berücksichtigt, wie in Abbildung 7.3 links unten zu sehen ist, wobei die Liste auf der rechten Seite der Übersicht dient. Jedes angelegte Individual bekommt automatisch eine zum Nutzer gehörige HTTP-URI, die öffentlich zugänglich ist. Für die Selbstbeschreibung können die Graphen der Individuals im JSON-LD Format exportiert werden.

Die Trennung des Front- und Backends durch eine REST-Schnittstelle erlaubt eine einfache Adaption an zukünftige Entwicklungen. So können bspw. auch Standalone-Anwendungen direkt auf die REST-Schnittstelle zugreifen. SemanticSens bietet nur ein kleines Subset der Fähigkeiten von Protégé an, das jedoch auf das Einsatzszenario der Instanziierung von Individuals passend zugeschnitten ist und eines der Hauptprobleme der manuellen Annotation im Semantic Web löst. In der MDDM-Architektur aus Kapitel 6 werden die Individuals des exportierten RDF-Graphen verwendet, um Fähigkeiten mit ausführbarem Code zu verknüpfen. Die Erweiterung von ROS zur Unterstützung von Fähigkeiten wird im folgenden Kapitel vorgestellt.

7.2 Integration von ROS

ROS ist ein De-Facto-Standard für die Steuerung von Robotersystemen in der Forschung [62]. Die schlanke Open-Source-Middleware bietet eine Vielzahl an externen Paketen, wie bspw. OpenCV oder die Simulationsumgebung Gazebo [236]. Ähnlich zu ROS unterstützt auch Semantic Plug & Play eine einfache Integration anderer Software und bietet Mechanismen für die Verteilung und Selbstbeschreibung. Da Dockercontainer beliebigen Code ausführen können und die Commandstruktur CRUD-Operationen für die Verwaltung der Containern unterstützt, können ebenfalls dedizierte ROS-Nodes durch Launchfiles gestartet werden. Das *Commands Protocol* (ComP) wird dabei lediglich für die Instanziierung der Infrastruktur und Verwaltung der Selbstbeschreibung genutzt, während die ROS-Nodes eine eigene Kommunikation verwenden, wie in Abbildung 7.4 illustriert.

Wie bereits in Kapitel Kapitel 4 erwähnt, werden in ROS 1 sogenannte Launchfiles für das parametrisierte Starten einzelner Nodes verwendet. Hierin liegt einer der Hauptunterschiede zu ROS 2, da ein Launchfile in ROS 1 eine besondere Rolle innerhalb der Architektur einnimmt. In ROS 1 wird anhand einer XML-Syntax ein Launchfile erstellt, das wiederum automatisch den *roscore* startet und bei einem unerwarteten Beenden einer einzelnen Node automatisch alle im Launchfile definierten Nodes beendet [236]. Für das Starten komplexer Anwendungen können ferner Launchfiles ineinander verschachtelt werden. In ROS 2 hingegen wird ein Launchfile als einzelne Node betrachtet, die bspw. mit Python programmiert werden kann und dadurch auch dynamische Aspekte ermöglicht, wie Schleifen oder bedingte Ausführungen. Dadurch können auch Schwär-

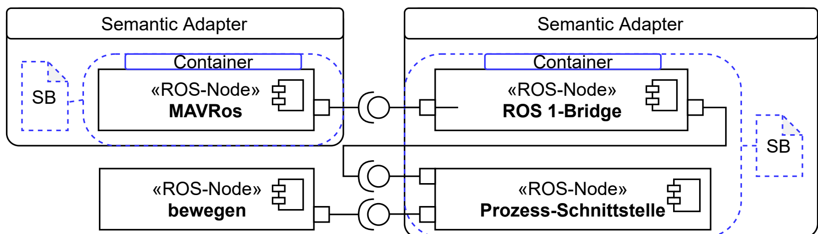

Abbildung 7.4. Exemplarischer Aufbau zusammenhängender ROS-Nodes, die über den Semantic Adapter gekapselt und mit einer Selbstbeschreibung (SB) ergänzt sind. Die Kommunikation der Nodes findet über ROS statt. (Eigenwerk)

me mit variierender Teilnehmeranzahl realisiert werden, da ein Ausfall in der Launchfile behandelt werden kann und kein kompletter Abbruch des Prozesses erfolgen muss [14]. Über eine zur Verfügung gestellten Bridge, wie in Abbildung 7.4 skizziert, können nun die Vorteile beider ROS-Versionen in einer Anwendung genutzt werden. **ROSSi** widmet sich der Erstellung und Verwaltung von Launchfiles und Nodes in ROS 2, die durch Python statt XML und der ROS 1-Bridge deutlich an Komplexität zugenommen haben. Die **Block Definition Language** (BDL) hingegen fokussiert Ausführungen für Schwärme mit einer **Dynamic Mission Definition and Execution** (DMDE) und einer automatisierten Verteilung zur Laufzeit. Einleitend wird das Experiment des sensorbasierten Fluges in einer ROS-Realisierung vorgestellt.

7.2.1 Experiment: Der sensorbasierte Flug

In diesem Experiment wird der sensorbasierte Flug untersucht, im speziellen die Reaktionszeit und Genauigkeit des eingesetzten Sensors. Als Vergleich dienen zwei Sensoren derselben Klasse, die eine relative Distanz zum Boden ermitteln können. Einerseits wird ein Ultraschallsensor, mit einem breiten, kegelartigen Messbereich und andererseits ein Lasersensor mit punktueller Messung verwendet. Um die Eigenschaften der Genauigkeit und Geschwindigkeit beider Sensoren im Praxistest zu verifizieren, wird ein externes Trackingsystem verwendet, das sowohl eine hohe Präzision als auch eine schnelle Abtastrate gewährleistet. In Abbildung 7.5 ist ein optisches, externes Trackingsystem skizziert, das diese Eigenschaften besitzt und über Triangulation die Position von Markern im Millimeterbereich messen kann. Die Marker sind sowohl auf dem Quadrocopter angebracht, wie in Abbildung 7.5 skizziert, als auch an den Hindernissen, um deren Position zu bestimmen und in einem gemeinsamen Bezugskoordinatensystem ausdrücken zu können. Damit soll zum einen der Verlauf der sensorbasierten Trajektorie, zum anderen soll die Abweichungen der Position Δ x und Δ z in Abhängigkeit zu den Hindernissen, der Zeit Δ t und der Geschwindigkeit v des Quadrocopters erfasst werden. Damit der Quadrocopter das Experiment nicht durch eine mangelnde Positionssteuerung negativ beeinflusst, wird das Trackingsystem zusätzlich als GPS-Ersatz für die Positionssteuerung verwendet und die Steuerungssoftware des Quadrocopters entsprechend angepasst. Die Details der Laborausstattung sowie das konkrete Trackingsystem werden in Abschnitt 8.1 vorgestellt.

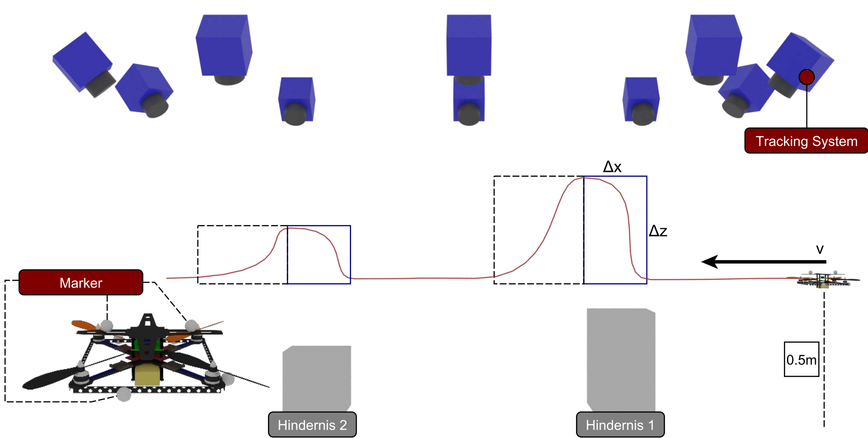

Abbildung 7.5. Aufbau des Experiments mit erwarteter Trajektorie, die durch die Hindernisse beeinflusst wird. Die Positionsabweichung während des Fluges wird mit Δ x und Δ z ausgedrückt unter der Bedingung, dass der relative Abstand von einem halben Meter zum Boden während des Fluges eingehalten wird. Das externe Trackingsystem registriert die Marker und bestimmt die Position für die Auswertung. (Eigenwerk)

Das Experiment unter Verwendung der Robotics API [8] ist in den Publikationen [84, 309] veröffentlicht. In diesem Kapitel wird das Projekt in ROS, mit einer Integration in Semantic Plug & Play überführt. Die komplexe Verteilung auf die Hardware des Experiments ist in Abbildung 7.6 skizziert, wobei jede Hardware eine bestimmte Rolle im Gesamtsystem erfüllt. Der Einplatinencomputer (1) am Quadrocopter dient als Plattform für die Flugregelung und kann Steuerbefehle umsetzen, wie etwa die Vorgabe einer Position. Die für die Positionsregelung benötigte Ist-Position des Quadrocopters wird mittels Mikrocontroller (a) über WLAN von der ROS-Node mit dem Namen *Tracking* empfangen. Die ROS-Node *MAVRos* sendet ebenfalls über den Mikrocontroller (a) die Soll-Position. Über die *ROS 1-Bridge* kann die ROS-Node *Regelung* die Soll Positionen anhand der Sensordaten erstellen. Die Sensoren sind mit der Schnittstellen-API erstellt und gemäß der MDDM-Architektur integriert. Der Semantic Adapter ist für beide Sensoren auf den Mikrocontrollern (b) und (c) verteilt, die mit dem Semantic Controller des Einplatinencomputers (2) kommunizieren. Durch die Kapselung der ROS-Nodes in einen Semantic Adapter können nicht nur die Artefakte in einer automatisierten Containerstruktur verteilt werden, sondern auch die Messdaten der Sensoren in die Regelung einfließen. Einerseits kann damit die Rekonfiguration des Sensors zur Laufzeit stattfinden, andererseits können die ROS-Pakete für die Steuerungslogik und das Trackingsystems genutzt werden. Dieses Vorgehen wird anhand der zwei Projekte *ROSSi* und *BDL* detailliert vorgestellt.

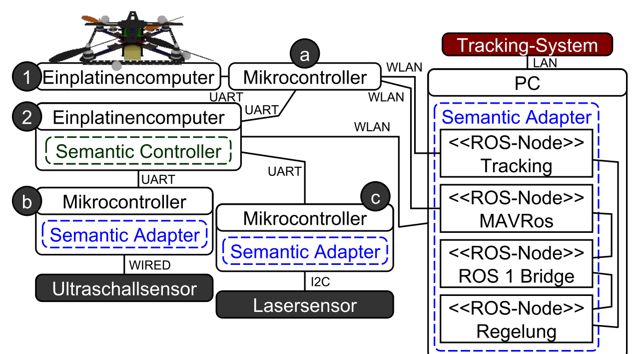

Abbildung 7.6. Übersicht der eingesetzten Einplatinencomputer und Mikrocontroller für das Experiment. Die ROS-Nodes befinden sich auf einem PC, der über WLAN mit dem Quadrocopter kommuniziert. Intern kommunizieren die Nodes über TCP-Sockets. (Eigenwerk)

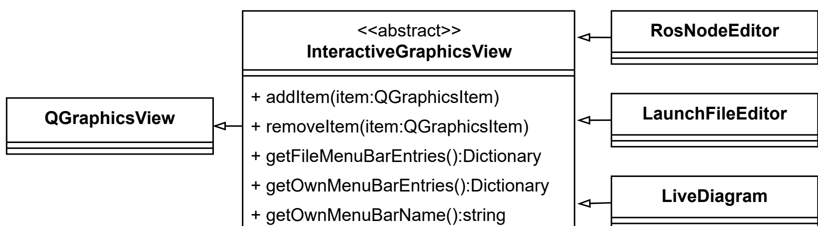

Abbildung 7.7. Aufbau der Kernfunktionalität von ROSSi durch die abstrakte Klasse *InteractiveGraphicsView*, von der die Typen *RosNodeEditor*, *LaunchFileEditor* und das *LiveDiagram* erben. (Eigenwerk)

7.2.2 ROSSi - Modellierung von Fähigkeiten

ROS Simple (ROSSi) ist eine grafische Entwicklungsplattform für die Modellierung von Nodes und Launchfiles in ROS, die im Zusammenhang mit dieser Arbeit entstanden ist. In ROS existiert bereits ein Basispaket für die Graphenvisualisierung zusammenhängender ROS-Nodes, der sogenannte RQT Graph [214]. Dieser erlaubt jedoch keinerlei Interaktion, Modellierung oder Visualisierungen dynamischer Abläufe. In dem Projekt RoBMEX [167] können Nodes und Launchfiles modelliert werden, jedoch nicht anhand einer grafischen Oberfläche, die Diagramme auf Basis der UML verwendet. ROSSi erlaubt nicht nur eine statische Modellierung der Nodes und Launchfiles, sondern auch die Inspektion des Programmablaufs durch eine dynamische Visualisierungen zur Laufzeit, wie es bspw. auch über Blueprints der Unrealengine möglich ist [294]. Für die softwaretechnische Umsetzung des ROSSi-Editors wird ein Diagramm-Editor mit dem Klassennamen *InteractiveGraphicsView* definiert, der von allen Diagrammtypen verwendet wird, wie in Abbildung 7.7 illustriert. Der Diagramm-Editor bietet die Möglichkeit, via Drag-and-Drop unterschiedliche Bausteine miteinander zu assoziieren oder CRUD-Operationen auf den Assoziationen sowie den Bausteinen auszuführen. Jeder Diagramm-Editor kann in beliebiger Anzahl geöffnet werden, so soll gewährleistet sein,

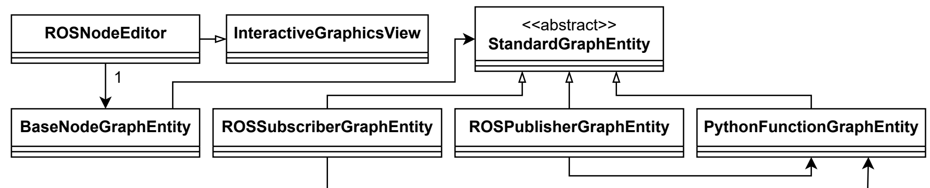

Abbildung 7.8. Aufbau des *ROS-Node-Editor* mit unterschiedlichen Entitäten. Die Entitäten bilden sowohl grafische Elemente als auch ein Codeskelett für die Codegenerierung. (Eigenwerk)

dass bspw. in einem *LaunchFile-Editor* eine Übersicht der Nodes gegeben ist, während mehrere *ROS-Node-Editoren* die Implementierung der Nodes visualisieren. Die Organisation mehrerer Editoren findet über das aus Browsern bekannte Tabsystem statt. Bei der Erstellung eines neuen Tabs wird der Diagrammtyp erfragt. Durch die Trennung des Diagramm-Editors und der Integration abstrakter Methoden nach dem Entwurfsmuster *Template Method* sind Erweiterungen für ROSSi einfach realisierbar. Die Codebasis ist in Python mit PyQT5 realisiert und benötigt keine zusätzlichen Abhängigkeiten, die nicht auch für ROS 2 erfüllt werden müssen, um die Richtlinien für Standardpakete zu erfüllen.

Node-Editor

Der Node-Editor bietet die Möglichkeit, einzelne Nodes mit den Standardfunktionen, wie etwa einem Publisher oder Subscriber von ROS, zusammenzusetzen und das Codeskelett zu generieren, bzw. bereits implementierte Fragmente einzusetzen. Die unterschiedlichen Entitäten erben, wie in Abbildung 7.8 dargestellt, von der abstrakten Klasse *StandardGraphEntity*, die von dem Basiselement, der *BaseNodeGraphEntity*, verwendet wird. Alle Entitäten besitzen auch eine Assoziation zu einem Codeskelett, das für die Codegenerierung verwendet wird. Neben der Erstellung von Entitäten können auch bereits gespeicherte Entitäten über ein Menü via Drag-and-Drop wiederverwendet werden. Folgende Entitäten und Komponenten bilden die Grundlage für den Node-Editor:

Basiskomponente
Die Basiskomponente ist in Abbildung 7.8 durch die Klasse *RosBaseNodeGraphEntity* repräsentiert. Für die Codegenerierung bildet die Basiskomponente das umschließende Element einer Node und ist somit die Grundlage für *StandardGraphEntity*-Entitäten. Der Name der Basiskomponente sowie die internen Entitäten sind in einem editierbaren Format visualisiert.

Funktions-Entität
Die Funktions-Entität bildet das Grundgerüst für die eigene Implementierung und ist in Abbildung 7.8 durch die Klasse *PythonFunctionsGraphEntity* vertreten. Sie dient der Kapselung interner Codebausteine, die in sich abgeschlossen und wiederverwendbar sind. Durch eine exemplarische Implementierung und geeignete Dokumentation können Funktions-Entitäten als Grundlage für die domänenspezifische Realisierung einer virtuellen Fähigkeit verwendet werden, die ebenfalls über Drag-and-Drop in die entsprechende Node eingeführt wird.

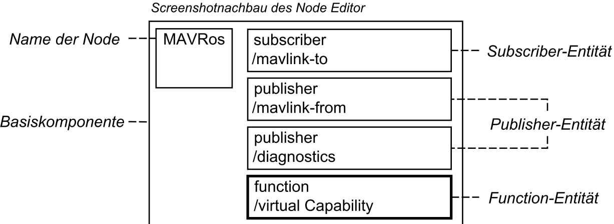

Abbildung 7.9. Nachbildung eines Screenshots des Node-Editors in einer Vektorgrafik für eine bessere Auflösung. Der Node-Editor zeigt eine MAVRos-Node mit einer Subscriber und zwei Publisher Entitäten. Über einen Doppelklick kann der Code einer Entität in externen IDEs bearbeitet werden. (Eigenwerk)

Publisher- & Subscriber-Entitäten
Eine besondere Funktions-Entität bilden Publisher und Subscriber, die ROS spezifisch ein Topic bereitstellen oder sich darauf registrieren. Die Klassen *ROSSubscriberGraphEntity* und *ROSPublisherGraphEntity* verwenden dafür die Funktions-Entität und ergänzen die Parameter Taktfrequenz und den Namen der Topic. Somit wird die Codehülle für die Kommunikation in ROS erzeugt, der Funktionsinhalt muss hingegen programmiert werden.

Parameter
ROS-Nodes können durch Werte aus dem sogenannten Parameter-Server des ROS-Systems von außerhalb konfiguriert werden. Die Parameterkomponente ermöglicht es dem Nutzer, auch diesen Aspekt von Nodes innerhalb des Node-Editors durch die Vergabe eines Namens in Kombination mit einem Standardwert zu verwenden. Wird der Wert des Parameters zum Beispiel außerhalb durch ein Kommandozeilenbefehl geändert, so wird dieser anstatt des Standardwerts innerhalb der laufenden Node verwendet.

Erweiterungen
Auf Basis der Funktions-Entität können beliebige Erweiterungen für den Node-Editor angelegt werden. Eine vorgefertigte Funktions-Entität ist die Anbindung an Semantic Plug & Play durch virtuelle Fähigkeiten, in welcher das *Commands Protocol* (ComP) genutzt wird. Damit können reine ROS-Anwendungen einfach in die Struktur von Semantic Plug & Play eingegliedert werden. In dem Beispiel aus Abbildung 7.9 wird die *MAVRos*-Node injiziert und die virtuelle Fähigkeit *Fliege zu Position* kann in der *function* mit dem Namen *virtual Capability* implementiert werden. Das *ComP* wird dadurch innerhalb der *MAVRos*-Node in *MAVLINK*-Nachrichten adaptiert. Für die Selbstbeschreibung der virtuellen Fähigkeit muss ein RDF-Graph angelegt werden. Die Schnittstellenanbindung des ComP in Ergänzung mit der Selbstbeschreibung bildet den Semantic Adapter, der in Abbildung 7.6 skizziert ist.

Der Node-Editor ist durch seine Struktur unabhängig von Semantic Plug & Play und kann für die Organisation wiederverwendbarer Entitäten genutzt werden. Erweiterungen erlauben die Injektion einzelner Nodes sowie komplexe Strukturen für virtuelle

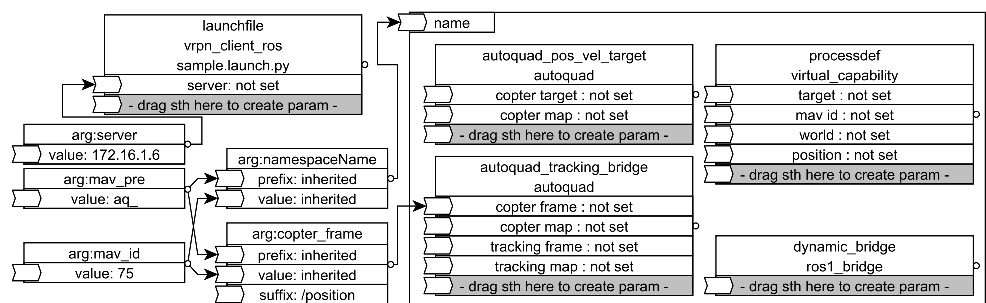

Abbildung 7.10. Ausschnitt einer in ROSSi erstellten Launchfile im Launchfile-Editor in Vektorgrafik. Das Modell zeigt eine nicht finalisierte Integration des Quadrocopters mit einer Steuerung über eine MAVLINK-Bridge. Der Flugcontroller des Quadrocopters heißt *Autoquad*. (Eigenwerk)

Fähigkeiten. Die Kommunikation und etwaige Abhängigkeiten zwischen den Nodes fokussiert der *Launchfile-Editor*.

Launchfile-Editor

Der Zusammenschluss und die Parametrisierung einzelner Nodes findet in ROS mittels einer *Launchfile* statt. Durch die Möglichkeit dynamischer Parameter und Schachtelungen verschiedener Launchfiles über Namespaces können diese sehr komplex werden. In Abbildung 7.10 ist eine Abbildung des Launchfile-Editors, mit einer dynamischen Parametrisierung und vier Nodes zu sehen, wobei drei Nodes in einem separaten Namespace liegen. Die unterschiedlichen Komponenten des Launchfile-Editors sind nachfolgend aufgeführt.

Parameter-Komponente
Die *Parameter Komponente* bietet drei optionale eingehende Ports, die eine Generierung der Parameter mit einem *Präfix, Value und Suffix* ermöglichen. Konkret können damit beispielsweise mehrere Devices instanziiert und unterschiedlich benannt werden. In dem Beispiel aus Abbildung 7.10 wird der Parameter *copterframe* der Node *autoquad_tracking_bridge* durch die Parameter Komponente *arg:copter_frame* generiert, die sich aus *arg:mav_id* und *arg:mav_pre* ergeben. Das Resultat für den Parameter *copter frame* ist demnach *„aq_75“*. Dieses Konstrukt ist sehr hilfreich, wenn mehrere Quadrocopter eingesetzt werden sollen, da bspw. auch die Namen der Namespaces eindeutig sein müssen, der sich im Beispiel analog zusammensetzt.

Node-Komponente
Eine Node kann mit beliebig vielen Parametern versehen werden, die in dem Editor aus Abbildung 7.10 am Ende der Liste hinzugefügt werden können. Die Parameter können hingegen nur von einer Parameter-Komponente gesetzt werden, wie bspw. die IP des Parameter *server* in der *launch-file - sample.launch.py*. Der Name der Node befindet sich in der ersten Zeile, die zweite Zeile bezieht sich auf den Namespace der Vorlage.

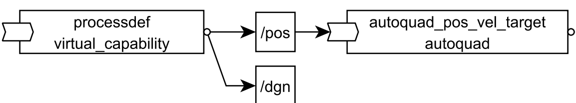

Abbildung 7.11. Die ROS-Node *processdef* bietet über zwei Publisher die Topics */pos* und */dgn* an, wobei die Node *autoquad_pos_vel_target* nur das */pos* Topic über einen Subscriber abonniert hat. (Eigenwerk)

ROS Representation Model
Damit die einzelnen Nodes samt Parameter auch wiederverwendet werden können, wird über das *Representation Model* eine Auswahl aller Nodes der verfügbaren ROS-Pakete bereitgestellt. Diese können wiederum mit Drag-and-Drop in den Launchfile-Editor eingefügt werden. Die Integration erfolgt durch ein automatisiertes Mapping des Quellcodes und durch ein Blackboxtesting für externe, kompilierte Nodes, das im Rahmen der Dissertation nicht weiter ausgeführt wird.

Launchfile-Komponente
Externe Launchfiles können über die *Launchfile Komponente* hinzugefügt werden, deren Struktur analog zu den Nodes aufgebaut ist. Die Benennung erfolgt durch den Stereotyp, gefolgt von dem Namen und der konkreten Datei, wie im Beispiel aus Abbildung 7.10 die externe Launchfile *vrpn_client_ros*.

Namespace-Komponente
Ein Namespace ist ein Kapselelement, in welchem eine lokale Eindeutigkeit umschlossener Elemente gilt. So kann eine Node mit demselben Namen in zwei unterschiedlichen Namespaces existieren. Die *Namespace Komponente* bildet diesen Sachverhalt durch eine grafische Repräsentation mit lediglich einem Parameter, dem Namen, ab, der in Abbildung 7.10 anhand der *Parameter Komponenten* erstellt wird.

Im Launchfile-Editor können die Bausteine für virtuelle Fähigkeiten aus dem Node-Editor parametrisiert werden, wie in Abbildung 7.10 die virtual_capability *processdef*, in der sich ein Protokolladapter befindet. Diese Node befindet sich allerdings in Abhängigkeit zu einer ganzen Reihe an weiteren Nodes, einer externen Launchfile und Parametrisierung. Ein naheliegender möglicher Lösungsansatz ist, das entweder das Gesamtergebnis der Modellierung in einen Dockercontainer überführt wird, oder die Abhängigkeiten in einer separierten Beschreibungssprache wie der Block Definition Language (BDL) behandelt werden. Für die Laufzeitüberwachung der einzelnen Nodes in ROS bietet ROSSi zusätzlich ein interaktives Live-Diagramm an.

Live Diagramm

Das im Rahmen dieser Arbeit entstandene *Live Diagramm* bietet im Gegensatz zum *RQT-Graphen* aus der Standardbibliothek von ROS [214] nicht nur eine Visualisierung der Struktur. Es können auch dynamische Aspekte, wie Änderungen der Systemkonfiguration zur Laufzeit oder eine Analyse der Interaktionen untersucht werden. Die Struktur der einzelnen Nodes wird automatisch generiert, wobei ein Algorithmus in

Anlehnung an Fruchterman et al. [90] die einzelnen Objekte verteilt: Der auf physikalischen Kräften beruhende Spring Embedders Algorithmus für Graphen kann innerhalb des *Live-Diagramm* parametrisiert werden, um die abstoßenden und anziehenden Kräfte der einzelnen Elemente zu definieren. Insgesamt werden drei unterschiedliche Elemente betrachtet: *Namespaces*, *Nodes* und *Topics*. Die *Namespaces* umhüllen analog zum *Launchfile-Editor* mehrere Nodes, haben für die Semantik aber keine Bedeutung. *Nodes* bieten alle Publisher über *Topics* an und werden vom Algorithmus graphisch auf der rechten Seite positioniert, Subscriber hingegen werden auf der linken Seite aufgeführt. In dem Beispiel aus Abbildung 7.11 bietet die Node *processdef* zwei Topics über Publisher an, die Node *autoquad_pos_vel_target* ist auf ein Topic mit einem Subscriber registriert. Treten zur Laufzeit strukturelle Änderungen des Systems auf, so wird der Graph aktualisiert und die assoziierten Elemente werden automatisch entfernt bzw. hinzugefügt. Einzelne Topics können selektiert werden, wobei sich ein generischer Subscriber im System anmeldet und als Sniffer (ermöglicht das Auslesen des Datenverkehrs) interagiert. Die Daten können damit zur Laufzeit ausgelesen werden, um Fehlerquellen diagnostizieren zu können, die sonst in ROS nur mit Netzwerkanalysetools wie bspw. Wireshark erkennbar sind.

Die Ebenen in ROSSi fokussieren unterschiedliche Aspekte und sind dennoch zusammenhängend. So können die Nodes im Node-Editor erstellt, im Launchfile-Editor parametrisiert und zur Laufzeit anhand des Live-Diagramms überwacht werden. Die erweiterten Abhängigkeiten von Kastens et al. [138] aus Abbildung 5.11 in Kapitel 5 zwischen dem Original und der Modellierung ist bidirektional realisiert. Aus dem Modell kann ein Codeskelett generiert werden, das befüllt und in der Modellierung wiederverwendet werden kann. Bereits existierende Implementierungen hingegen werden als Modell abstrahiert und in ROSSi zur Verfügung gestellt. Diese bidirektionale Realisierung bezieht sich auch auf Änderungen, die beidseitige Aktualisierungen beachten. ROSSi ist als Open-Source[4] unter der MIT-Lizenz verfügbar und wurde in der Publikation [312] veröffentlicht. An dieser Stelle sei auf die erfolgreiche Abschlussarbeit von Sebastian Rossi verwiesen, die im Rahmen der Dissertation betreut wurde und einen erheblichen Anteil an der Realisierung des gleichnamigen Projektes beigetragen hat.

Die *Block Definition Language* (BDL) dient als Grundlage für das Dynamic Mission Definition and Execution (DMDE)-Framework, in dem Quadrocopter-Missionen in ROS geplant und ausgeführt werden können. Der Fokus liegt dabei auf der verteilten Ausführung und Orchestrierung von ROS, sodass die Bündelung zusammenhängender Nodes in einem Semantic Adapter nicht mehr in einem Container zusammengefasst werden muss.

[4]https://github.com/isse-augsburg/ROSSi

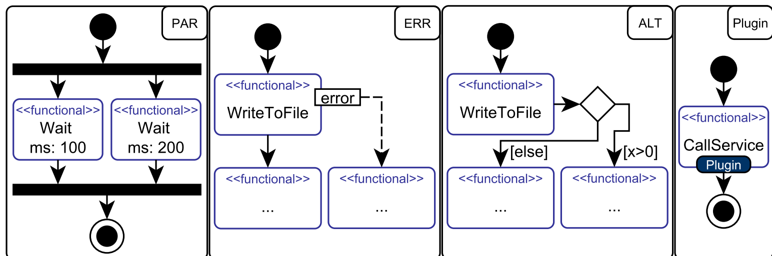

Abbildung 7.12. Die unterschiedlichen Elemente der DMDE mit einer grafischen Repräsentation durch an die UML angelehnte Aktivitätsdiagramme. Konkret werden vier Fälle vorgestellt: Die parallele Ausführung, Fehlerbehandlungen, Alternierungen und Erweiterungen über Plugins. (Eigenwerk)

7.2.3 Die Block Definition Language und das DMDE-Framework

Die *Block Definition Language* (BDL) ist in Python realisiert und dient als Grundlage für domänenspezifische Prozesse. Die BDL ist dabei unabhängig von ROS und Semantic Plug & Play und kann über Plugins sowohl für die Definition von Missionen als auch bspw. für Steuerprozesse eines Kaffeeautomaten verwendet werden [268]. Dies bildet auch die bedeutsamste Abgrenzung zu den verwandten Arbeiten, die sich allesamt nicht mit der Kombination aus ROS [167] und dem Themengebiet der Steuerung [238] gleichzeitig befassen. Eine konkrete Realisierung der BDL findet sich in dem *Dynamic Mission Definition and Execution* (DMDE)-Framework, in dem alle Komponenten der BDL realisiert und Plugins für ROS und für das Multiagentensystem Multipotenter Systeme aus Abschnitt 3.2 implementiert sind. Für die Definition einzelner Prozesse oder Missionen dient das Austauschformat JSON [134], das ebenfalls für Blaupausen und deren Instanziierung in der MDDM-Architektur verwendet wird. Für eine Übersicht der Prozessmodellierung anhand der DMDE werden folgend vier Fälle betrachtet: Die parallele Ausführung, Fehlerbehandlungen, Alternierungen und Erweiterungen durch Plugins. Eine exemplarische Übersicht der Fälle ist in Abbildung 7.12 skizziert, die im Folgenden in DMDE modelliert werden.

Parallele Ausführung

Analog zur Tokensemantik von UML-Aktivitätsdiagrammen werden Aktionen in BDL als Blöcke betrachtet. Assoziationen werden instantan ausgeführt, während Blöcke Zeit konsumieren und erst nach Beendigung eine ausgehende Assoziation auslösen. Für die parallele Ausführung existieren spezielle Blöcke wie der *fork* und *join* für die Parallelisierung und die Zusammenführung paralleler Programmflüsse.

Listing 7.1. Notation für den Block *PAR* aus Abbildung 7.12 in DMDE

```
{"mission": {
  "s": {            // Startblock
    "type":"Start",   // Typ des Blocks
    "next":"par" },   // Nachfolgeblock
  "par": {        // Forkblock
    "type":"Fork",
    "next":["w1","w2"] },  // Liste der Nachfolger
  "w1": {           // linker Wait
    "type":"Wait",
    "ms":"100",     // Parameter in ms
    "next":"join" },
  "w2": {           // rechter Wait
    "type":"Wait",
    "ms": "200",    // Parameter in ms
    "next":"join" },
  "join": {         // Joinblock
    "type":"Join",
    "next":"b5" },
  "b5 ": {        // Endblock
    "type":"End" } } }
```

In Anlehnung an das *PAR*-Element aus Abbildung 7.12 beginnt der Kontrollfluss in Listing 7.1 mit einem Kontrollknoten, der als Startblock in BDL definiert ist. Startblöcke erzeugen bei einem Start der Mission je einen eigenen Kontrollfluss und können mehrfach vorhanden sein. So könnten bspw. auch zwei Startknoten für die Parallelisierung verwendet werden. Blöcke besitzen einen Typ, der die Funktionalität definiert, und ein *next* Attribut, in dem Nachfolgeblöcke angegeben werden können. So besitzt der Forkblock *par* in Zeile 7 zwei Nachfolger, die parallel ausgeführt werden. Die Blöcke *w1 und w2* unterbrechen den Kontrollfluss für eine parametrisierbare Wartezeit in *„ms“* Millisekunden. Für die Synchronisierung beider Blöcke wird ein Joinblock verwendet, der auf alle eingehenden Kanten wartet. Da lediglich ein Verweis auf Folgeknoten gegeben ist und nicht auf die Vorgänger, wird vor der Ausführung einer DMDE-Mission der gerichtete Graph aller Blöcke einmal durchlaufen, um eingehende Kanten für Joins zu identifizieren. Eine Sammlung anliegender Kontrollflusstoken, wie in der UML spezifiziert, ist hingegen nicht realisiert.

Fehlerbehandlung

Das Aktivitätsdiagramm mit dem Titel *ERR* aus Abbildung 7.12 zeigt eine von der UML-Spezifikation abweichende Modellierung, da der Fehlerfall *error* als rechteckiger Kasten an dem Block *WriteToFile* angebracht ist. Da ein Fehlerfall innerhalb der Aktion auftritt und diese nicht beendet wird, ist eine alternative Modellierung durch einen Entscheidungsknoten nach der Aktion semantisch nicht korrekt. Eine zusätzliche ausgehende Kante ist hingegen ein impliziter Fork, der auch durch eine Bedingung nicht alterniert wird. Ein Unterbrechungsbereich könnte die Ausführung stoppen, der Fehler müsste

allerdings durch ein Signal modelliert werden [216]. Um diesen Sachverhalt in einfacher Form zu illustrieren, wird die Syntax entsprechend der Abbildung 7.12 erweitert.

Listing 7.2. ERR Element aus Abbildung 7.12 in DMDE

```
"wtf":{
  "type":"WriteToFile"
  "file":"~/hello.txt"  // Dateipfad und Name
  "text":"HelloWorld"  // Inhalt für die Speicherung
  "next":"b2"     // Nachfolgeblock im Erfolgsfall
  "nextError":"b4" }  // Nachfolgeblock im Fehlerfall
```

Der Fehlerfall ist in einem *WriteToFile* Block modelliert, der über mehrere Parameter eine Datei mit definiertem Inhalt anlegt und persistiert. Die interne Alternierung im Fehlerfall wird durch die Angabe entsprechender Blöcke in *next* und *nextError* angegeben.

Listing 7.3. Python-Implementierung des ERR Elements aus Abbildung 7.12 in DMDE

```
def block_write_to_file(args):
  try:
    f = open(args["file"],"w+") # Datei öffnen
    f.write(args["text"])   # Text in Datei schreiben
    f.close()         # Datei schließen
    return args["next"]   # Im Erfolgsfall -> next
  except Exception:
    return args["nextError"]  # Im Fehlerfall -> nextError
```

Der in Listing 7.3 abgebildete Pythoncode illustriert die einfache Erweiterbarkeit in der BDL. Die Steuerung des Kontrollflusses findet in der dazugehörenden API statt.

Alternierung

Alternierungen verhalten sich analog zu der internen Alternierung eines Fehlerfalles. Durch die Vorgabe der deterministischen Modellierung ist allerdings der *else* Fall zwingend erforderlich. Zwar steht nur ein Alternierungsblock mit der Bedingung als Parameter zur Verfügung, Mehrfach-Bedingungen wie in einer *Switch*-Anweisung müssen über diese gestaffelt oder als eigener Block ergänzt werden.

Plugins

Durch die Definition einzelner Methoden in Python können beliebige Erweiterungen vorgenommen werden. In der DMDE gibt es bereits zahlreiche Blöcke für die Verwendung von **ROS 1 und 2**, wie etwa den Callservice aus Abbildung 7.12, der analog zu den anderen Blöcken über Parameter in die Struktur eingefügt werden kann.

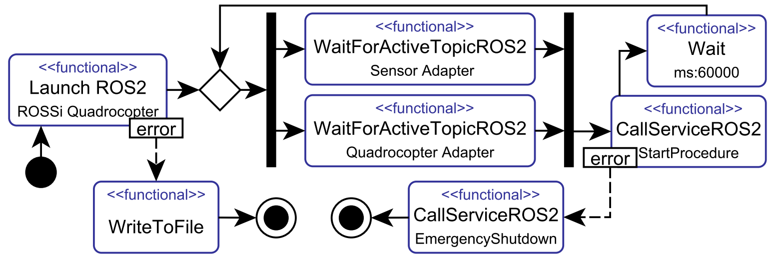

Abbildung 7.13. Auszug aus der Modellierung eines Sensortausches für das Experiment des sensorbasierten Fluges. Die Launchfile für die Verteilung und der Flug der Drohne ist mit ROSSi modelliert, der Konfigurationsprozess hingegen ist in DMDE verankert. (Eigenwerk)

Listing 7.4. Plugin-Element aus Abbildung 7.12 in DMDE

```
1  "mission": {
2    "b1 ":{
3      "type": "Callservice",
4      "name": "/example/service",  // Servicename
5      "type": " std_srvs/SetBool",  // Typ des Services
6      "args": "{data:true}",    // Parameter für Service
7      "next ": "b2" } }
```

Dieser Block verwendet eine Reihe von Shell-Skripten, welche die cli-tools (Command Line Interface) von ROS 1 und ROS 2 ausführen. Folgende Blöcke sind bspw. für das Experiment des sensorbasierten Fluges implementiert: *CallServiceROS1, CallServiceROS2, StartRoscore, WaitForActiveTopicROS1, WaitForActiveTopicROS2, StartNodeROS2, LaunchROS1, LaunchROS2, StartROS1Bridge, StartLogging* und *StopLogging*. In ROSSi können in den einzelnen Nodes auch Prozesse definiert und Semantic Plug & Play-Adapter integriert werden. DMDE bietet eine Alternative zur Prozessdefinition und ist durch das ROS-Plugin mächtig genug, auch eigene Launchfiles zu bilden. Die externe Ausführung von DMDE bietet neben der terminologischen Verwaltung, wie der Synchronisation des Initialisierungsprozesses in Abbildung 7.13 vor dem Prozessstart *StartProcedure*, auch eine externe Fehlerbehandlung sowie die Schleifenbildung für eine Rekonfiguration des Systems innerhalb einer definierten Zeit.

Die *Multiagentensystemerweiterung* nutzt diverse ROS 2-Nodes sowie das ROS 2-Nachrichtensystem, um Barrieren zur Synchronisierung der verwendeten Systeme zu integrieren. Über unterschiedliche Blöcke kann ein Server-Client System aufgebaut werden, indem sich alle Clients beim Server registrieren. Dadurch wird ein *Barriereblock* ermöglicht, der in einer DMDE-Mission platziert werden kann, um den Schwarm wie in Abbildung 7.14 modelliert zu synchronisieren. Dieser Anwendungsfall ist in dem Experiment für die synchrone Glasfasermessung aus Abschnitt 3.3.2 gegeben. Das Multiagentensystem aus Abschnitt 3.2 ist in JAVA geschrieben und kann über die OOD-API ebenfalls auf ROS zugreifen, indem virtuelle Fähigkeiten in Knoten injiziert werden. So kann auch die Infrastruktur des Multiagentensystems genutzt werden, um den in Abbil-

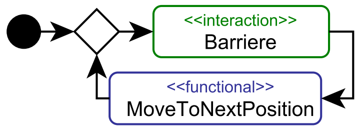

Abbildung 7.14. Einfache Modellierung einer Mission, die eine in einzelne Positionen unterteilte Trajektorie über Barrieren zwischen verschiedenen Elementen synchronisiert. (Eigenwerk)

dung 7.14 illustrierten Ablauf zu realisieren. BDL hingegen ist speziell für die einfache Integration von Prozessen aus unterschiedlichen Domänen vorgesehen und bietet ebenfalls automatische Verteilungsmechanismen für DMDE-Missionen über GIT. Ähnlich zu der unscharfen Trennung zwischen Prozessen und virtuellen Fähigkeiten können auch hier Überschneidungen stattfinden, die domänenspezifisch ausgelegt werden.

Eine ähnliche Überschneidung offenbart sich bei der Einordnung in Semantic Plug & Play. Durch die Implementierung der Commands Protocol-Schnittstelle in Python können virtuelle Fähigkeiten als Block in BDL integriert und durch DMDE gesteuert werden. Damit gehen allerdings die Vorteile der Anforderungsspezifikation sowie die Mechanismen für aufeinander aufbauende Fähigkeiten der OOD-API verloren. Somit bedient BDL vielmehr den Rekonfigurationsaspekt innerhalb des ROS-Universums, der wie in Abbildung 7.13 unabhängig von der OOD-API modelliert werden kann. Die Ausführung der als JSON hinterlegten DMDE-Mission kann analog zu den Launchfiles in ROSSi ebenfalls in einen Semantic Adapter gekapselt und über Docker bereitgestellt werden. So kann das Multiagentensystem die OOD-Schnittstelle verwenden, Anforderungen definieren und die virtuellen Fähigkeiten samt Selbstbeschreibung der externen Semantic Adapter verwenden. Zur Laufzeit wird das in ROSSi modellierte System in Kombination mit der DMDE-Mission aus Abbildung 7.13 als Dockercontainer heruntergeladen und gestartet. Die Node des injizierten Sensor Adapters im *WaitForActiveTopicROS2* der Abbildung 7.13 wartet auf Sensordaten des adaptieren Sensors. Falls in der definierten Wartezeit von einer Minute der Sensor nicht durch einen Anwender demontiert wurde, wird ein neuer Sensorwert gesendet und der in ROS definierte Prozess startet erneut. Ist jedoch der Sensor binnen der Minute demontiert, so wird gewartet, bis ein neuer Sensor angeschlossen wird. Durch die virtuelle Fähigkeit ist lediglich ein Sensor des Typs *Distanz* relevant und der Ultraschallsensor kann z.B. durch den Lasersensor ersetzt werden. Somit ist die statische Struktur in ROSSi gegeben, die Rekonfiguration in DMDE ausgedrückt und die Abstraktion in virtuellen Fähigkeiten möglich. Die Verteilung und das semantisch annotierte Logging der Sensordaten ist in der OOD-API verankert.

Die Block Definition Language (BDL) und das Dynamic Mission Definition and Execution (DMDE)-Framework sind publiziert [268] und ebenfalls Open-Source[5] unter der MIT Lizenz verfügbar. Besonderer Dank gilt meinem Kollegen Martin Schörner, der in seiner Abschlussarbeit, die im Rahmen der Dissertation betreut wurde, einen Anteil zur Realisierung des DMDE-Frameworks beigetragen hat.

[5] https://github.com/isse-augsburg/BlockDefinitionLanguage

In den folgenden Kapiteln dient ROS ebenfalls als Grundlage für zentrale Simulationen und verteilte Visualisierungen. Semantic Plug & Play bedient dabei wiederum die Verteilung und Sammlung der 3D-Objekte als Teil der Selbstbeschreibung.

7.3 Modulare Simulation

Die Begriffe der Simulationen und der Visualisierungen werden häufig synonym verwendet, was unter anderem durch große Gameengines, wie der Unrealengine [294] oder Unity [114], begründet wird. Für die Spielentwicklung können in beiden Engines durch grafische Editoren die Umgebungen gebaut und physikalische Kräfte parametrisiert werden. In der Domäne der Robotik hat sich hingegen Gazebo [146] in Kombination mit ROS etabliert, dennoch verwenden Projekte mit illustrativen Umgebungen vermehrt Gameengines, wie bspw. das Projekt USARSim [52], das auf der Unrealengine aufbaut. Das liegt vor allem an den unterschiedlichen Zielen der Domänen. Gameengines zeichnen sich primär durch eine gute *Rendering Engine* für die Visualisierung, vereinfacht ausgedrückt der visuellen Wahrnehmung [7] aus. Dafür werden bspw. *Tesselation Shader* eingesetzt, die auf speziellen ALUs der Grafikkarten aufbauen [282]. Simulationen im Bereich der Robotik fokussieren hingegen korrekte physikalische Effekte, wie etwa das Flugverhalten eines Quadrocopters. Dafür existieren Werkzeuge wie das Gazebo Plugin *RotorS* der ETH Zürich [91].

Die Idee der modularen Simulation ist die Trennung der Visualisierung samt Gameengine von einer zentralen Simulation. Die *Unity*-Gameengine dient als Instrument der Visualisierung auf unterschiedlichen Plattformen, während Gazebo in Kombination mit *RotorS* die Simulationskomponenten bedient. Durch die Trennung soll ebenfalls die 1:1-Multiplizität zwischen Simulation und Visualisierung aufgehoben werden, damit mehrere Visualisierungsinstanzen erzeugbar sind, wie in Abbildung 7.15 illustriert. Die einzelnen 3D-Objekte werden als Teil der Selbstbeschreibung aufgenommen und müssen im Semantic Adapter gesammelt und der Simulation bereitgestellt werden.

Als Einführung wird das Experiment GoLive aus Kapitel Kapitel 3 erweitert und die konkrete Umsetzung aufgezeigt. Anschließend werden die Selbstbeschreibungsmechanismen für 3D-Objekte vorgestellt, die über die modulare Simulation an die verteilte MR-Virtualisierung verteilt werden.

7.3.1 Experiment: GoLive

Das Experiment GoLive dient der schrittweisen Realisierung des Katastrophenszenarios der Fallstudie aus Kapitel 3. Die Ausgangsbasis ist eine klassische Simulation mit Visualisierung, in der die Simulation der konfigurierbaren Quadrocopter, Gassensoren und eine Gaswolke verwendet werden, um die erste Stufe, das Finden der Gasquelle, simuliert wird. Der Schritt in die Realität ist nicht trivial, was auch dadurch erkennbar wird, dass alleine die Messung von Gasen auf einem Quadrocopter in zahlreichen Publikationen thematisiert ist [223, 253]. So beeinflussen bspw. Faktoren wie der Luftstrom der Rotoren die Messungen des Sensors oder die Luftfeuchte vor Ort wirkt sich auf optische Gassensoren aus [34], bzw. chemische Sensoren müssen kalibriert werden, da durch die Reaktion mit Gasen Abnutzungserscheinungen auftreten [120]. In der simulierten

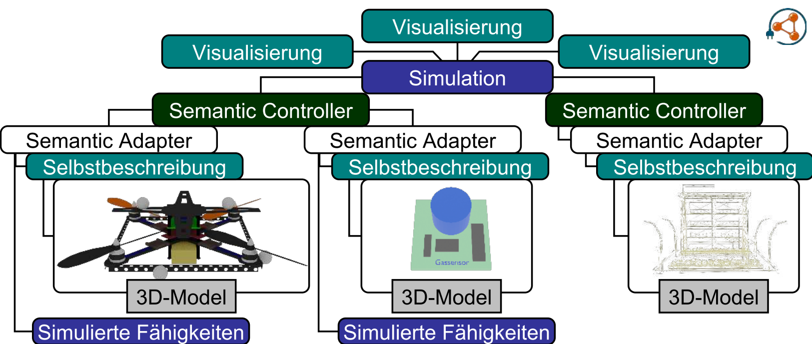

Abbildung 7.15. Struktureller Aufbau der Simulation als zentrales Element mit unabhängiger Visualisierung, sowie dezentrale Beschreibung aller 3D-Modelle über die Selbstbeschreibung. (Eigenwerk)

Welt wird stattdessen häufig nur ein Rauschen für Sensorungenauigkeiten definiert und praktischerweise ist auch die Batterie des Quadrocopters immer voll geladen. Bei realen Einsätzen der Feuerwehr sind jedoch nicht nur die allgemeinen Funktionen realer Hardware relevant, sondern unterschiedliche Umgebungen resultieren auch in Herausforderungen für das eingesetzte Equipment. Damit die Simulation und der Entwicklungsprozess bis zum Einsatz auf realer Hardware entsprechend den Ansprüchen aus dem illustrierten Katastrophenszenario gerecht wird, werden im Experiment *GoLive* mehrere Aspekte beleuchtet:

Hardwareverteilung
MR-Bausteine sind Einplatinencomputer, die in Kombination mit der Hololens ein 3D-Modell projizieren. Durch die Verteilung der 3D-Modelle, wie in Abbildung 7.15 illustriert, können die Verteilungsmechanismen von Semantic Plug & Play in diesem Experiment getestet werden.

Interaktive Umgebung
MR-Bausteine dienen auch als Tangible Userinterfaces, wodurch das Szenario beliebig verändert werden kann, wie in Abbildung 7.16 durch die *MR-Bausteine - Infrastruktur* illustriert.

Steuerung des Digitalen Zwillings
Die Steuerung des digitalen Zwillings findet über simulierte Fähigkeiten statt, die eine zu realen Fähigkeiten identische Schnittstelldefinition bietet. So sollen im Experiment sowohl reale als auch simulierte Quadrocopter austauschbar gesteuert werden können.

Visualisierung der Simulation
Die Mixed Reality-Visualisierung wird von der Simulation entkoppelt, damit die Vorteile der Unity-Gameengine verwendet werden können. MR-Bausteine in Kombination mit der tragbaren Hololens bieten ein direktes Feedback für die Interaktion mit der Simulation.

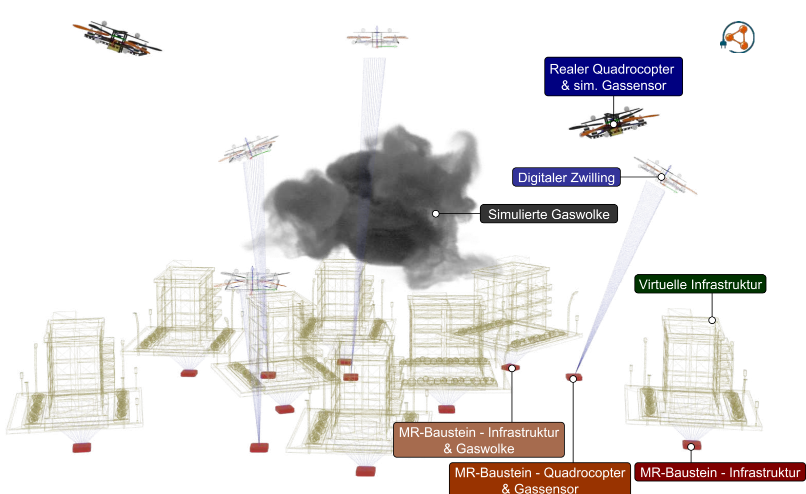

Abbildung 7.16. Struktureller Aufbau der Simulation als zentrales Element mit unabhängigen Visualisierungen, sowie dezentrale Beschreibung aller 3D-Modelle über die Selbstbeschreibung. (Eigenwerk)

Hybride Konfiguration
Reale Quadrocopter werden mit einem virtuellen Sensor ausgestattet, um sich schrittweise der Realität zu nähern. So können auch reale Quadrocopter die simulierte Gaswolke messen, bzw. können die Kollisionserkennungs- und Vermeidungsstrategien getestet werden.

Evaluation von Schwärmen
Die Mischung aus realen und simulierten Quadrocoptern bietet die Möglichkeit Schwarmverhalten anhand der digitalen Zwillinge zu visualisieren und einzelne Aspekte durch den Einsatz echter Quadrocopter glaubhaft zu illustrieren.

Dieses leitende Beispiel führt durch die Implementierung der modularen Simulation, mit der auch alle Szenarien der Fallstudie realisierbar sind. Beginnend bei der Verteilung der digitalen Zwillinge durch die Semantic Adapter und Controller über den Simulationskern in ROS bis hin zur Verteilten MR-Virtualisierung werden die Möglichkeiten von Semantic Plug & Play in Kombination mit den Werkzeugen ROSSi und BDL in der modularen Simulation demonstriert.

7.3.2 Selbstbeschreibung mittels digitalen Zwillings

Die Selbstbeschreibung des digitalen Zwillings wird in zwei Kategorien unterteilt. Einerseits die **simulierten Fähigkeiten**, die für die Ausführung innerhalb der Simulation benötigt werden, und andererseits **3D-Modelle** für die Visualisierung.

Die Simulation basiert auf Gazebo, der De-Facto-Standard in ROS. **Simulierte Fähigkeiten** in ROS sind nachgebildete Nodes, die eine Schnittstelle zur Simulation bieten,

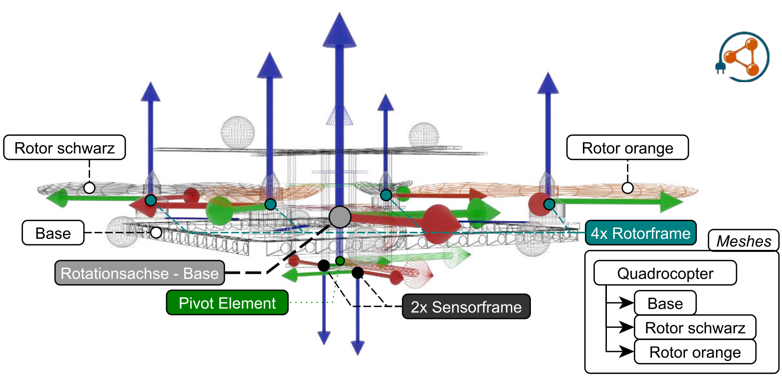

Abbildung 7.17. 3D-Modell eines Quadrocopters, das sich aus mehreren Meshes zusammensetzt. Die *Base* bildet das statische Grundmodell, die jeweils 2 schwarzen und orangen Rotoren rotieren um die nach oben gerichtete Z-Achse. (Eigenwerk)

wie etwa in RotorS für Quadrocopter. Die Beispielmodelle in RotorS hingegen müssen passend für die Näherung an die physischen Quadrocopter parametrisiert werden, die zu einem späteren Zeitpunkt eingesetzt werden sollen. Auch die Schnittstellen können sich unterscheiden. So sind Adapter in ROS stellenweise notwendig, damit die Schnittstelle für die Ansteuerung zwischen echtem und simuliertem Quadrocopter identisch bleibt. Native Semantic Plug & Play-Fähigkeiten können ebenfalls in den Nodes implementiert werden, allerdings verschiebt sich dadurch die Prozessdefinition zwangsweise in ROS.

Für die Beschreibung eines **3D-Modells** sind einerseits *Meshes* und andererseits Beschreibungen der *Abhängigkeiten* zwischen den Meshes nötig. Die Bereitstellung der Meshes findet über die RDF-Graphen mit einer Referenz auf ein GIT-Repository statt. Das Dateiformat für Meshes ist in der modularen Simulation analog zu AutomationML [69] auf Collada ausgelegt. Collada bietet die Möglichkeit, mehrere Meshes mit Abhängigkeiten zueinander zu kapseln [12]. Meshes verwenden üblicherweise mehrere Koordinatensysteme (Frames), die in der Rotation und Translation eingeschränkt sind. Das in Abbildung 7.17 abgebildete 3D-Modell setzt sich aus einem *Base Mesh* und jeweils zwei Meshes für die schwarzen und orange-farbenen Rotoren zusammen. Jedes Mesh hat ein Pivot Element, das exemplarisch für das *Base Mesh* eingezeichnet ist. Dieser Frame ist der Ankerpunkt, an dem das Mesh in die Visualisierung eingehängt wird. Üblicherweise überschneidet sich das Pivot-Element mit der Rotationsachse, um die sich der Quadrocopter drehen kann. Da der Quadrocopter keine Einschränkung in der Rotation besitzt und durch die omnidirektionale Beweglichkeit auch translatorisch nicht beschränkt ist, müssen für den Frame keine Bedingungen festgelegt werden. Die Rotorframes und Sensorframes bilden die Ankerpunkte für weitere Meshes. An dem Rotorframe sind bereits die vier Rotormeshes dargestellt, deren Pivot-Elemente und Rotationsachsen sich überschneiden. Translatorisch besteht die Abhängigkeit zur Rotationsachse. Eigenständige translatorische Bewegungen sowie rotatorische Bewegungen der x- und y-Achse sind hingegen nicht möglich. Diese Art der Abhängigkeiten zwischen Frames

und Meshes kann in Collada durch die ROS-spezifische Erweiterung, das *Universal Robotic Description Format* (URDF), ausgedrückt werden [60]. Collada und URDF bauen auf dem XML auf und können in einer Datei verwendet werden.

```
1  <joint name="rotorSchwarz_1" type="continuous">
2    <origin rpy="0 0 0" xyz="1.0 1.0 0.2"/>
3    <axis xyz="0 0 1"/>
4    <parent link="Rotationsachse"/>
5    <child link="rotorSchwarz">
6  </joint>
```

In diesem Ausschnitt einer URDF-Beschreibung wird die Abhängigkeit (joint) zwischen einem Rotor Frame (rotorSchwarz_1) und der Rotationsachse des Quadrocopters modelliert. Das Attribut *origin* definiert die Transformation zwischen beiden Frames, also den Roll-Nick-Gier-Winkel, sowie die Translation durch X,Y,Z. Im Beispiel wird also der Rotor ausgehend von der Rotationsachse der relativen Basis verschoben und nicht gedreht. *Axis* definiert die rotatorischen Einschränkungen, im Beispiel die Rotation um die z-Achse. Die referenzierten Meshes werden ebenfalls in XML beschrieben, zum Beispiel besteht der in Abbildung 7.17 abgebildete Quadrocopter aus insgesamt 188.938 Dreiecken (Triangles) inkl. der Rotoren.

Durch die Nutzung der Infrastruktur von Semantic Plug & Play kann die Collada-Datei in einem GIT-Repository bereitgestellt und über den RDF-Graphen referenziert werden. Als Basiskonstrukt dient das *Device*, in dem auch die Referenzen der simulierten Fähigkeiten assoziiert sind. Da die Simulation ROS als Basis verwendet wird, können zu den simulierten Fähigkeiten über ROSSi verwaltet und implementiert werden, insbesondere bei komplexen Prozessen kann zudem BDL verwendet werden. Semantic Plug & Play bedient die Verteilung sowie die Selbstbeschreibungsmechanismen für die zentrale Simulation, wie in Abbildung 7.15 illustriert.

7.3.3 Simulation

Als Simulationsplattform dient Gazebo in Kombination mit RotorS, die über die Middleware ROS angesteuert werden können. Sie steht als zentrales Element über den Semantic Controllern und benötigt zur Simulation und Visualisierung aus der Selbstbeschreibung aller Semantic Adapter die Collada Dateien der Meshes, URDF Beschreibungen und die simulierten Fähigkeiten. Gemäß der Struktur von ROS werden die virtuellen Fähigkeiten als Nodes gekapselt, wofür ROSSi eine einfach Migration bereitstellt. Da sowohl bei Gazebo als auch in Gameengines wie Unreal und Unity die 3D-Modelle oft bereits zur Designzeit angelegt werden müssen, jedoch zur Laufzeit ein- und ausgeblendet werden können, müssen die 3D-Modelle vor der Ausführung von den Semantic Controllern heruntergeladen werden. Damit verliert die Simulation den Vorteil der Erweiterung systemfremder Objekte zur Laufzeit. Diesem Problem widmet sich mein Kollege Michael Filipenko aus derselben Forschergruppe mit einem Konzept eines Mixed Reality Browsers.

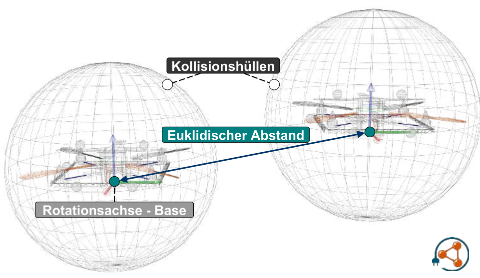

Abbildung 7.18. Zwei simulierte Quadrocopter mit jeweils einer Sphere als Kollisionshülle. Der euklidische Abstand wird anhand der jeweiligen Rotationsachsen (Base) gemessen. (Eigenwerk)

Der erste Schritt für Abhängigkeiten zwischen Objekten der Simulation und der Realität innerhalb der Simulation bietet die Kollisionserkennung und -vermeidung. In URDF können vereinfachte Kollisionshüllen definiert werden, damit die Kollisionserkennung keine komplexen Strukturen berechnen muss, wie etwa für den Quadrocopter aus Abbildung 7.17 mit knapp 189 tausend Triangles. Eine einfache Kollisionshülle, oder auch Bounding Box genannt wird bspw. durch eine Kugel (Sphere) ausgedrückt, die an der Rotationsachse der Base angehängt wird.

```
1  <collision>
2    <geometry>
3      <sphere radius="3.0"/>
4    </geometry>
5  </collision>
```

Die Kugel bietet im Fall der komplexen Geometrie eines Quadrocopters (siehe Abbildung 7.18) keine so gute Bereichseingrenzung wie etwa ein Quader, der an der Achse (Axis Aligned Bounding Box - AABB) oder der Orientierung (Oriented Bounding Box - OBB) des Objektes ausgerichtet ist [308]. Die Berechnung für die Kollision kann jedoch effizient durch die euklidische Distanz gleicher Kollisionshüllen berechnet werden, was in Anbetracht der Simulation von Katastrophenszenarien mit Quadrocopterschwärmen deutliche Performance-Vorteile bringt.

Für die Kollisionsvermeidungsstrategie wird in der Simulation die in ROS verfügbare Potentialfeldmethode angewendet, die über anziehende und abstoßende Kräfte sowohl Ziel als auch Hindernisse definiert. Die vereinfachte Hülle bietet in dieser Methode den Vorteil einer nahezu trivialen Definition der abstoßenden Kräfte über die Distanzfunktion. Der Vorteil einer Potentialfeldmethode ist die Ausführung ohne vorhergehende Planung und die Möglichkeit, auf dynamische Hindernisse zu reagieren [259]. Nachteile bilden nicht optimale Routen und zusätzliche Algorithmen für lokale Minima [327]. Die Simulation ist das zentrale Element, das alle Kollisionshüllen der 3D-Modelle verwaltet und damit Kollisionserkennungs- und Vermeidungsstrategien ermöglicht. Für die Realisierung in den Katastrophenszenarien wird die Potentialfeldmethode hingegen mit einer verteilten Wissensbasis auf Agentenebene angewendet [149, 153].

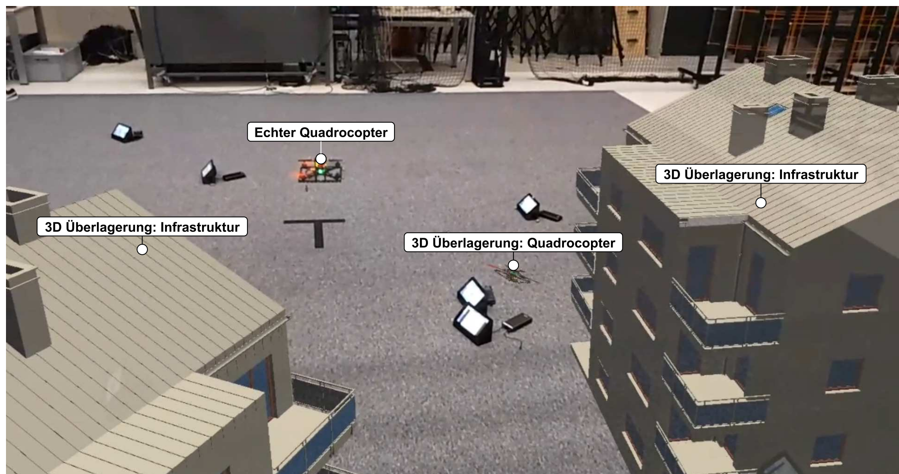

Abbildung 7.19. Bild einer Überlagerung der Realität mit einer Microsoft Hololens und dem digitalen Zwilling des Quadrocopters und zwei Häusern (Infrastruktur). (Eigenwerk)

Abhängigkeiten zur Umgebung wie etwa eine Kollision eines Quadrocopters mit der Infrastruktur werden ebenfalls durch Kollisionshüllen gehandhabt. Neben der Möglichkeit, die Umgebung mittels der MR-Bausteine umzustellen, sind auch Objekte innerhalb der Simulation ohne Anbindung an MR-Bausteine über das Softwareartefakt des Semantic Adapters vorgesehen. Die Nutzerschnittstelle für die Positionierung der Objekte bildet die ROS-Komponente des *TF-Graphen*. Innerhalb des Pakets werden alle geometrischen Abhängigkeiten, wie in Abbildung 7.17 dargestellt, festgehalten. Die Struktur und Beschreibung des TF-Graphen dient in der modularen Simulation auch als gemeinsame Basis für die Spiegelung in externe Visualisierungen, wie etwa der MR-Virtualisierung.

7.3.4 Verteilte MR-Virtualisierung und MR-Bausteine

Die Zielplattform für die *MR-Virtualisierung* ist die Microsoft Hololens, die Hologramme der digitalen Zwillinge in die Realität überlagert, wie in Abbildung 7.19 dargestellt. Die Hololens der ersten Generation verwendet eine Kombination aus Tiefenbildkamera, RGB-Kameras, sowie eine Inertiale Messeinheit (IMU) für die Odometrie [137]. Als Betriebssystem wird Windows 10 eingesetzt, das mit der nativen Integration von ROS nicht kompatibel ist. Für den Abgleich des TF-Graphen wird daher Ros# verwendet, eine Open-Source-Sammlung an Bibliotheken für die Kommunikation zwischen ROS und .NET-Anwendungen. *.NET* ist ein Framework von Microsoft für die Programmiersprache C#. Die *Unity*-Gameengine verwendet das .NET-Framework und bietet gleichzeitig eine Unterstützung für MR-Anwendungen speziell für die Hololens. In der Unity-Anwendung der MR-Visualisierung wird der TF-Graph zur Laufzeit über eine ROS-Node übertragen, die 3D-Objekte werden hingegen initial ausgetauscht. Wenn ein 3D-Objekt zur Laufzeit in der Simulation aus- oder eingeblendet wird, um die Rekonfiguration zu simulieren, so wird der TF-Graph aktualisiert und analog in der MR Visualisierung angepasst.

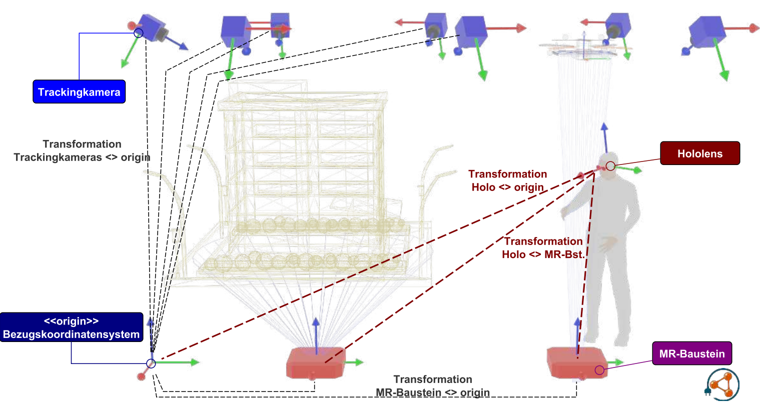

Abbildung 7.20. Struktureller Aufbau der Überlagerung aus Abbildung 7.19. Ein Bezugskoordinatensystem dient als gemeinsame Basis für die Transformationen zwischen den MR-Bausteinen, der Hololens und dem Trackingsystem. (Eigenwerk)

Die verwendeten Koordinatensysteme und Bezugspunkte für die Visualisierung stellen ein Problem für die MR-Visualisierung dar. Die Hololens verwendet ein eigenes Koordinatensystem, das in etwa im Zentrum der Brille definiert ist. Das Trackingsystem für die Quadrocopter verwendet ebenfalls ein eigenes Ursprungs-Koordinatensystem, das manuell festgelegt wird. Zuletzt bleibt noch die Simulation, das selbst wiederum ein eigenes Koordinatensystem definiert. Für die korrekte Darstellung aller Elemente wird ein Bezugskoordinatensystem, wie in Abbildung 7.20 dargestellt, definiert. Die Transformation zwischen den Trackingkameras und der Simulation ist innerhalb von ROS unproblematisch, da die Transformation innerhalb der Anwendung gesetzt werden kann und ROS bereits eine Integration dafür bereitstellt. Für die Transformation zwischen den MR-Bausteinen und der Hololens wird ein besonderes Verfahren, das Markertracking der Software Vuforia [323] verwendet. Anders als bei April-Tags [215] unterstützt Vuforia beliebige Bilder als dreidimensionale Tags. So können auch kodierte Formate als Basis dienen, wie etwa QR-Codes, die eine Referenz auf die URI eines Individuals der Selbstbeschreibung geben. Die Individuals haben wiederum Referenzen auf die 3D-Modelle und bilden quasi den Einstiegsknoten des RDF-Graphen. So können zwei Arten von MR-Bausteinen gestaltet werden:

MR-Bausteine für statische Objekte
Für statische Objekte, wie etwa Gegenstände der Infrastruktur, können die QR-Codes ausgedruckt und frei im Raum positioniert werden.

MR-Bausteine für steuerbare Objekte
Alternativ können auch Einplatinencomputer mit Bildschirmen oder Tablets als MR-Bausteine verwendet werden. Die Generierung der QR-Codes erfolgt anhand der Selbstbeschreibung automatisch. Die virtuellen Fähigkeiten für die Steuerung der digitalen Zwillinge werden über den Einplatinencomputer oder das Tablet bereitgestellt.

Abbildung 7.21. QR-Code, der als Vuforia-Tag verwendet werden kann und eine Referenz auf die Selbstbeschreibung des MR-Bausteins kodiert. In diesem Beispiel ist die Übersichtsseite von Semantic Plug & Play kodiert. (Eigenwerk)

Mit beiden Varianten kann die Transformation zwischen der Hololens und den MR-Bausteinen hergestellt werden. Für eine rein simulative Anwendung kann die Transformation zwischen der Hololens und der Simulation auf die Basis der Hololens gelegt werden, da die Visualisierung auf das Frustum des Nutzers gerichtet ist. Für das Bezugskoordinatensystem hingegen muss mindestens jeweils ein Frame sowohl im Trackingsystem als auch auf der Hololens als Bezugspunkt erkannt werden. Somit muss mindestens ein MR-Baustein für die Erkennung über Marker des Trackingsystems ausgestattet werden. Dieser muss initial von der Hololens und dem Trackingsystem einmalig erkannt werden, unterliegt aber der Problematik des Odometry Shifts der Hololens zur Laufzeit. Das bedeutet, dass nach einer gewissen Zeit die Ungenauigkeiten der internen Sensoren der Hololens den initialen Bezugspunkt zunehmend verschieben und die Positionen der realen Quadrocopter nicht mehr mit den Positionen der visualisierten Umgebung übereinstimmen. Der Odometry Shift wird hingegen korrigiert, wenn der MR-Baustein mit Markern von beiden Systemen zur Laufzeit registriert wird.

Die modulare Simulation bietet eine entkoppelte Visualisierung für MR-Anwendungen. So können akkurate Simulationen in Gazebo und der Erweiterung RotorS durchgeführt werden und die Ergebnisse mithilfe einer Gameengine visualisiert. Die Mechanismen von Semantic Plug & Play bieten eine einfache Verteilung der 3D Modelle, sowie der zugehörigen Beschreibungen virtueller Fähigkeiten. Durch die Unterstützung der Mixed Reality und der MR-Bausteine kann zudem mit der Simulation zur Laufzeit interagiert werden, um die Umgebung analog zu den Katastrophenszenarien zu manipulieren. Statische MR-Bausteine ohne virtuelle Fähigkeiten profitieren von den Linked Open Data Prinzipien, indem eine URI als Einstiegspunkt für einen RDF-Graphen als QR-Code kodiert und verteilt werden kann. Für die schrittweise Überführung in die Realität wird ein externes Trackingsystem verwendet, mit einem Bezugskoordinatensystem als Grundlage der Interaktion.

In dem folgenden Kapitel werden die Werkzeuge in die Methodologie von Semantic Plug & Play eingeordnet und kurz zusammengefasst.

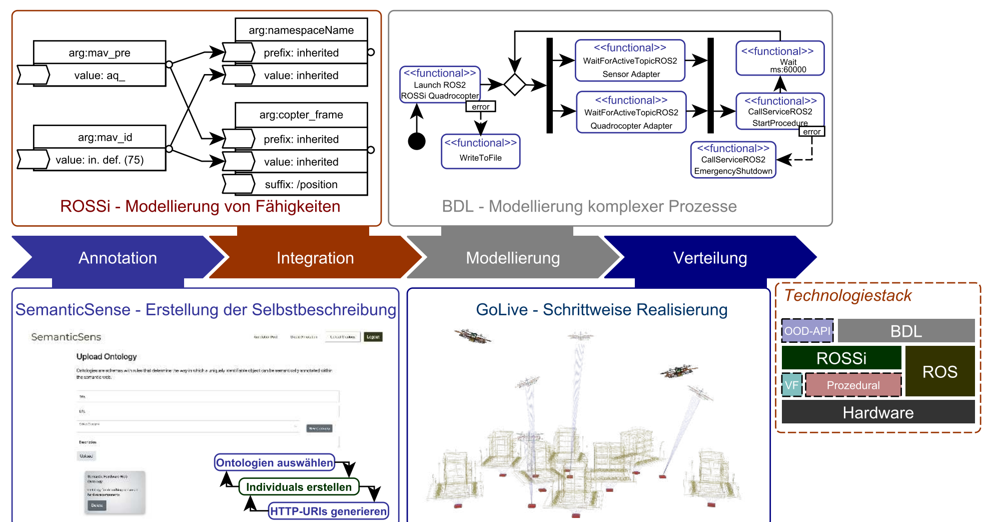

Abbildung 7.22. Einordnung der Werkzeuge in die Methodologie von Semantic Plug & Play. (Eigenwerk)

7.4 Einordnung der Werkzeuge in die Methodologie

Die Architektur von Semantic Plug & Play bietet bereits eine volle Integration der Methodologie, die in Abbildung 7.22 abgebildet ist. Die Werkzeuge aus diesem Kapitel vereinfachen hingegen die Realisierung einzelner Schritte der Methodologie und erweitern den Technologiestack mit Werkzeugen für die Integration des Robot Operating Systems (ROS).

Für die Verwaltung der Individuals bietet **SemanticSense** eine intuitive Weboberfläche, die durch eine REST Schnittstelle mit einem in Python implementierten Backend verbunden ist. Die scharfe Trennung ermöglicht einen Austausch des Frontends oder domänenspezifische Aufbereitungen für individuelle Lösungen. Der Import zusätzlicher Ontologien ermöglicht eine detaillierte Beschreibung der konkreten Hardware. Die einfache Aufbereitung der komplexen Semantic Web Mechanismen bietet den idealen Einstiegspunkt für Forscher und auch Sensorhersteller die konkrete Hardware maschinenlesbar zu beschreiben. Linked Open Data Prinzipien in Kombination mit einer automatisierten Bereitstellung ermöglichen nicht nur den Einsatz innerhalb von Semantic Plug & Play . Auch als Grundlage für die Integration von Rich Snippets oder für die Sortierung und Filterung in Online Shopsystemen kann die Annotation herangezogen werden.

Die Integration von Semantic Plug & Play in das ROS findet über das Modellierungswerkzeug **ROS Simple** (ROSSi) statt. Die einzelnen Nodes können im Node-Editor durch wiederverwendbare Bausteine erstellt und ein Codeskelett generiert werden. Neben den typischen Bausteinen für ROS, wie ein Publisher oder Subscriber kann auch die Implementierung des Commands Protocol für virtuelle Fähigkeiten verwendet werden. Damit lassen sich ROS-Nodes injizieren, die einen Adapter für Semantic Plug & Play bilden. Die

Parametrisierung und der Startvorgang einzelner Nodes wird durch ein Launchfile ausgedrückt, das ebenfalls über eine grafische Benutzerschnittstelle konfiguriert werden kann. Die zweite Integration mit Semantic Plug & Play bildet die Kapselung der Launchfiles, sowie der zugehörigen Nodes in einen Container, der über die Verteilungsmechaniken von GIT innerhalb eines Semantic Controllers auf die Hardware verteilt werden kann. Für die Überwachung des Programmablaufs steht eine interaktive Visualisierung des Node Graphen zur Verfügung, mit der Möglichkeit, die Kommunikation zwischen den Nodes zur Laufzeit einzusehen. ROSSi bietet nicht nur eine Integration für Semantic Plug & Play , sondern kann auch eigenständig für ROS-Applikationen verwendet werden. Die grafische Modellierung bietet eine Übersichtlichkeit in komplexen ROS-Systemen, die mit einer Codegenerierung und wiederverwendbaren Fragmenten leicht überführt und zur Laufzeit überwacht werden kann.

Die Kontrollflussstruktur der **Block Definition Language (BDL)** ermöglicht eine einfache Modellierung von domänenspezifischen Prozessen in JSON. Eine exemplarische Realisierung ist das Dynamic Mission Definition and Execution (DMDE)-Framework mit einer Unterstützung für ROS und Multiagentensystemen. Damit wird eine explizite Schnittstelle für die Prozessmodellierung geschaffen, die in ROS üblicherweise über Nodes verteilt ist. Die Mechaniken des DMDE-Frameworks verfolgen primär den Aspekt der Modellierung von Rekonfiguration zur Laufzeit in ROS. So bieten Kontrollknoten in Anlehnung der UML-Kontrollflusssemantik von Aktivitätsdiagrammen Möglichkeiten, einfache Parallelisierungen zu instanziieren und synchronisieren. Blöcke können domänenspezifisch erweitert und angepasst werden, um bspw. eine Kommunikation zwischen Schwarmelementen zu realisieren.

Die Kombination von Semantic Plug & Play und ROS ermöglicht letztlich eine **modulare Simulation** mit einer entkoppelten MR-Virtualisierung, die den Entwicklungsprozess unterstützt und den Schritt zur Realität unterteilt. Die zentrale Simulation sammelt die Informationen für die Visualisierung durch die Selbstbeschreibungsmechanismen aller verfügbaren MR-Bausteine. Ein MR-Baustein kann über simulierte Fähigkeiten gesteuert und ausgelesen werden, wofür ein Einplatinencomputer die Basis bildet, der auch bei einer Realisierung eingesetzt werden kann. Für statische Objekte ohne Interaktionsmöglichkeiten kann die Selbstbeschreibung in einen QR-Code kodiert werden, der auf eine Instanz URI verweist, in der die entsprechenden 3D Modelle assoziiert sind. In Kombination mit einem externen Tracking wird ein Bezugskoordinatensystem erstellt, dass eine simultane Verwendung der digitalen Zwillinge und realer Hardware ermöglicht.

Zusammengefasst dienen die vorgestellten Werkzeuge nicht nur der Methodologie sondern demonstrieren gleichzeitig das Potential von Semantic Plug & Play . Im folgenden Kapitel wird dieses Potential auf realer Hardware illustriert, beginnend mit der Forschungsplattform, die im Rahmen der vorliegenden Dissertation entwickelt wurde.

Zusammenfassung. Das Konzept von Semantic Plug & Play zielt darauf ab, generisch und universell einsetzbar zu sein. Um dies zu zeigen, werden in diesem Kapitel unterschiedliche Prototypen in den einzelnen Experimenten vorgestellt und in der Fallstudie verortet. Dabei werden nicht nur domänenspezifische Lösungen präsentiert, sondern universell einsetzbare Systeme, Platinen und ein Forschungsquadrocopter.

8 Prototypen und Auswertung der Fallstudie

8.1 Hardwareprototypen & Trackingsysteme **150**
8.1.1 Der Forschungsquadrocopter 151
8.1.2 Die Flugarena 153
8.2 Virtuelle Messtürme **155**
8.3 Synchrone Glasfasermessung **161**
8.4 Ausbreitungsmodellierung **168**
8.5 Der sensorbasierte Flug **172**
8.6 Pick & Place **178**
8.7 GoLive **183**
8.8 Zusammenfassung der Experimente **189**

Semantic Plug & Play beinhaltet eine Architektur für die Selbstbeschreibung von Hardware aufbauend auf Semantic Web-Technologien. Angewendet wird die Selbstbeschreibung über die abstrakte Modellierung von Fähigkeiten und Eigenschaften konkreter Hardware. Die vorgestellten Werkzeuge vereinfachen nicht nur die Schritte der Methodologie für die Verwendung von Semantic Plug & Play, wie die Annotation der Individuals, sondern erweitern das Robot Operating System (ROS) um Modellierungswerkzeuge für Prozesse und die Infrastruktur. Die Kombination aus Semantic Plug & Play und ROS bildet die Grundlage für eine Simulation mit interaktiven MR-Bausteinen und einer schrittweisen Realisierung durch die Interaktion simulierter und realer Objekte. Das Ziel der Simulation ist die Überführung eines Programms in die Realität, also dessen Anwendung auf realer Hardware. Der Schritt in die Realität wird in diesem Kapitel anhand unterschiedlicher Experimente aufgezeigt. Zur Prüfung der Universalität des Ansatzes von Semantic Plug & Play wurde in Kapitel 3 die Fallstudie eines Katastrophenszenarios der Feuerwehr eingeführt. Aufbauend auf dieser Fallstudie wurden die sogenannten *ScORe* Missionen klassifiziert und auf mehrere Experimente aufgeteilt. Jedes Experiment bildet sowohl eine Teildisziplin der Fallstudie, als auch selbst eine ScORe Mission ab, wie in Abbildung 8.1 illustriert. Dabei werden jeweils unterschiedliche Technologien des Semantic Plug & Play-Technologiestacks verwendet und es wird ein Hauptfokus gesetzt.

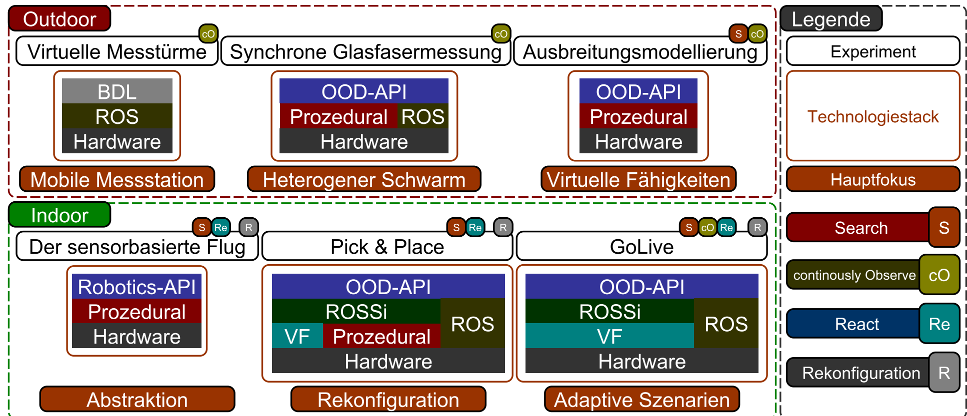

Abbildung 8.1. Eingliederung der Experimente in das ScORe-Schema, die Umgebung, den verwendeten Technologiestack und den Hauptfokus. (Eigenwerk)

Das Pick & Place-Beispiel verwendet exemplarisch den vollen Technologiestack von Semantic Plug & Play mit Ausnahme von BDL und fokussiert dabei die Rekonfiguration. Anhand des ScORe-Schema kategorisiert, ist hingegen nur ein *Search* und *React* in dem Experiment realisiert. Jedes Experiment ist anhand echter Hardware realisiert und entweder *outdoor* oder *indoor* in den Laboren durchgeführt und dokumentiert worden. Als Hardwarebasis dient ein modifizierter Quadrocopter, der im Rahmen der Dissertation als Auftragsarbeit von EMCOTEC GmbH realisiert wurde und von ihnen mittlerweile als reguläres Produkt gefertigt wird. Mit der Universität als Lizenzgeber wird der Quadrocopter bei der Firma Rosewhite als Indoor- und Outdoorvariante vertrieben. Die Indoorszenarien der Experimente finden in den Flugarenen des *Instituts für Software & Systems Engineering* (ISSE) statt, deren Konzept, Aufbau und Integration ebenfalls im Rahmen der Dissertation stattgefunden haben. Nachfolgend werden die einzelnen Experimente detailliert, und mit einem Fokus auf die den Experimenten zugehörigen Prototypen und den erzielten Ergebnissen beschrieben.

8.1 Hardwareprototypen & Trackingsysteme

Der Einsatz von Semantic Plug & Play und ROS erfordert keine spezielle Hardware, und unterstützt Arduino-kompatible Mikrocontroller, wie bspw. den ESP-8266 [198] und Einplatinencomputer mit Linux Kernel, wie bspw. die Raspberry Pi-Plattform oder Replika [205]. Die Ausführung des Semantic Controllers benötigt einen Einplatinencomputer. Der Semantic Adapter kann sowohl auf einem Einplatinencomputer als auch auf einem Mikrocontroller verteilt werden. Sowohl Mikrocontroller als auch Einplatinencomputer können ebenfalls als Basis für die Flugsteuerung eines Quadrocopters dienen, wie etwa ein Arduino Uno R3 (siehe Ghosh et al. [96]) oder ein Raspberry PI, der in dem Erle Copter System [163] verwendet wird. Die Mikrocontroller-Variante bietet nur begrenzte Erweiterungsmöglichkeiten für externe Sensoren, da der ATmega328P-Chip bereits für die Regelung des Quadrocopters ausgelastet ist und eine Positionssteuerung über

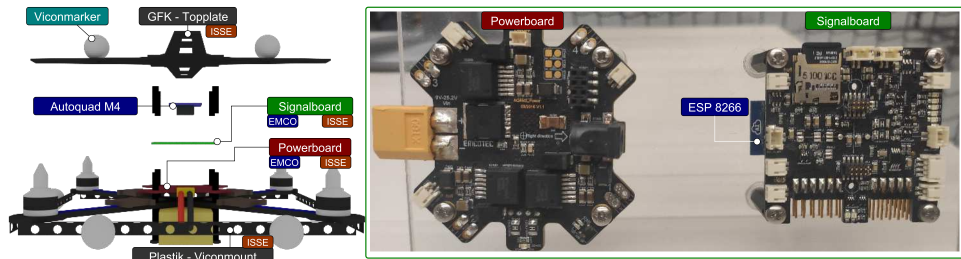

Abbildung 8.2. Explosionszeichnung des Forschungsquadrocopters mit der Grundausstattung für Indooranwendungen. Das von EMCOTEC gefertigte Signalboard ist rechts detailliert abgebildet. Das ISSE symbolisiert, dass das Institut entweder an der Entwicklung beteiligt war (Signal-, Powerboard), oder dass jeweilige Teile gänzlich im Rahmen der Dissertation entstanden ist (Topplate, Viconmount). (Eigenwerk)

GPS nur mit Optimierungen des Codes integriert werden konnte [96]. Der Raspberry Pi von Erle bietet hingegen genügend Ressourcen, verwendet aber ein spezielles, auf Debian aufgebautes Betriebssystem mit Unterstützung von ROS 1 Hydro Medusa, eine bereits eingestellte ROS-Version. Die Separierung der Flugcontrollerhardware durch einen zusätzlichen Einplatinencomputer bietet eine austauschbare Basis für die Ausführung des Semantic Controllers, der auf einem Standardbetriebssystem wie Ubuntu auch für neue ROS-Versionen geeignet ist. Zusätzlich müssen keine Echtzeitkriterien erfüllt werden wie bei der Regelung eines Quadrocopters, dessen Regelkreis durch zeitintensive Interrupts unterbrochen werden kann. Ein zusätzlicher Einplatinencomputer sowie modulare Sensorik erfordern jedoch zusätzliche konstante Spannungsversorgungen sowie Schnittstellen für die Kommunikation untereinander. Für diese Ansprüche ist im Rahmen der Dissertation eine Forschungsplattform in Kooperation mit EMCOTEC und Rosewhite entwickelt worden.

8.1.1 Der Forschungsquadrocopter

Da zahlreiche und verschiedene Sensoren und Aktuatoren in den Experimenten und der Fallstudie der vorliegenden Dissertation verwendet werden, ist ein Forschungsquadrocopter entwickelt worden, der den damit einhergehenden umfangreichen Anforderungen gerecht wird. Der Forschungsquadrocopter basiert auf einen Rahmen aus kohlenstofffaserverstärktem Kunststoff (CFK) der Firma Rosewhite, der mit einem Autoquad M4-Flugcontroller in der Basisvariante ausgestattet ist. Die Firmware des Controllers ist Open-Source verfügbar und kann beliebig erweitert werden, die physikalischen Schnittstellen werden in der Standardplatine allerdings nicht im vollen Umfang nach außen geführt. Die Grundidee ist, die Schnittstellen in einem Breakoutboard für die Kommunikation mit externen Systemen zu erweitern und eine stabile Stromversorgung für externe Komponenten bereitzustellen. Zusätzliche Bauteile zur Unterstützung der Forschungsarbeit, wie ein Buzzer für akustische Signale oder ein SD-Karten-Slot für das Aufzeichnen der Flugdaten, finden sich auf der Signalplatine des Forschungsquadrocopters. Die technischen Merkmale der Power- und Signalplatine, wie in Abbildung 8.2 aufgeführt, umfassen:

5V 5A Stromversorgung
Externe Geräte können über die Pins des Powerboards mit Strom versorgt werden. Low-Drop-Out (LDO) Spannungsregler sorgen für eine konstante Spannungsversorgung, die über 4S-6S-Akkumulatoren gespeist werden.

ESP8266
Die serielle Schnittstelle des Autoquad ist direkt mit einem auf die Signalplatine gelöteten ESP8266-Modul (ESP) verbunden, damit eigene Anwendungen programmiert werden können. Durch das integrierte WLAN des ESP werden bspw. bei Indoor-Szenarien die Trackingdaten im ESP adaptiert und dem Autoquad M4 bereitgestellt.

Buzzer
Neben der Telemetrie kann auch ein akustischer Signalton verwendet werden, um Debugginginformationen oder einen niedrigen Akkustand zu signalisieren. Der zusätzliche Buzzer kann über einen Jumper mit dem ESP verbunden und individuell angesteuert werden. Alternativ wird eine akustische Warnung über den niedrigen Akkustand ausgegeben.

SD-Karten-Slot
Über die externe SD-Karte können Log-Dateien des Autoquad M4 gespeichert werden, die bspw. die Odometriedaten oder den Trajektorienverlauf des Quadrocopters beinhalten.

Externe Schnittstellen
Die verbleibenden Schnittstellen des ESP und Autoquad M4 werden hauptsächlich über Pins nach außen geführt. Darunter sind bspw. der CAN Bus, UART und I2C. Speziell für die externe Beleuchtung bei Flügen in der Dämmerung können vier LED-Ausgänge mit Jumpern über den ESP gesteuert, konstant eingeschaltet oder deaktiviert werden. Die Signalspannung kann ebenfalls über Jumper zwischen 1.5 und 5,5V gesetzt werden.

Die Platinen bilden die Grundlage für die Indoor- und Outdoorvariante des Forschungsquadrocopters, die in Abbildung 8.3 abgebildet sind. Die Indoorvariante nutzt für das Trackingsystem sogenannte Vicon-Marker, die auf der aus glasfaserverstärktem Kunststoff (GFK) gefertigten Topplate und dem Viconmount aus Plastik montiert sind. Vicon ist der Hersteller des optischen Trackinsystems, das im nächsten Kapitel genauer vorgestellt wird. Je nach Einsatzszenario und Experiment können unterschiedliche Adapterplatten unterhalb des Quadrocopters befestigt werden, die bspw. einen Einplatinencomputer oder eine Kamera tragen. Die konkreten Erweiterungen werden in den Experimenten separiert vorgestellt. Die Outdoorvariante aus Abbildung 8.3 nutzt ein Landegestell aus CFK mit 3D-gedruckten Halterungen aus Polyactid (PLA) für eine werkzeuglose Montage.

Die Hardware des Forschungsquadrocopters wird durch die Firma Rosewhite lizenziert vertrieben[1]. Die technischen Zeichnungen zum Fräsen oder Meshes für den 3D-Druck, die außerhalb der Kooperation mit EMCOTEC entstanden sind, wie der Vicon-Rahmen

[1] https://rosewhite.flydrotec.de/wissenschaftliche-drohnen

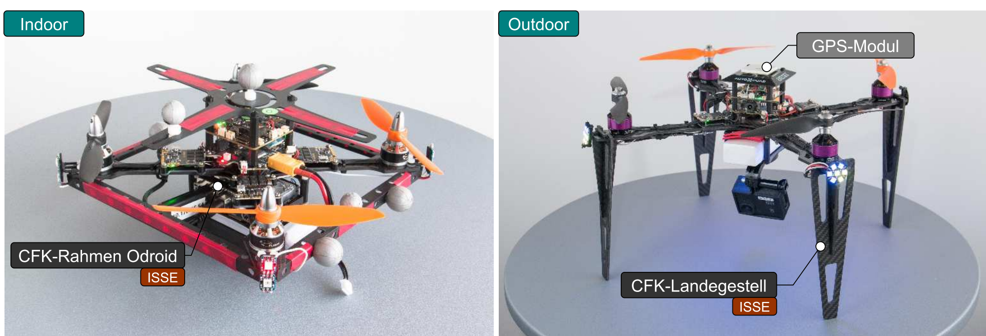

Abbildung 8.3. Fotos der Indoor- und Outdoorvariante des Forschungsquadrocopters. Die Indoorvariante nutzt Vicon-Marker für die Positionssteuerung, während die Outdoorvariante eine GPS-Antenne verwendet. Verschiedene aus CFK gefertigte Adapterplatten unterhalb des Quadrocopters erlauben bspw. die Montage eines Odroid XU4-Einplatinencomputers. (Eigenwerk)

oder das Landegestell, stehen aktuell unter keiner Lizenz. Die Indoorvariante mit Vicon-Markern ist speziell für die Erstellung unterscheidbarer Markermuster konzipiert, die über das Trackingsystem in der Flugarena des Instituts erkannt und eindeutig zugeordnet werden können.

8.1.2 Die Flugarena

Für die Positionssteuerung des Quadrocopters in geschlossenen Räumen existieren unterschiedliche Verfahren, die über eine Triangulation die Position im Raum bestimmen können. Über den Funkstandard Ultra-Wideband (UWB) werden bspw. Transceiver im Raum verteilt und eingemessen, ein weiterer Transceiver befindet sich für die Triangulation auf dem Quadrocopter. Dieses Verfahren wird bei Gou et. al [113] eingesetzt und mit einem deutlich präziseren optischen Tracking als Grundwahrheit evaluiert. Das eingesetzte optische Trackingsystem ist von der Firma Vicon und wird in zahlreichen Forschungsabteilungen verwendet, wie etwa der Flying Machines Arena der ETH Zürich [71] oder der Indoorarena der University of Pennsylvania [199]. Im Rahmen der vorliegenden Dissertation sind zwei Flugarenen mit dem Vicon-System für die Evaluation konzipiert und realisiert worden: Eine große Variante für Schwarmexperimente[2] mit 16 Kameras, wie in Abbildung 8.4 abgebildet, und eine mobile Variante für kleinere Szenarien[3], die bpsw. bei Konferenzen [151] eingesetzt wird. Die verlinkten Videos in den Fußnoten zeigen die Varianten in der Realität. Beide Varianten bauen auf einem Traversensystem auf, das in Abbildung 8.4 abgebildet ist. An dem Traversenring werden die Kameras montiert und über Ethernet an das Vicon-Rack angeschlossen. Die Software des Vicons bietet neben einer Kalibrierung und der Definition eines Ursprungs auch die Verwendung von eindeutigen Mustern für die Identifikation von Objekten

[2]ATV Beitrag ab Minute 19: https://www.augsburg.tv/mediathek/video/vorsprung-schwaben-roboterforschung-ladestellen-fuer-e-autos-flugtaxi/
[3]https://www.youtube.com/watch?v=G8JHShUlQY0

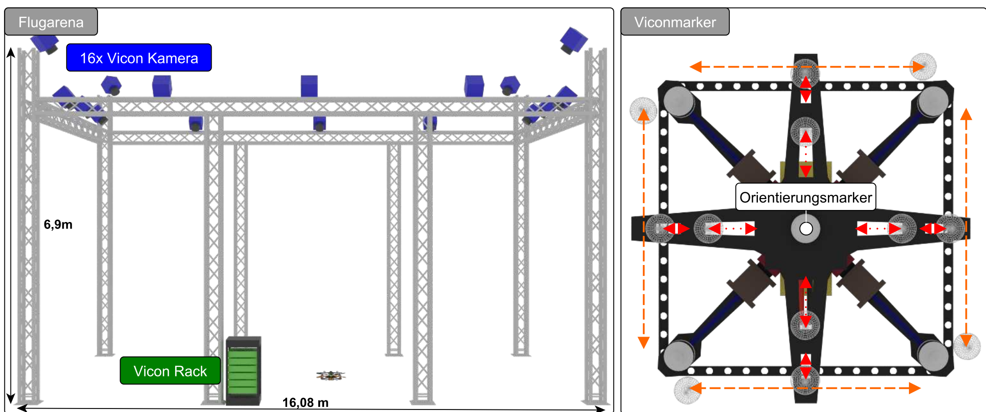

Abbildung 8.4. In der Abbildung links ist der strukturelle Aufbau der Flugarena skizziert, mit 16 Kameras, die an einem Traversensystem angebracht sind. Das rechte Bild zeigt die möglichen Markerpositionen der Topplate und Seitenteile des Indoor-Forschungsquadrocopters. (Eigenwerk)

und deren Rotationserkennung an. Ein Muster wird durch mindestens drei Marker definiert, die für die Zuordnung zu einem konkreten Objekt unterschiedlich angebracht werden müssen. Für die Montage der Marker bieten die Indoor-Forschungsquadrocopter unterschiedliche Positionen an, die in Abbildung 8.4 im rechten Teil skizziert sind. Die GFK-Topplate unterstützt bis zu acht verschiebbare Markerpositionen und einen mittigen Orientierungsmarker. Die aus Plastik gefertigten, seitlichen Viconmounts, die ebenfalls zu Befestigung von Markern verwendet werden können, verwenden ein Lochraster mit fixen Positionen, sodass eine leichte Kollision nicht zu einem Versatz der Markerposition führt. Die vergleichbaren Systeme der University of Pennsylvania verwenden keine Muster sondern die Historie der Markerpositionen für ein durchgängiges Tracking. Muster bieten hingegen die Information der Rotation eines Quadrocopters sowie eine automatische Unterscheidung mehrerer Quadrocopter unabhängig zusätzlicher Identifikationsmechanismen. Speziell in Experimenten mit mehreren Quadrocoptern können die Aufzeichnungen der Muster in der Auswertung und Fehleranalyse auch lange nach der Ausführung des Experiments im Vicon eindeutig dem entsprechenden Quadrocopter zugeordnet werden.

Die Genauigkeit des Vicon-Systems hängt von unterschiedlichen Faktoren ab und kann nicht pauschal angegeben werden. So befinden sich die Kameras des Typs *Vantage V16* in der großen Variante aus Abbildung 8.4 auf einer Höhe von ca. 6 Meter. Die Marker haben unterschiedliche Größen und die Kalibrierung der Kameras kann durch minimale Erschütterungen die Messergebnisse beeinflussen. In einem Laborversuch von Merriaux et al. [201] lag die gemessene Genauigkeit des Vicon-Systems bei einem Maximum von 0.15 mm bei unbeweglichen Objekten. Dieser Wert ist allerdings eher als Einordnung der Größenordnung zu betrachten, da sich die Montageart, das Kameramodell und die Anzahl der Kameras zu den Flugarenen unterscheiden.

Die Outdoor-Variante des Forschungsquadrocopters errechnet die Position durch das Global Positioning System (GPS) in Kombination mit einem Barometer und dem Lagesensor der Integrated Measurement Unit (IMU). Filteralgorithmen und Umweltbedingungen, wie etwa seitlicher Wind, beeinflussen die Genauigkeit, wobei eine Größenordnung im einstelligem Dezimeterbereich als Richtwert dient [97]. Die folgenden Experimente sind in Indoor- und Outdoorkategorien unterteilt und verwenden dementsprechend die Flugarena oder das Vicon-Trackingsystem für die Positionsregelung.

8.2 Virtuelle Messtürme

Das Karlsruher Institut für Technologie (KIT) veranstaltet in unregelmäßigen Abständen kooperative Messkampagnen mit dem Namen *ScaleX* zur Bestimmung der langfristigen ökologischen und klimatischen Auswirkungen des globalen Klimawandels auf regionaler Ebene [317]. Für die Erhebung der Messwerte werden neben stationärer Sensorik vor allem mobile Plattformen verwendet, wie bspw. unbemannte Starrflügler oder motorisierte Hängegleiter, die jeweils mit unterschiedlicher Sensorik ausgestattet sind. Das Institut für Software & Systems Engineering (ISSE) der Universität Augsburg hat an ScaleX teilgenommen und das Experiment der *Virtuellen Messtürme* realisiert. Im Rahmen der vorliegenden Dissertation sind die Prototypen der Quadrocopter mit einer Messvorrichtung, sowie die Verteilung und Ablauf der Missionen entstanden. Die visuelle Aufbereitung der Messdaten hingegen wird in der Dissertation von meinem Kollegen Oliver Kosak beschrieben [150]. Das Experiment hat an zwei Tagen der Messkampagne ScaleX in Fendt stattgefunden, einem Außenmessbereich des KIT mit unterschiedlichen stationären Sensoren, das unter anderem für die Beobachtung von Grenzschichten errichtet wurde. Die Vorteile der Quadrocopter im Gegensatz zu den Starrflüglern finden sich in längeren in situ Messperioden, also der Möglichkeit, länger an einer Stelle in der Luft zu verweilen und Messungen vorzunehmen. Der Ablauf des Experiments umfasst einen koordinierten Auf- und Abstieg an drei Orten. Jedem Quadrocopter wird dafür eine Mission aufgespielt, die an drei Höhenpunkten für wenige Minuten einen Haltepunkt definiert hat. Der Missionsstart der drei Quadrocopter wird manuell über Funkgeräte initiiert. Die Durchführung des ca. 36-stündigen Experiments mit einem stündlichen Auf- und Abstieg der Forschungsquadrocopter ist von mehreren Mitarbeitern und Hilfswissenschaftlern des ISSE unterstützt worden.

Prototypen

Für das Experiment wurde das Grundgestell der Indoorvariante verwendet und für die Anforderungen der Messkampagne erweitert. Die Basis für die Erweiterung des Prototypen bildet ein Montagekreuz, das unter den Quadrocopter angebracht wird, wie in Abbildung 8.5 illustriert. An ihm ist eine zentrale Einheit in Form eines Einplatinencomputers montiert, des Odroid XU4 mit einem Linuxderivat als Betriebssystem. Er dient der Missionsverwaltung sowie dem Zusammenführen und Persistieren der Messdaten in einem Quadrupel aus Uhrzeit, Quadrocopterposition und den Messwerten beider Sensoren, sowohl für die Temperatur als auch die Luftfeuchte. Konzeptuell werden zwei Arduino Nano verwendet, die jeweils einen SHT75-Sensor für den Einplatinencomputer adaptieren. In der Realisierung ist gewichtsbedingt nur ein Arduino Nano

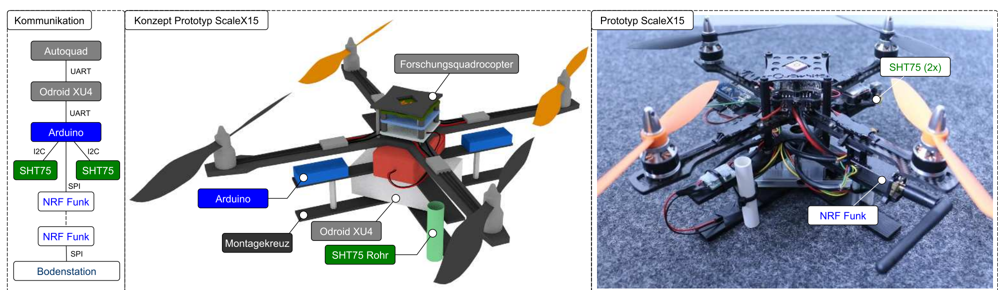

Abbildung 8.5. Abbildung des Konzepts für den Prototypen und die tatsächliche Realisierung mit Erweiterungen. Dieser Prototyp ist in fünffacher Ausfertigung realisiert, um eine Redundanz für Ausfälle während der ScaleX Messkampagne zu gewährleisten. (Eigenwerk)

für zwei SHT75-Sensoren verwendet worden, sowie ein NRF24L01 Funkmodul (NRF) für die Überwachung der Messdaten während des Fluges, das an eine Bodenstation die Messwerte via Funk überträgt. Das in Abbildung 8.5 angedeutete SHT75-Rohr resultiert aus Beobachtungen der ersten Versuche, in denen sich der SHT75-Sensor durch die Sonneneinstrahlung erwärmte und erst nach wenigen Minuten plausible Temperaturwerte lieferte. Das Rohr umgibt die Messfläche des Temperatursensors zum Schutz direkter Sonneneinstrahlung und wird unter den Rotoren angebracht, damit der Downwash während des Fluges eine ausreichende Luftzirkulation verursacht. Das in Abbildung 8.6 skizzierte Verfahren vermindert zwar einen Wärmestau an der Messfläche, sorgt jedoch für eine lokale Abweichung, da die Luft oberhalb des Quadrocopters gemessen wird. Eine andere Forschergruppe hat zu einem späteren Zeitpunkt dieses Verfahren genauer untersucht und in Abhängigkeit zur Sonneneinstrahlung und Wind evaluiert [109]. Für die geforderte Genauigkeit der Messkampagne ist der Versatz der Messung nicht relevant, da die stationären Sensoren eine erheblich höhere Ungenauigkeit aufweisen und eine Verbesserung der bisherigen Verfahren mit den Quadrocoptern erzielt werden konnte.

Ergebnisse

Als initiales Experiment im zeitlichen Verlauf der Dissertation sind nur die Grundprinzipien der Verteilung von Semantic Plug & Play angewendet worden. So diente eine in Python geschriebene Applikation für die Verteilung und Ausführung einer Mission auf dem Odroid XU4 und der Sammlung der Messdaten, die aufgrund unterschiedlicher Frequenzen der Sensoren nicht in einem Quadrupel zusammengefasst, sondern als Tupel mit der Uhrzeit persistiert wurden. Erst in der Nachbereitung der Messdaten konnten durch einen definierten Zeitraum die Mess- und Positionsdaten zusammengefasst werden. Eine Auswertung eines synchronen Aufstiegs ist in Abbildung 8.7 zu sehen, wobei die Messwerte von den zwei Sensoren in erhöhter Frequenz als Bias abgebildet sind. Eine potentielle Grenzschicht zeichnet sich in der Höhe von 60 und 90 Metern ab, wenn die Temperaturkurve einknickt und der Temperaturaufstieg in einen -abstieg übergeht.

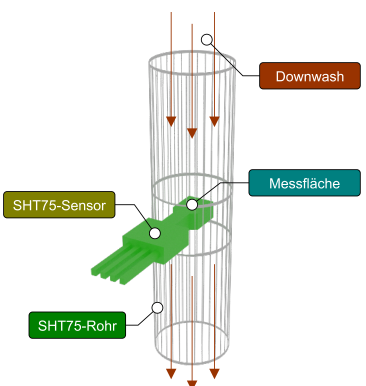

Abbildung 8.6. Aufbau des SHT75 Sensors in einem weißen Rohr, das unter den Rotoren des Quadrocopters befestigt wurde. Die Messfläche des Sensors befindet sich innerhalb des Rohres. (Eigenwerk)

Mit diesem Experiment ist es geglückt, den Quadrocopter als Messinstrument in der Forschung der Geographie, im speziellen zur Nachverfolgung von Grenzschichten, erfolgreich einzusetzen. Die Messpunkte, in der die Quadrocopter längere Zeit während einer Mission verweilen, sind in der Kurve nicht zu erkennen, da einerseits der Aufstieg mit geringer Geschwindigkeit (<1 Meter/s) erfolgt ist und andererseits die beiden Sensoren in einer Frequenz von 4 Hz, also minimal 8 Messwerte pro Meter, aufgezeichnet haben.

Die negativen Erfahrungen des experimentellen Aufbaus sind folgend zusammengefasst:

1. Bei einem Hardwaredefekt ist eine Fehleranalyse sehr aufwendig, da Mikrocontroller mit Ausnahme der vorhandenen LEDs, keine visuelle Ausgabe haben.
2. Die feste Verdrahtung der Sensoren führt zu erheblichem Aufwand bei einem Austausch eines defekten Sensors oder bei Erweiterungen, wie etwa durch unterschiedliche Sensoren.
3. Für die Interpretation der Messdaten ist eine detaillierte Dokumentation essentiell. Die Tupel aus dem Zeitstempel und dem Messwert müssen den Tupeln der Position und des konkreten Quadrocopters zugeordnet werden. Auch der konkrete Sensor bei redundanten Systemen dient der späteren Interpretation.
4. Der manuelle Missionsstart durch die Synchronisation zwischen Piloten führt zu einem erhöhten Personalbedarf und menschlichen Fehlern, die wiederum zu Defekten führen.

Die negativen Erfahrungen des Experiments und die bis dato mangelnde Umsetzung der Semantic Plug & Play-Prinzipien sind zu einem späteren Zeitpunkt der Entwicklung von Semantic Plug & Play aufgegriffen worden, um eine mobile Messstation für meteorologische Messungen bereitzustellen, die ohne Expertenwissen mit modularer Hardware konfigurierbar ist.

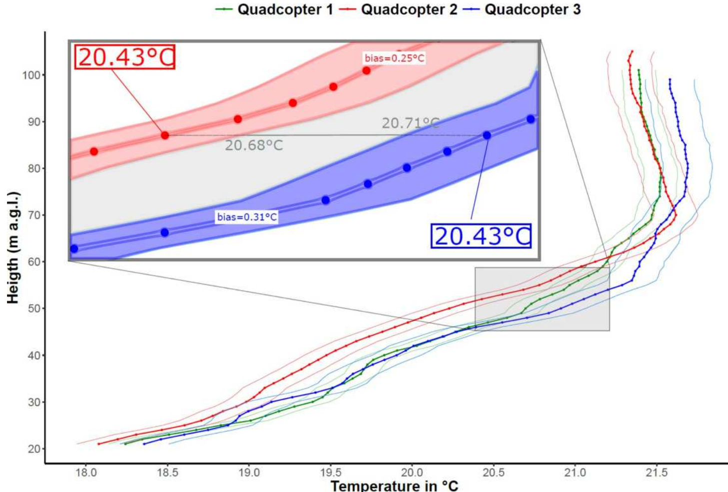

Abbildung 8.7. Auswertung eines synchronen Messfluges mit drei Quadrocoptern in der ScaleX-Kampagne. Die Messwerte beziehen sich auf die zwei SHT75-Sensoren mit mehreren Messungen in den jeweiligen Höhen. (Bild entnommen aus der kooperativen Publikation [153])

Die mobile Messstation

Die mobile Messstation besteht aus einer Weiterentwicklung des Prototypen der Messkampagne und einer modularen Sensorplattform in Form einer Platine für den ESP 8266 Mikrocontroller, der zu Arduino kompatibel ist, eine stärkere Recheneinheit verwendet und WLAN besitzt. Das Platinenlayout aus Abbildung 8.8 lehnt sich an das Design des NodeMCU an, das unter anderem für Internet of Things-Anwendungen verwendet wird [222]. Es unterscheidet sich jedoch in einigen Aspekten. So bietet das Platinenlayout, das im Rahmen der Dissertation entwickelt wurde, einen USB-C Stecker, einen CH340G USB-Serial Converter und einen Adapter-Steckplatz für ein OLED-Display, das oberhalb des USB-Steckers angebracht wird, um Statusinformationen bereitzustellen, wie in Abbildung 8.8 illustriert. Die freien Pins an den Seiten können für die Integration von Sensoren verwendet werden, während die bestückte Platine über ein USB-Kabel mit dem Einplatinencomputer kommuniziert. Damit ein sicherer Halt für den adaptierten Sensor gewährt werden kann, kommen zusätzlich zwei Neodym-Magnete zwischen den USB-C Anschluss und das OLED-Display, das von einem aus Polyethylen (PETG) gedruckten Steg fixiert wird, wie in Abbildung 8.8 zu sehen. Am Quadrocopter befindet sich das ebenfalls mit PETG gedruckte Gegenstück mit Neodym-Magneten an den Rotohrenstegen. Dieses Stecksystem ermöglicht einen einfachen Tausch der Sensoren bei einem Defekt oder wenn andere Sensoren eingesetzt werden sollen. Zur Diagnose dient das OLED-Display auf der Platine, das bspw. in Abbildung 8.8 die Temperatur, Frequenz und den Kalibrierstatus ausgibt. Die mobile Messstation bietet die Grundlage für die Integration des Experiments in die Block Definition Language (BDL) in Kombination mit den Semantic Plug & Play-Verteilungsstrategien und der Selbstbeschreibung für die Auswertung der Messdaten.

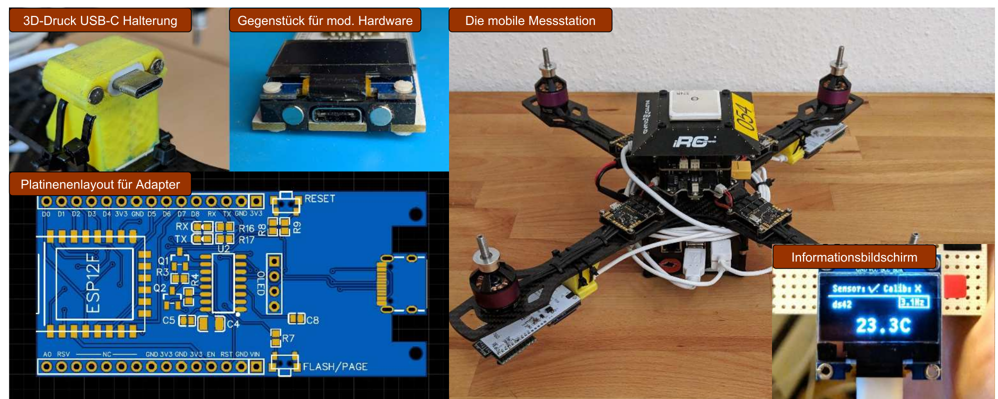

Abbildung 8.8. Abbildung der mobilen Messstation mit modularem Hardwareadapter über die USB-C Schnittstelle. Die Platine führt die Anschlüsse eines ESP-Moduls nach außen und besitzt einen Anschluss für einen Informationsbildschirm, der oberhalb des USB-C-Anschlusses montiert werden kann. (Fotos aufgenommen von Martin Schörner, ISSE)

Der mobile Messturm in BDL

Als Grundsystem für die Modernisierung des Experiments dient das Robot Operating System (ROS) für die Abstraktion der Fähigkeiten. Auf der Basis einer domänenspezifischen Anwendung ohne Rekonfiguration und Variation der Sensorik kommen die Semantic Plug & Play-Aspekte, wie die Spezifikation der Sensoren, erst in der Auswertung der Messdaten zum tragen. Die Mission hingegen kann durch das Dynamic Mission Definition and Execution (DMDE)-Framework ausgedrückt werden. Die in Abbildung 8.9 abgebildete Mission wird auf jeden Quadrocopter automatisiert durch die Semantic Plug & Play-Verteilgungsalgorithmik über GIT verteilt und von einer Bodenkontrollstation gestartet. Nach der Synchronisierung der Initialisierungsphase wird der Quadrocopter aktiviert (*arming(true)*) und in einer Höhe von einem Meter erneut synchronisiert, bevor der experimentelle Ablauf beginnt. Der Vorteil der Missionsdefinition im Gegensatz zu der manuellen Variante ist die Skalierbarkeit der Teilnehmer. So werden in der initialen Barriere alle Quadrocopter verwendet, die aktuell verfügbar sind. In der Mission werden lediglich zwei Haltepunkte mit einer Höhe von 2 und 3 Metern definiert, da die Testversuche in der Indoorarena durchgeführt wurden. Die Replizierbarkeit in Feldexperimenten wird in den folgenden Experimenten aufgezeigt. Damit die Personenzahl ebenfalls reduziert werden kann, muss jedoch eine Übernahme durch nur einen Piloten mit manueller Steuerung sichergestellt werden. Dieses Konzept wird ebenfalls im nächsten Experiment, der synchronen Glasfasermessung, vorgestellt.

Die Selbstbeschreibung der Sensoren dient in diesem Beispiel der Nachbereitung und Interpretation der Sensormesswerte. Durch die Kommunikation über ROS dient die Bodenkontrollstation als zentrales Element für die Persistierung der Messwerte. Als Grundlage für die Bereitstellung der Messergebnisse wird die von der W3C empfohlene Semantic Sensor Network (SSN) Ontology verwendet, die bereits bei meteorologischen Messungen eingesetzt wird [51]. Die Kontrollstation sammelt die Selbstbeschreibung der Sensoren und die Messwerte. Für die Bereitstellung kann die Semantic Hardware

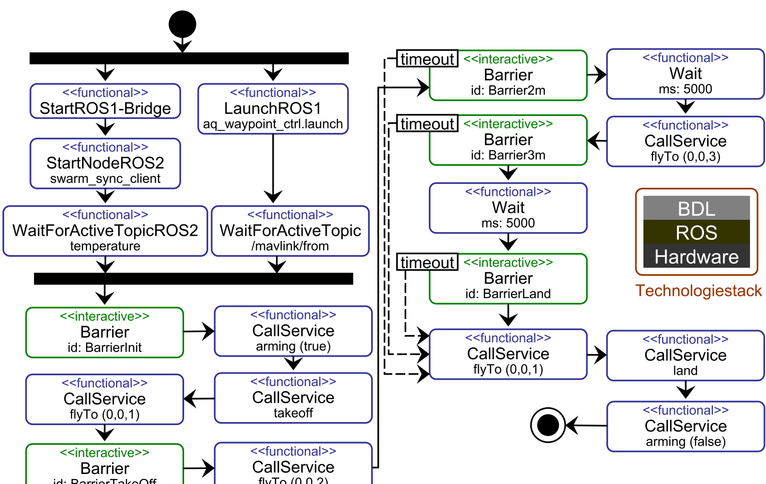

Abbildung 8.9. Mission der einzelnen Quadrocopter in der BDL modelliert. Synchronisationspunkte sind durch Barrieren mit einer Timeoutfunktion realisiert. Der verwendete Technologiestack der Anwendung ist rechts abgebildet. (Eigenwerk)

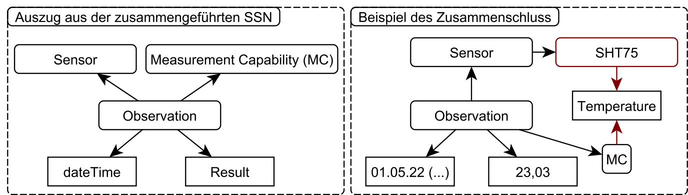

Abbildung 8.10. Abbildung der Verknüpfung zwischen der SSN und der Semantic Hardware Web Ontology. (Eigenwerk)

Web (SHW) Ontology in das Format der SSN bei Bedarf konvertiert werden, bspw. durch *same as* Assoziationen. So werden in der SHW Eigenschaften wie die Latenz des Sensors durch *Properties* ausgedrückt, in der SSN hingegen über das Subjekt *Measurement Capability*. Dieses Vorgehen bildet kein Novum, sondern ist eine bereits praktizierte Technik in der Bereitstellung von Messwerten in der Forschung [319]. Das Novum bildet hingegen die verteilte Speicherung der Selbstbeschreibungen direkt an den Sensoren, die eine Nachbereitung vereinfachen. Das Projekt von Dibley et al. [61] sieht zwar einen Einplatinencomputer für die Integration der Messwerte in Kombination mit SSN vor, verfolgt aber kein modulares System mit Selbstbeschreibungsmechanismen.

Einordnung in die Fallstudie

Die Messkampagne im Forschungsfeld meteorologischer Messungen bildet die Grundlage für eine einfache *continously Observer* (cO) Mission im *ScORe*-Kontext. Die Fallstudie des Gasunfalls als vollständige ScORe-Mission verwendet die in situ Messung mit unterschiedlichen Sensoren in fast allen Phasen, beginnend bei der Gasquellenlokalisierung, der Ausbreitungsobservierung und der Überwachung kritischer Infrastrukturen. Die

Bereitstellung der Messdaten verknüpft mit einer Ontologie bietet die Möglichkeit, über einen SPARQL-Endpoint auf die Graphen zuzugreifen und den Einsatz anhand der Messwerte nachträglich auszuwerten. Die mobile Messstation bietet einen ersten Prototypen für modulare Sensoren, die mit einfachen Steckmechanismen ausgetauscht werden können und über einen Bildschirm einen Einblick in den Status des Sensors bieten.

Der Feldversuch im Rahmen des ScaleX-Experiments ist in den Publikationen [153, 317] veröffentlicht. Die Erweiterung mit der Block Definition Language (BDL) und der mobilen Messstation ist in der Publikation [268] publiziert. Das Platinenlayout, der Bestückungsplan und die 3D-Meshes für den Druck sind lizenzfrei. Mit der BDL können bereits einfache Experimente mit mehreren Teilnehmern realisiert werden. Ein Schwarm verwendet emergente Effekte und bietet eine Robustheit bei Ausfällen, die in BDL nicht ausgedrückt werden können. Durch Einbezug der Rekonfiguration einzelner Systeme in einem heterogenen Ensemble wird der Begriff *Multipotente Systeme* definiert. Einen ersten Schritt in die Richtung eines heterogenen Schwarms multipotenter Systeme bietet das nächste Experiment: Die synchrone Glasfasermessung.

8.3 Synchrone Glasfasermessung

Die synchrone Glasfasermessung bietet eine erste nicht zentral koordinierte Mission mit einem heterogenen Roboterschwarm bestehend aus einem Bodenroboter und drei Quadrocoptern. Das Ziel des Experiments ist eine horizontale, engmaschige Temperaturmessung, um eine genauere Analyse einer Grenzschichtverteilung zu gewinnen. Das Experiment reiht sich in die ScaleX-Kampagne des KIT ein und ereignete sich ebenfalls in Fendt. Für das Experiment kommen drei spezielle Quadrocopter zum Einsatz, die ein Lichtwellenleiterkabel (LWL-Kabel) im Abstand von zehn Metern tragen, dessen Auswertungseinheit (DTS) an einer mobilen Roboterplattform angebracht ist und die Temperatur im Abstand von 10 cm entlang des LWL-Kabels messen kann. Analog zu dem Experiment der mobilen Messtürme wird ein Sicherheitspilot für jeden Quadrocopter benötigt, der ebenfalls Start und Landung ausführt. Damit sich das LWL-Kabel nicht im Gras verfängt, sind zwischen den Quadrocoptern und der mobilen Plattform LWL-Ablagen aufgestellt und Bodenmatten für die Quadrocopter ausgelegt, wie in Abbildung 8.11 abgebildet. Sobald die Quadrocopter eine Höhe von 50 Metern erreichen, setzt ein Schwarmalgorithmus ein, der die gewünschten Positionen für den Messflug kommandiert, also das Ensemble wie in Abbildung 5.13 skizziert nach vorne bewegt. Die Implementierung des Schwarmalgorithmus für die Bewegung ist Bestandteil der Dissertation meines Kollegen Oliver Kosak [150], das Konzept und die Realisierung der Prototypen sowie deren Ansteuerung über Fähigkeiten ist hingegen Bestandteil der vorliegenden Dissertation.

Prototypen
Die mobile Plattform baut auf der Basis der INNOK Robotics Plattform auf und bietet sowohl einen Rechner als auch GPS, Funk und einen WLAN-Router für die Kommunikation. Die DTS befindet sich in einem entkoppelten Aluminiumrahmen, auf dem sich zusätzlich eine Plastikwanne befindet, wie in Abbildung 8.12 abgebildet. Über

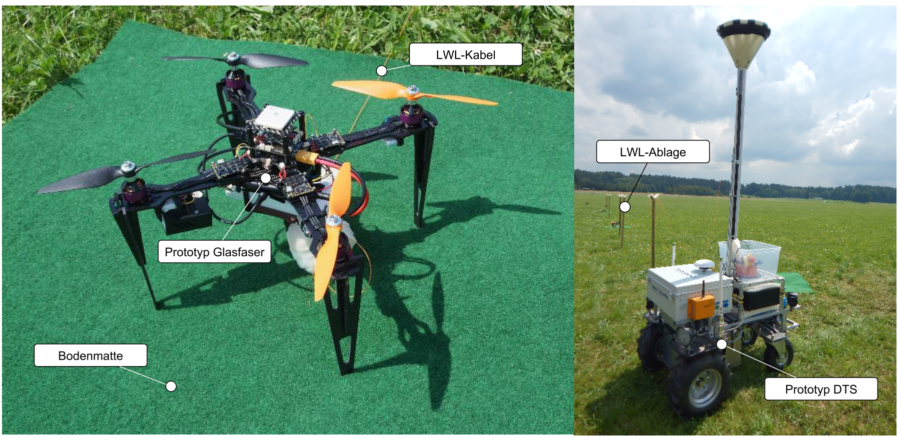

Abbildung 8.11. Aufnahmen des Experiments während der Messkampagne ScaleX. Der Prototyp *Glasfaser* startet auf einer Bodenmatte und das LWL-Kabel wird für den Start über die Ablagen zum Prototyp *DTS* geführt. (Die Fotos der Prototypen wurden von Stefanie Seubert, Universität Augsburg, aufgenommen.)

die aus Kunststoff bestehende LWL-Führung und den Auslass wird die Spindel bei einem Aufstieg abgewickelt. Zusätzlicher Schaumstoff in und an dem Auslass sowie der Führung verhindern Schleifpunkte für das sensible LWL-Kabel. Das LWL-Kabel sollte nicht unter Spannung stehen, weswegen auf eine automatische Aufwicklung verzichtet wurde. Stattdessen befindet sich der Auslass auf einer Höhe von ca. zwei Metern, um Schleifkontakte des LWL-Kabels mit dem Boden zu vermeiden. Der Hersteller *INNOK Robotics* liefert bereits ROS-Pakete für die Ansteuerung der Plattform aus.

Die Basis des Prototypen *Glasfaser* bildet das größere Outdoormodell mit einer montierten LWL-Kabelhalterung gefertigt aus Plastik und einer Schaumstofffütterung, wie in Abbildung 8.13 abgebildet. Die LWL-Halterung ist durch eine flexible Nylonkette und ein Kugellager an der Unterseite der Quadrocopterbasis montiert, damit auch bei einer Drehung das Kabel nicht reißt. Das aus carbonfaserverstärktem Kunststoff (CFK) gefräste Landegestell bietet genügend Abstand für eine sichere Landung. An dem Quadrocopter sind wie bei der mobilen Messstation an den Rotorstegen zwei Anschlüsse für modulare Sensoren angebracht. Diese sind allerdings nicht mit magnetischer Verschlusstechnik, sondern Molex Steckkontakten ausgestattet. Die modularen Temperatursensoren bestehen aus einem aus Kunstharz 3D-gedruckten Gehäuse, einem STM-Verteilerbord und zwei Temperatursensoren, die analog zu den Sensoren im Experiment der virtuellen Messtürme in einem Luftkanal angebracht sind. Das Ziel der modularen Temperatursensoren ist eine präzisere Messung der Temperatur als Referenzwert für die Messwerte des LWL-Kabels. Als gegenseitiges Kontrollelement befinden sich zwei TEMOD-Module mit PT1000 Widerstandsthermometern in jedem modularen Temperatursensor. Beide Module sind über den I2C-Bus an das STM-Verteilerboard angeschlossen, das die Anschlüsse eines STM32 ARM Prozessors herausführt. Die Grundidee des Verteilerboards ist in der stärkeren Prozessoreinheit und den sechs seriellen Schnittstellen (UART) begründet, von denen fünf einen Hardwareinterrupt unterstützen.

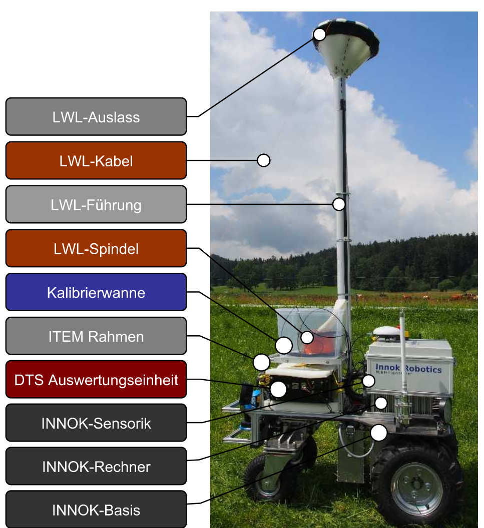

Abbildung 8.12. Aufnahme und Beschreibung des Prototypen *DTS* mit einer Kalibrierwanne, in der für den Versuch die DTS-Spindel gelagert ist. (Das Foto des Prototypen wurde von Stefanie Seubert, Universität Augsburg aufgenommen.)

Auf den modularen Temperatursensor ist bereits die erste Implementierung der Schnittstellen-API für einen Semantic Adapter in Kombination mit dem Self Descriptive Protocol (SDP) angewendet worden. So konnte die Kalibrierung in dynamischen Fähigkeiten gespeichert und direkt mit dem RDF-Graphen kombiniert werden. Für die Kalibrierung wurde ein Kühlschrank verwendet und ein SHT75-Sensor diente als Referenz für die lineare Regression. Die Schnittstelle zur Steuerung der Quadrocopter für die in JAVA programmierten Agenten bildet die Fähigkeit *Bewege*. Die Schnittstelle der ROS-Integration des Prototypen *DTS* bietet dieselbe Fähigkeit an, auch wenn eine virtuelle Fähigkeit für die Adaption verwendet wird. Der Ablauf des Experiments kann auch mit der Block Definition Language (BDL) durch eine im Dynamic Mission Definition and Execution (DMDE)-Framework erstellten Mission durchgeführt werden. Die Verwendung der Eigenschaften, um Fähigkeiten zu spezifizieren, ist jedoch speziell in der für Agenten entwickelten OOD-API möglich. Die Unterscheidung und Bewertung der Informationen ist eine Disziplin der Agentenschicht multipotenter Systeme und wird in den Publikationen [150, 153] behandelt.

Ergebnisse

Das Experiment hat innerhalb eines Tages stattgefunden und konnte zweimal erfolgreich durchgeführt werden. Die Messdaten wurden vor Ort kontrolliert und im Nachgang aufbereitet. Die Grafiken aus Abbildung 8.14 zeigen die Temperaturprofile der DTS-Auswertungseinheit, die eine proprietäre Software für den Datentransfer verwendet und die Messergebnisse in einer CSV-Datei mit Zeitstempel und Position der Messung am Kabel bereitstellt. So ist in den Grafiken der Verlauf einer Messung über die Zeit dargestellt, an der ein Aufstieg und die Vorwärtsbewegung stattgefunden haben. Für die Verifikation der Messdaten mit den kalibrierten, modularen Temperatursensoren und

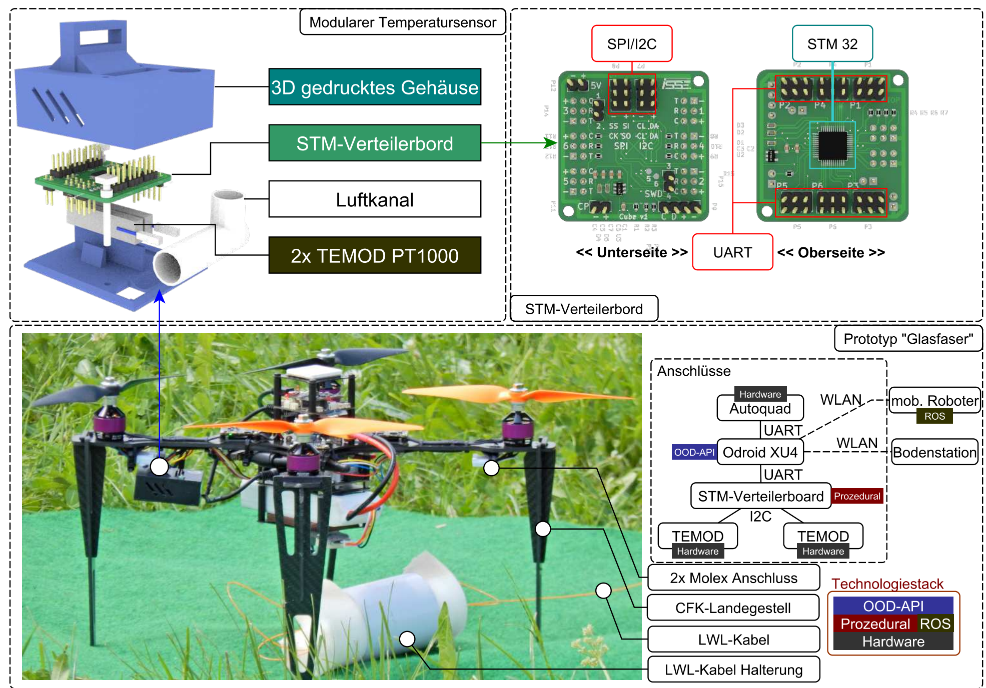

Abbildung 8.13. Die Aufnahme zeigt den Prototypen *Glasfaser*, der das LWL-Kabel in einer Halterung trägt. Als Referenzsensoren für die spätere Analyse finden sich zwei Steckplätze an den Rotorstegen für modulare Temperatursensoren. Das in den Temperatursensoren verwendete STM-Verteilerboard bietet sechs Anschlüsse für UART und jeweils einen SPI- und I2C-Anschluss. (Das Foto des Prototypen wurden von Stefanie Seubert, Universität Augsburg, aufgenommen. Renderings und Beschreibung sind Eigenwerke.)

den Flugdaten der Quadrocopter sollten die vermeintlich synchronisierten Zeitstempel verwendet werden. Das DTS hat nicht die Systemzeit des Rechners verwendet, was erst durch den Abgleich im Nachgang ersichtlich wurde. Zu diesem Zeitpunkt stand das DTS für eine Synchronisation nicht mehr zur Verfügung. So kann die in Abbildung 8.14 die y-Achse als relative Uhrzeit für einen Flug betrachtet werden, für eine konkrete Vergleichsanalyse zwischen den modularen Temperatursensoren und dem DTS ist allerdings eine Synchronisation der Systemzeiten unabdingbar, um Aussagen über die Genauigkeit zu treffen.

Der Kernaspekt des Versuchs ist die Illustration eines durchgängigen Ansatzes einer Ensembleanwendung, das Fähigkeiten und Eigenschaften als Entscheidungsgrundlage für die Prozessdefinition der Agenten verwendet. So konnte ein heterogener Schwarm mit fester Hardwarekonfiguration bereits auf dem Prinzip der Abstraktion von Fähigkeiten aufbauen und eine komplexe, neuartige Herausforderung im Bereich der meteorologischen Forschung bewältigen. Der verwendete Technologiestack aus Abbildung 8.13 besteht aus einer objektorientierten Schnittstelle für die Prozesse der Agenten (OOD-API), einer prozeduralen Implementierung der Sensoren sowie der Ansteuerung der Fähigkeiten über ROS für die Bodenstation, bzw. aus virtuellen Fähigkeiten für die Qua-

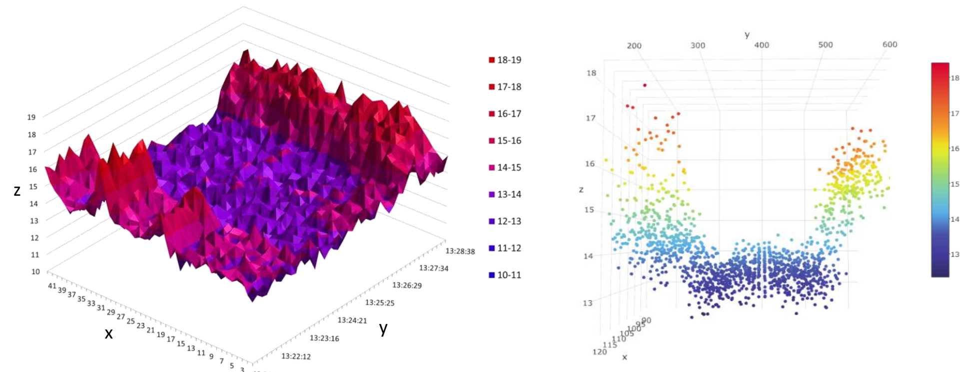

Abbildung 8.14. Die linke Grafik zeigt die Temperaturmessungen (z) der DTS Auswertungseinheit entlang des LWL-Kabels (x) im Verlauf der Zeit (y) während des Experiments. Die farbliche Markierung symbolisiert zusätzlich den Temperaturgradienten. Die rechte Abbildung illustriert den Aufstieg aus einer anderen Perspektive, in der die Temperaturkurve zwischen dem Start und der Landung der Mission visualisiert ist. (Die Grafiken sind der kooperativen Publikation [153] entnommen.)

drocopter. Die Ergebnisse des Experimentes sind in der Publikation [153] veröffentlicht. Der Versuchsaufbau und die Durchführung stehen als Video[4] zur Verfügung.

Ein negativer Aspekt des Experiments bildet der erhebliche Personalbedarf, so benötigt jeder Quadrocopter einen Sicherheitspiloten, der im Fall eines unerwarteten Verhaltens die Kontrolle übernehmen kann und sich durch die Einschränkungen des LWL-Kabels gut mit den anderen Piloten koordinieren muss. Aufbauen auf dieser Problemstellung ist eine neue Schwarmkontrollstation im Rahmen der Dissertation entwickelt worden, die im Folgenden genauer vorgestellt wird.

Schwarmkontrollstation

Bei Quadrocopterschwärmen wird oftmals die verwendete Anzahl an Drohnen als Vergleichsindex herangezogen, bspw. bei den *Intel Drone Shows*, die eine Anzahl von 1218 Drohnen für die Eröffnung der Olympischen Spiele 2018 in ihren Lichtshows verwendeten [159]. Die Anzahl kann nicht mehr mit einzelnen Sicherheitspiloten für jeden Quadrocopter selbst bewältigt werden und ein Verlass auf die Failsafestrategien der Drohne ist vorausgesetzt. Eine Publikation der ETH Zürich [306] thematisiert den Mangel allgemeiner Vorschriften sowie fehlender Standards in solchen Shows. In der vorangegangenen Entwicklung und Forschung werden hingegen Prototypen eingesetzt, deren Fehlerpotential deutlich höher liegen. Für die Reduktion der Sicherheitspiloten von Schwarmexperimenten in der Forschung mit zehn Quadrocoptern bietet Dousse et al. eine Erweiterung für die QGround Control an [68]. Farbliche Markierungen der Quadrocopter können in der für einzelne Quadrocopter ausgelegten Bodenstation markiert und kommandiert werden. Dabei wird keine separierte Schnittstelle zu den Quadrocoptern verwendet, sondern die Schwarminteraktionssoftware erweitert. Dieser Problemstellung widmet sich die Schwarmkontrollstation, die für das Experiment der

[4]https://www.youtube.com/watch?v=MWNyUymtNSs

Abbildung 8.15. Die linke Grafik zeigt die grafische Benutzeroberfläche der Schwarmkontrollstation auf einem im mittleren Bild dargestellten DGX-Funkgerät an einer 3D-gedruckten Adapterplatte. Rechts sind RGB-LEDs für ein visuelles Feedback an den Drohnen montiert (Die Fotos wurden von Martin Schörner, ISSE, aufgenommen.)

synchronen Glasfasermessung konzipiert wurde. Die Grundidee ist, direkt den Funkkanal der Fernbedienung zu verwenden und nicht innerhalb der Agentensoftware oder der BDL die Fehlerfälle zu behandeln, damit bei einem Softwarefehler immer noch reagiert werden kann. Die in Abbildung 8.15 abgebildete DGX-Funkfernbedienung bietet einen Port für einen Lehrer-/Schüler-Modus an, in welchem der Lehrer eine weitere Fernbedienung anstecken kann und bei Bedarf die Steuerung übernimmt. Dieser Port dient als Grundlage für die Schwarmkontrollstation, die auf einem Tablet mit einer 3D-gedruckten Halterung an die Funkfernbedienung angebracht ist. Insgesamt werden fünf Sicherheitsfunktionen bereitgestellt, die global auf alle Quadrocopter des Schwarms aber auch auf eine Teilmenge oder einzelne Individuen angewendet werden können.

Home
Die Home Position ist die Startposition des Quadrocopters, kann aber während jeder Mission neu definiert werden. Die Interaktion veranlasst den Quadrocopter keine Befehle mehr entgegenzunehmen und direkt zur Home Position zu fliegen. Der Befehl kann jederzeit durch andere Sicherheitsfunktionen unterbrochen werden.

Stay
Auch in diesem Fall wird die Kommunikation zur Prozesslogik unterbrochen und der Quadrocopter versucht die aktuelle Position zu halten. Besonders bei dem Experiment der synchronen Glasfasermessung ist eine globale Anwendung der *Stay* Funktion bei Fehlverhalten nützlich. Andere Sicherheitsfunktionen können danach ausgeführt werden, wie bspw. das Landen.

Land
Die Funktion *Land* veranlasst eine Landung an der aktuellen Position und kann ebenfalls von anderen Sicherheitsfunktionen unterbrochen werden.

Control
Die Fernbedienung ist an alle Quadrocopter gebunden, so kann ein einzelner Quadrocopter auch direkt übernommen werden. Wenn mehrere Quadrocopter selektiert sind, so gelten die Steuerbefehle der Fernbedienung für alle selektierten. In Formationsflügen, wie dem Experiment der *synchronen Glasfasermessung*, kann damit bspw. eine simultane Höhenkorrektur initiiert werden, um eine synchrone Landung zu ermöglichen.

Stop
Die Sicherheitsfunktionen *Stop* veranlasst einen sofortigen und irreversiblen Notstop durch Abschaltung aller Rotoren. Diese Funktion sollte nur im absoluten Notfall verwendet werden, da sie einen Absturz des Quadrocopters mit erheblichem Gefahrenpotential verursachen kann.

Für eine Zuordnung zwischen der grafischen Benutzeroberfläche und den selektierten Quadrocoptern werden neben farblichen Markierungen auch RGB-LEDs verwendet, die einen selektierten Quadrocopter aufleuchten lassen, bzw. bei der Ausführung einer Sicherheitsfunktion ein visuelles Feedback bieten. Die Schwarmkontrollstation ist in der Publikation [269] veröffentlicht und wurde sowohl indoor, als auch outdoor getestet.

Einordnung in die Fallstudie
In dem Experiment der *synchronen Glasfasermessung* dient der frühe, vertikale Prototyp von Semantic Plug & Play als Middleware zwischen der Hardware und einem Multiagentensystem. Die Schnittstelle beider Systeme bilden Fähigkeiten, die durch Eigenschaften beschrieben und auf realer Hardware ausgeführt werden. In der Fallstudie des Gasunfalls bildet diese Schnittstelle die Grundlage für alle Schwarmaktivitäten und auch die Vorbedingung für multipotente Systeme, die zur Laufzeit rekonfiguriert werden können.

Die smarten, modularen Temperatursensoren verwenden ein STM 32-Verteilerboard für die Kapselung der Temperatursensoren mit einem Stecksystem als Vorbereitung für die Realisierung multipotenter Systeme. Durch die Implementierung dynamischer Eigenschaften mit einer Persistierung der Kalibrierungskorrektur können gerade in den Szenarien der Feuerwehr schnelle Entscheidungen getroffen werden, ohne auf die Nachbereitung der Messdaten warten zu müssen.

Die Schwarmkontrollstation bildet ein wichtiges Instrument für die tatsächliche Verwendbarkeit der Fallstudie. So ist das Ziel, die Feuerwehr zu unterstützen und den Personalbedarf zu reduzieren, obsolet, wenn der Schwarm für jeden einzelnen Quadrocopter einen dedizierten Sicherheitspiloten benötigt. Das Experiment der horizontalen Glasfasermessung mit speziellen Anforderungen durch physische Abhängigkeiten des LWL-Kabels zwischen den Quadrocoptern und mangelnde Alternativen führten zur Eigenentwicklung der Schwarmkontrollstation.

Das letzte Outdoor-Experiment befasst sich mit der Ausbreitungsmodellierung des Gasunfalls, die durch den Schwarm eruiert werden soll und als *Observierung der Gaswolkenausbreitung* in Abbildung 3.2 der Fallstudie tituliert ist.

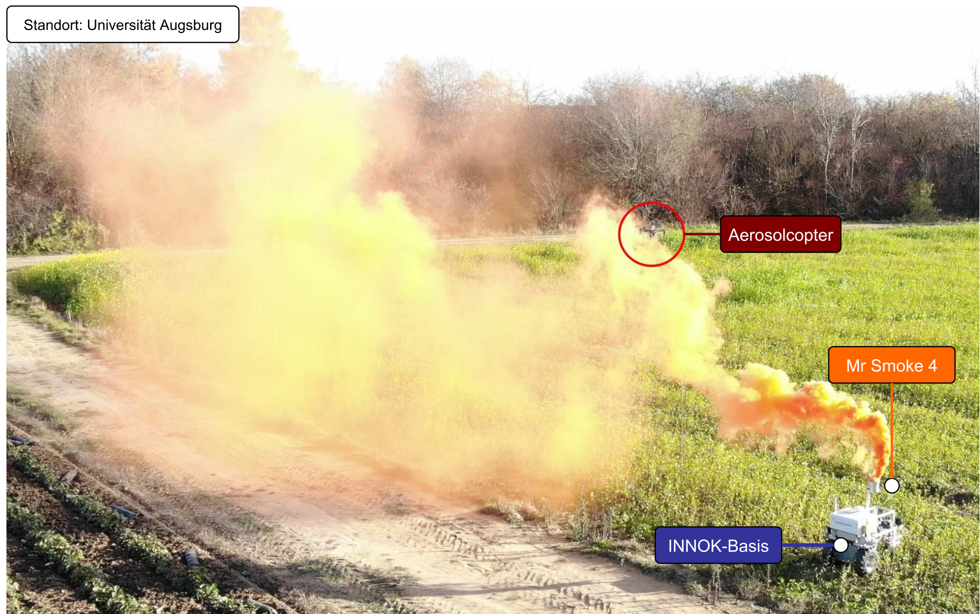

Abbildung 8.16. Feldexperiment in der Nähe der Universität Augsburg mit einem Rauchemitter (Mr Smoke 4) an einer mobilen Roboterplattform (INNOK Basis) und einem Quadrocopter mit Aerosolsensor (Aerosolcopter). (Eigenwerk)

8.4 Ausbreitungsmodellierung

Im Experiment der Ausbreitungsmodellierung wird untersucht, wie sich Aerosolwolken unter realen Wettereinflüssen ausbreiten. Für die Modelle werden in situ Messungen vorgenommen, um die Windrichtung und -stärke sowie die Konzentration der Aerosole zu messen. Als Aerosolemitter wird ein Rauchtopf der Marke *Mr Smoke 4* mit orange-farbenem Rauch benutzt und auf eine mobilen Roboterplattform angebracht, wie in Abbildung 8.16 illustriert. Das Suchfeld ist in der Software auf eine 20 mal 20 Meter große Fläche eingegrenzt, auf der in Absprache mit der lokalen Feuerwehr der Aerosolemitter gezündet wurde. Aufgrund der kurzen Brenndauer des Aerosolemitters von ca. 3 Minuten ist der Quadrocopter nicht automatisiert geflogen, sondern wurde von Hand gesteuert, um gezielt Messungen in der Aerosolwolke entnehmen zu können. Die Windrichtung und -stärke wurde anhand der Lage des Quadrocopters im schwebenden Zustand gemessen, ein Windsack in der Nähe des mobilen Bodenroboters dient als Referenz.

Prototypen

Für dieses Experiment ist der mobile Bodenroboter der Firma INNOK modifiziert worden. Ein aus Edelstahl gefertigter Topf dient als Gefäß für den Aerosolemitter, der an einem Aluminiumprofil in sicherem Abstand befestigt ist. Die INNOK Basisstation bietet ein GPS-Modul, um die Position des Emitters zu ermitteln, während der interne WLAN-Router die Infrastruktur für die Kommunikation zwischen der Bodenkontrollstation und dem Quadrocopter bereitstellt.

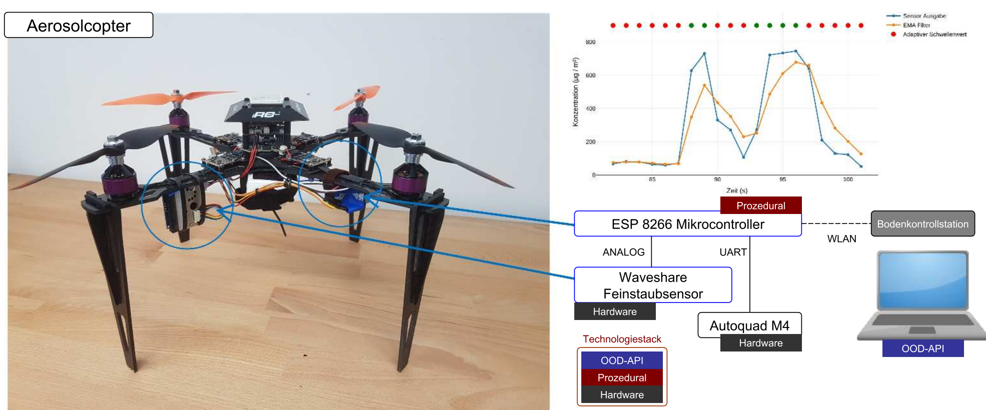

Abbildung 8.17. Aufbau des Aerosolcopter-Prototypen mit einem ESP 8266-Modul als Adapter für den Waveshare Aerosolsensor und den Autoquad M4. Das Bild rechts ist eine grafische Darstellung des Sensormesswertes zur Laufzeit mit einem adaptiven Schwellwert. (Das Foto stammt aus der betreuten Abschlussarbeit von Timo Peters, Student am ISSE)

Die Basis des in Abbildung 8.17 abgebildeten Prototypen mit dem Namen *Aerosolcopter* bildet die Outdoor-Variante des Forschungsquadrocopters. An einem Rotorausleger ist ein Aerosolsensor der Marke Waveshare angebracht, der lediglich über ein analoges Signal das Ergebnis einer optischen Messung von Aerosolpartikeln ausgibt. Eine exemplarische Messung ist in Abbildung 8.17 skizziert, in der die rohen Messwerte in blau angegeben sind und in orange ein *Exponential Moving Average* (EMA) als adaptiver Schwellenwert genutzt wird, um binäre Aussagen (rote und grüne Punkte) ohne Konstante zu treffen [144]. Der Aerosolsensor ist an ein ESP8266-Modul angebracht, das an einen weiteren Rotorausleger montiert ist. Als Vorbereitung für eine Kapselung in einen modularen Sensor ist das ESP-Modul des Quadrocopters nicht verwendet worden. Neben dem Aerosolsensor ist auch der Autoquad M4 des Quadrocopters an das ESP-Modul über UART verbunden, um neben den Steuerbefehlen auch die Sensorwerte des Lagesensors zu übermitteln und dem virtuellen Anemometer bereitzustellen. Auf dem ESP-Modul befindet sich die prozedurale Implementierung der Schnittstellen-API des Semantic Adapaters. Die Fähigkeiten und Eigenschaften werden über WLAN an die Bodenkontrollstation gesendet, auf der sich auch die virtuelle Fähigkeit des Anemometers befindet. So werden die Telemetriedaten des Quadrocopters an das ESP-Modul gesendet, auf dem sich der Adapter für die Fähigkeit des Lagesensors befindet. Diese Fähigkeit wird über WLAN von dem virtuellen Anemometer genutzt, um die Windrichtung und -stärke zu bestimmen. Das virtuelle Anemometer nutzt ebenfalls die Fähigkeit *Fliege zu Position*, die mit der aktuellen Position des Quadrocopters parametrisiert wird, und veranlasst ein *Verweilen* an der aktuellen Position für mehrere Sekunden. Windrichtung und Stärke werden in der Zeit des *Verweilens* gesammelt und es wird ein Richtungsmittelwert durch zwei Einheitsvektoren gebildet. Für Schwankungen während der Messungen wird die zirkulare Varianz verwendet [189]. Die in Abschnitt 5.1.1 vorgestellten Algorithmen für die Lokalisierung der Quellen (Partikelfilter) und für die Vorhersage der Ausbreitung (Kernel DM) verwenden neben der Fähigkeit des virtuellen Anemometers auch die Qua-

drocopterposition und die Messwerte des Aerosolsensors und werden ebenfalls auf der Bodenkontrollstation ausgeführt. Die grafische Benutzeroberfläche ist in Abbildung 8.18 abgebildet und mit dem Plotly Dash Framework realisiert.

Ergebnisse

Vor der Zündung des Rauchemitters wurden Windmessungen vorgenommen, die deutliche Unregelmäßigkeit demonstrierten, was auch der Windsack bestätigte. So konnten stellenweise Änderungen der Windrichtungen von 180 Grad beobachtet werden, wie in Abbildung 8.18 illustriert, wobei die Gesamtübersicht links den Versuchsanfang beschreibt und die Windmessung rechts das Versuchsende aufzeigt. Während des Versuchs herrschte hingegen nahezu Windstille. Trotz der unregelmäßigen Verhältnisse konnte das Verfahren, die Windrichtung anhand der Quadrocopterneigung zu bestimmen, erfolgreich anwendet werden. Die Unstetigkeiten verdeutlichen indes, dass verwertbare Ergebnisse nicht unmittelbar nach dem Beginn eines Standfluges errechnet werden können. Das Verfahren auf Basis der Quadrocopterneigung gewinnt seine Genauigkeit erst durch Durchschnittsbildung über einige Messperioden. In den Versuchen konnte meist bereits nach etwa fünf Sekunden ein stabiles Ergebnis erzielt werden, sofern der Wind innerhalb der Messzeit annähernd konstant blieb. In Abbildung 8.18 ist das Anemometer zu Beginn des Experiments (A) zu sehen, mit einem Ausschlag gegen Osten, der im Laufe des Experiments eine hohe zirkuläre Varianz aufzeigt (AL). Gegen Ende des Experiments dreht sich der Wind gegen Westen, wobei der blaue Kegel die aktuelle Messung angibt. Die Konzentrationsmessung des Sensors ist analog zu Abbildung 8.17 ebenfalls in der grafischen Benutzeroberfläche in Abbildung 8.18 unter dem Punkt (M) visualisiert. Die Messungen durch eine visuelle Aerosolwolke in Abbildung 8.16 und die Messergebnisse offenbaren den Einfluss des Luftstroms, der vom Quadrocopter erzeugt wird. So wird die Wolke praktisch nach unten gedrückt und der Sensor verzeichnet keine Aerosole. Erst wenn sich verwirbelte Partikel oberhalb befinden, kann ein Ausschlag des Sensors registriert werden. Der Thematik des *Downwash* auf die Messergebnisse widmen sich im Rahmen von Gasmessungen auf einem Quadrocopter unterschiedliche Publikationen [100, 161], in denen punktuelle Messungenauigkeiten evaluiert werden. Diese durch die Bewegung der Drohne induzierte Verbreitung der Rauchwolke wirkt sich auch auf das Ausbreitungsmodell (CL) aus, das in Abbildung 8.18 visualisiert ist. Das rote X in Abbildung 8.18 (CL) illustriert die Aerosolquelle, wobei am Versuchsende der Konzentrationsdurchschnitt südöstlich der Quelle am höchsten ist. In der Nähe der Quelle dagegen war die Rauchwolke, wie in Abbildung 8.16 ersichtlich, kleiner und stärker konzentriert, jedoch durch den Einfluss des Downwash nicht messbar. Dennoch erkennt der Kernel DM-Algorithmus die durch Wind beeinflusste Konzentrationsverteilung, wohingegen der Partikelfilter für das Auffinden der Gaswolke zuständig ist. Die Parametrisierung des Partikelfilteralgorithmus' ist essentiell für eine Lokalisierung, wie in Abbildung 8.18 illustriert. Das Suchareal in (BL) verwendet eine sehr hohe Varianz, was dazu führt, dass praktisch nur die letzten getätigten Messungen das Ergebnis beeinflussen. In (B) hingegen ist die Adaptivität deutlich höher angesetzt, so kann zumindest ein Ausschluss der Gasquelle im östlichen Teil der Karte angenommen werden, wobei die kurze Messzeit von 3 Minuten nicht für die Lokalisierung der Quelle gereicht hat.

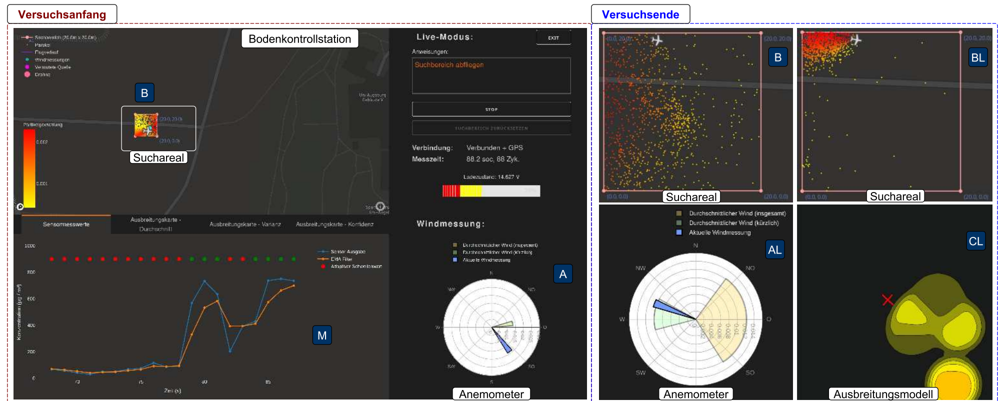

Abbildung 8.18. Screenshot der grafischen Benutzeroberfläche für die Windmessung (A, AL), Ausbreitungsmodellierung (CL), Gasquellenlokalisierung (B, BL) mit unterschiedlicher Parametrisierung, und Messwerte (M). Die verwendeten Parameter der Gasquellenlokalisierung:
(B): Adaptilität: 0,05; Gradient: 1 für kein Aerosol entdeckt
(BL): Adaptilität: 0,4; Gradient: 10 für kein Aerosol entdeckt
(Die Benutzeroberfläche stammt aus der betreuten Abschlussarbeit von Timo Peters, Student am ISSE)

Einordnung in die Fallstudie

Sowohl der Partikelfilter als auch der Kernel DM-Algorithmus unterstützen simultane Messungen, die bspw. durch einen Quadrocopterschwarm getätigt werden. Die durch Semantic Plug & Play bereitgestellten Fähigkeiten und Eigenschaften dienen dabei nicht nur der Automatisierung einer Mission innerhalb eines Agentenframeworks, sondern auch als Evaluations- und Parametrisierungswerkzeug. Grafische Visualisierungen helfen der Interpretation gemessener Werte und lassen sich durch die Adaptivität von Semantic Plug & Play auch auf andere Sensoren anwenden. So dient die Benutzeroberfläche nicht nur dem konkreten Experiment, sondern kann auch in den Phasen der Fallstudie eingesetzt werden, um die korrekte Parametrisierung der Algorithmen zu evaluieren. Durch die Möglichkeit, den Semantic Controller und die OOD-API auf einem Gerät, der Bodenkontrollstation, zu verteilen, können sehr prototypische Aufbauten für die Evaluation verwendet werden. Ebenso bietet eine grafische Benutzeroberfläche mit dynamischen Elementen eine Einsicht in die aktuelle Konfiguration. In den Outdoor-Experimenten wurden bislang keine Rekonfigurationen demonstriert, die das Potential der Semantic Plug & Play -Prinzipien untermauern. Die folgenden Indoorszenarien werden eben diese Aspekte, bspw. durch eine Ersetzbarkeit im sensorbasierten Flug, einer Rekonfiguration im Pick & Place-Szenario oder innerhalb der modularen Simulation im GoLive Experiment fokussieren.

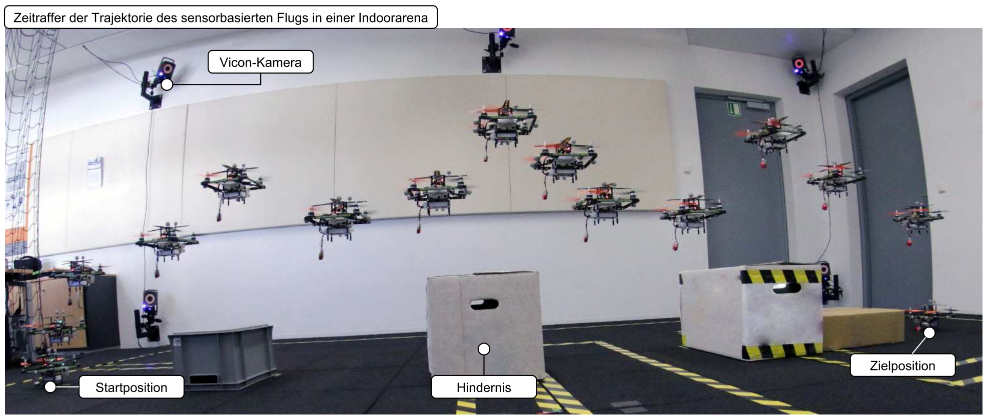

Abbildung 8.19. Zeitraffer des sensorbasierten Fluges in einer Indoor-Arena mit dem Vicon-Trackingsystem. Die Hindernisse bestehen aus Kisten. (Das Foto stammt aus der Abschlussarbeit von Christian Eymüller, ISSE)

8.5 Der sensorbasierte Flug

Der sensorbasierte Flug ist das erste Experiment, das indoor stattgefunden hat. Die in Abbildung 8.19 illustrierte Flugarena basiert nicht auf einer Traversenkonstruktion, sondern ist mit Wandhalterungen realisiert und umfasst zehn Vicon-Kameras. Das Konzept, der Aufbau und die Realisierung dieses Vorgängers der großen Flugarena ist ebenfalls im Rahmen der Dissertation entstanden. Der Versuchsaufbau beginnt mit einer Trajektorie, in der ein Quadrocopter an der Startposition startet und auf einen halben Meter Höhe über Boden steigt. Anschließend beginnt der sensorbasierte Flug zur Zielposition, wobei der Abstand zum Boden von einem halben Meter eingehalten werden soll. Abbildung 8.19 zeigt einen Zeitraffer eines Quadrocopters während des sensorbasierten Fluges. In diesem Experiment werden speziell drei Aspekte der Semantic Plug & Play-Architektur untersucht:

Sensortausch gleicher Klasse
Eine Stärke der Semantic Plug & Play Architekur bildet die Austauschbarkeit der Sensoren. Zur Veranschaulichung werden zwei unterschiedliche Sensoren verwendet, die beide eine relative Distanz messen können und anhand ihrer Eigenschaften beschrieben sind. Die Fähigkeit *Messen* verwendet die Sensoreigenschaften, um die Geschwindigkeit der Fähigkeit *Fliege zu Position* zu parametrisieren. Dieser Prozess wird als eigene Fähigkeit gekapselt, der *Gradientenflug*.

Adaption einer Fähigkeit
Die Fähigkeit des *Gradientenflugs* muss nicht zwangsweise einen Distanzsensor als Grundlage verwenden, sondern kann auch auf andere Sensoren mit der Aktuatorik eines Quadrocopters reagieren. Für die meteorologischen Experimente der Grenzschichtforschung kann bspw. ein Quadrocopter mit Temperatursensor aufsteigen, solange die Temperatur ansteigt oder abfällt. Bei einer Grenzschicht wird der Gradient invertiert und der Quadrocopter kann durch ein einpendeln

des lokalen maximalen oder minimalen Temperaturwertes eine Grenzschicht identifizieren. Für diesen Versuch werden ebenfalls zwei Temperatursensoren mit Eigenschaften beschrieben und anhand der Fähigkeit des *Gradientenflugs* getestet.

Auswirkung externer Eigenschaften
Neben den Sensoreigenschaften werden auch Eigenschaften systemfremder Gegenstände betrachtet, die sich auf die Fähigkeit *Fliege zu Position* und damit auch auf den *Gradientenflug* auswirken. So wird der Quadrocopter mit unterschiedlichen Gewichten ausgestattet und die Flugzeit abhängig des Stromverbrauchs der Rotoren estimiert.

Für die Konfiguration der Quadrocopter werden folgend die Prototypen der Sensoren und der Quadrocopterbasis vorgestellt.

Prototypen
Die Basis des Prototypen *Modular* baut auf der kleineren Indoorvariante der Forschungsquadrocopter auf und ist mit einem Einplatinencomputer unterhalb einer Montageplatte bestückt. Die Montageplatte aus Abbildung 8.20 verwendet, wie die Basis der Sensoren, eine magnetische Halterung, damit ein schneller Tausch der Sensoren möglich ist. Durch vier Neodymmagnete an den Seiten werden alle Bauteile zusammengefügt, wobei die Anordnung immer identisch und ein Stapeln mehrerer Bausteine möglich ist. Die unterschiedlichen Sensoren sind auf Adapterplatten montiert, die eben dieses Prinzip adaptieren. Sowohl die Montageplatte als auch das Gehäuse der Basis und die Adapterplatten sind im Rahmen der Dissertation entwickelt und mit einem Stereolithographiedrucker mit Kunstharz (SLA) gedruckt worden. Die Basis verwendet den Einplatinencomputer *Orange Pi Zero*, der einen ARM Cortex A7 mit 512MB DDR RAM und einem SD-Kartenslot bereitstellt. Für die Kommunikation bietet der Einplatinencomputer dreizehn PIN IOs, die I2C, UART und SPI unterstützen, sowie WLAN, das für die Kommunikation mit dem Odroid XU4 verwendet wird. Eine Übersicht der Sensoreigenschaften wird in der folgenden Tabelle gegeben:

Sensor	SR 04[160]	Parallax LRF[250]	TEMOD-PT1000[98]	SHT75[271]
Sensortyp	Ultraschall	Laser	Temperatur	Temperatur & Luftfeuchte
Messbereich	20-3000mm	150-2400mm	-32° – 96° C	-40° – 124° C
Genauigkeit	± 3mm	± 8mm	± 0,15° C	± 0,3° C
Frequenz	50 Hz	5 Hz (tuned)	-	-
Schnittstelle	GPIO	UART	I2C	I2C
Gewicht	86g	92g	76g	70g

Die Frequenz der Temperatursensoren ist bewusst ausgelassen, da die Angabe zu einer falschen Schlussfolgerung führt. So bezieht sich die Frequenz der Distanzsensoren auf die Messfrequenz, bei einem Temperatursensor hingegen auf die Abtastfrequenz. Der Parallax Laser Range Finder (LRF) bietet üblicherweise ein Messfrequenz von 1 Hz, die für das Experiment durch ein Übertakten der Sensorplatine auf 5 Hz erhöht ist.

Auf dem Odroid XU4 wird sowohl der Semantic Controller, als auch eine ältere Variante der OOD-API verteilt, die auf der Robotics API [8] aufbaut. Die Orange Pi Zeros hingegen dienen als Plattform für den Semantic Adapter, auf dem die Hardware angeschlossen wird, die RDF-Graphen verteilt sind und die Fähigkeiten der Hardware wie *Messen* oder *Fliege zu Position* bereitgestellt werden. Die virtuelle Fähigkeit des *Gradientenflugs* wird

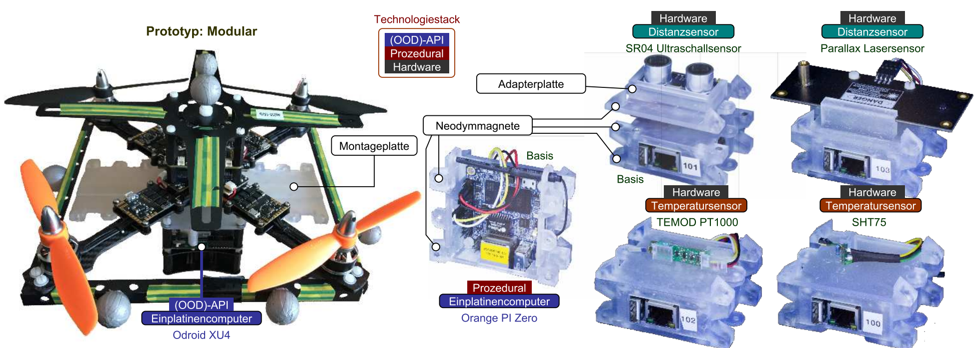

Abbildung 8.20. Abbildung der modularen Sensorik für den Prototypen *Modular*, der eine Montageplatte für zwei Sensoren bietet. Als Basis für die Sensoren wird ein Einplatinencomputer verwendet, auf dem die vier unterschiedlichen Sensoren angebracht und somit die Selbstbeschreibungen lokal an den Sensoren verfügbar gemacht werden können. (Die Fotos sind von Christian Eymüller im Rahmen einer Abschlussarbeit aufgenommen worden, ISSE)

hingegen auf dem Semantic Controller mit den abhängigen Fähigkeiten assoziiert. Innerhalb des *Gradientenflugs* werden erstmals Anforderungen an die abhängigen Fähigkeiten gestellt, die durch Eigenschaften ausgedrückt werden. So bestimmt die Frequenz der Sensoren die maximale Geschwindigkeit des Flugs oder das Gewicht aller Sensoren und Zuladungen wird für die Abschätzung der Flugzeit verwendet.

Ergebnisse

Die Fähigkeit *Fliege zu Position* ist so umgesetzt, dass sie asynchron ausgeführt wird und die Zielposition jederzeit überschrieben werden kann. Der Gradientenflug nutzt diese Fähigkeit und implementiert einen Fehlervektor, der auf die Messwerte der Distanzsensoren das Ziel entsprechend aufmoduliert. In Abbildung 8.21 ist die Auswertung des Gradientenflugs mit einem SR04 Ultraschallsensor anhand des Vicon-Systems zu sehen. Der Graph zeigt den Abstand des Quadrocopters zum Boden während des Abfliegens der Trajektorie zu insgesamt sieben Versuchsdurchläufen. Das Ergebnis zeigt, dass die Hindernisse mit einer geringen Verzögerung erkannt werden, der Quadrocopter auf die Hindernisse reagiert und die Höhe entsprechend anpasst. Die Verzögerung nach einem Hindernis lässt sich durch den Sensor selbst erklären, der einen Abstrahlwinkel des Ultraschallsignals von ungefähr 15° aufweist [271]. Das Experiment mit dem Parallax LRF ist gescheitert, da die Vibrationen des Quadrocopters die Messwerte des gerichteten Sensors mit sehr geringem Abstrahlwinkel negativ beeinflusst hat. Trotz Erhöhung der Messfrequenz auf 5 Hz konnten keine verwertbaren Messungen während eines Flugs aufgezeichnet werden.

Für die Adaption des Gradientenflugs auf die Temperatur wurde der Messbereich auf 10-80° eingegrenzt und die Höhe des Quadrocopters mit der folgenden Formel in Zentimetern bestimmt:

$$height = (temperature - 10) * 2,43 + 30 \tag{8.1}$$

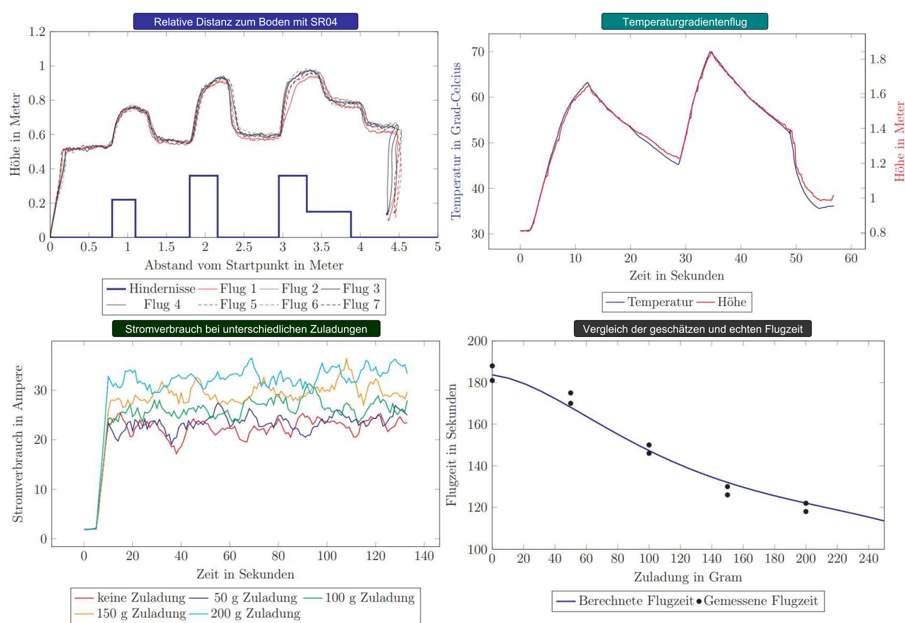

Abbildung 8.21. Die Graphen zeigen die Auswertungen der Versuche des Gradientenflugs, sowie die Abschätzung der Flugzeit. Für die Positionsdaten wurde das Vicon-Trackingsystem verwendet. Für die Messung des Stromverbrauchs befindet sich ein spezieller SMD IC-Baustein auf der Powerplatine des Forschungsquadrocopters. (Die Auswertungen der Versuche stammen aus der betreuten Abschlussarbeit von Christian Eymüller, ISSE)

Da die modularen Sensoren über WLAN mit dem Odroid XU4 kommunizieren, ist die lokale Abhängigkeit zwischen dem Quadrocopter und den Sensoren nicht gegeben. So sind für die Versuche die Sensoren getrennt von dem Quadrocopter mit einem haushaltsüblichen Föhn erhitzt worden, während der Quadrocopter eine Position in der Flugarena hält und entsprechend der Messdaten die Höhe variiert. In Abbildung 8.21 ist die Korrelation zwischen der gemessenen Temperatur und der Flughöhe, während zwei Erhitzungsvorgängen und einer aktiven Ventilation nach 48 Sekunden dargestellt. Der Vergleich zwischen dem TEMOD-PT1000 und dem SHT75-Sensor offenbart keine signifikanten Unterschiede, da sich zwar die Abtastfrequenzen unterscheiden, die Messfrequenz beider Sensoren hingegen ähnlich träge verläuft. In diesen Versuchen ist allerdings der Fokus nicht auf die Auswertungen gelegt, sondern vielmehr auf die Wiederverwendbarkeit der Fähigkeit, die durch unterschiedliche Eigenschaften jeweils geeignete Ergebnisse erzielt. So kann der Gradientenflug für die Suche nach Grenzschichten verwendet werden, sofern ein Sensor angeschlossen ist, der für die Fähigkeit *Messen* die Eigenschaft *Temperatur* verwendet und anstatt absoluter Werte die Differenz zur Zeit betrachtet. Der zugrundeliegende Regelalgorithmus des *Gradientenflugs* entscheidet anhand der Eigenschaft *Temperatur* oder *Distanz*, welche Berechnungsgrundlage und Parameter verwendet werden. So wirken sich Frequenzeigenschaften nur bei Distanzsensoren auf die maximale Geschwindigkeit der Fähigkeit *Fliege zu Position* aus, während diese bei einem Temperaturgradienten vernachlässigt werden können.

Eine besonders komplexe Einschränkung für die Fähigkeiten ist die Abschätzung der Flugzeit, die in Abhängigkeit des Akkumulators und des Gewichts der Zuladung berechnet wird. Ähnlich dem Verbrauch eines Autos wird auch die Flugzeit eines Quadrocopters durch das individuelle Verhalten des Piloten beeinflusst. Um eine grobe Abschätzung der Flugzeit zu gewinnen, wird das Experiment des Gradientenflugs verwendet und mit unterschiedlichen Zuladungen die Abhängigkeit zum Stromverbrauch gemessen. In Abbildung 8.21 ist die Kurve des Stromverbrauchs in dem Experiment mit in 50g gestaffelten Zuladungen skizziert. Die Limitierung wird bei 220 Gramm gesetzt, so dass die Flugfähigkeit nicht mehr ausgeführt werden kann, wenn die Zuladung zu groß ist. Insgesamt haben sich folgende Durchschnittswerte bei den Messungen ergeben:

Zuladung in Gramm	0	50	100	150	200
∅-Stromverbrauch in A	19,57	21,31	24,21	28,00	29,27

Durch die Approximation ergibt sich die Kurve aus Abbildung 8.21 für die Berechnung der Flugzeit, die mit mehreren Referenzmessungen je Gewicht geprüft wurde. Das Minimum und Maximum der Referenzmessungen ist mit den Punkten in der Kurve skizziert und beträgt maximal knapp 5%. In den Experimenten ist ein 1800mAh Lithium Polymer Akkumulator mit 3 seriellen Zellen eingesetzt worden. Über die Selbstbeschreibung der modularen Sensoren können die konkreten Gewichte aller Elemente abgefragt, gesammelt und für die Berechnung der Flugzeit verwendet werden. Zwar sind die individuelle Mission und der konkrete Akku nicht in die Berechnungen eingeflossen, dennoch konnte ein signifikanter Unterschied in der Flugdauer in Abhängigkeit zum Gewicht nachgewiesen werden. Die Ergebnisse des Experiments sind ebenfalls in den Publikationen [83, 309] veröffentlicht.

Einordnung in die Fallstudie
Im Rahmen der Fallstudie werden multipotente Systeme eingesetzt, also rekonfigurierbare Quadrocopter, die mit unterschiedlicher Sensorik und Aktuatorik ausgestattet sind. Das Experiment stellt eine Plattform für modulare Sensoren vor und untersucht drei Aspekte, die für eine Rekonfiguration benötigt werden.

Sensortausch gleicher Klasse
Das Experiment hat gezeigt, dass ein Tausch eines Sensors derselben Klasse sehr stark von den Eigenschaften abhängen kann, die nicht in Datenblättern zu finden sind. So ist der Versuch mit einem Laserdistanzsensor gescheitert, da die Vibrationen eines Quadrocopters während des Flugs einen erheblichen Einfluss auf die Messwerte hatte. Im Fall der Temperatursensoren hingegen hatten die Eigenschaften der Dokumentation keinen Einfluss auf das Ergebnis.

In der Fallstudie aus Abbildung 3.2 wird ein universeller Gassensor eingesetzt, um eine Typenbestimmung zu ermöglichen. Ist der Typ gefunden, so wird eine Rekonfiguration eingeleitet und Sensoren des konkreten Typs werden eingesetzt. In diesem Fall soll die Algorithmik zur Beobachtung der Gaswolkenausbreitung nicht von dem konkreten Typ abhängig sein, weswegen eine Untersuchung der Austauschbarkeit von Sensoren der gleichen Klasse obligatorisch ist. Semantic Plug & Play bietet die Möglichkeit, die Dokumentation der Sensoren anhand von Eigenschaften in der Algorithmik zu verwenden.

Adaption einer Fähigkeit
Eigenschaften können nicht nur die Fähigkeiten beschreiben und beschränken, sondern auch für eine Wiederverwendbarkeit eines Algorithmus' verwendet werden. So kann der Gradientenflug nicht nur für eine relative Distanz zum Boden verwendet werden, sondern auch als virtueller Messturm, um Grenzschichten zu orten, wenn sich der Richtungsvektor der Z-Achse an Temperaturdifferenzen orientiert.

Suchalgorithmen oder Observierungen verwenden häufig eine ähnliche Algorithmik, die auf spezielle Sensoren angepasst wird. So wird für die Gasquellenlokalisierung im Experiment der *Ausbreitungsmodellierung* der Partikel Filter-Algorithmus angepasst, der auch für die Lokalisierung mobiler Roboter verwendet wird [58]. Über Eigenschaften können die nötigen Varianten identifiziert und ausgeführt werden.

Auswirkung externer Eigenschaften
Nicht nur die Eigenschaften der Fähigkeiten beeinflussen deren Ausführung, sondern auch Eigenschaften, die das Gesamtsystem betreffen. So beeinflusst das Gewicht des modularen Quadrocopters in Kombination mit den bestückten Sensoren die Flugzeit.

Die Traglast modularer Quadrocopter ist für die Fallstudie essentiell, da in jeder Konfiguration mindestens ein externer Sensor verwendet wird. Eine Berechnung der Flugzeit ist als präventive Schutzmaßnahme für die Planung erforderlich, um eine reaktive Notlandung bei niedrigem Akkustand zu vermeiden.

Semantic Plug & Play bietet eine Plattform für multipotente Systeme, in der Hardware nicht mehr individuell implementiert werden muss. Die Eigenheiten einzelner Hardwareelemente können durch Eigenschaften ausgedrückt und maschinenlesbar durch Anforderungen in den Algorithmen verwendet werden. Das Experiment des sensorbasierten Flugs verdeutlicht jedoch, dass die Dokumentation der Hersteller in manchen Fällen nicht ausreicht, sondern Testverfahren nötig sind. Die aus den Testverfahren gewonnenen Erkenntnisse können wieder als Eigenschaft eingetragen und der RDF-Graph erweitert werden, ohne die Sensorimplementierung zu ändern. Die Granularität ist durch die Ontologien frei wählbar. So kann die Vibrationsproblematik über die Reaktionszeit und den Ausfallwinkel des Sensors festgelegt werden, oder einfach nur ein boolescher Wert auf die Problematik hinweisen. Eben dieses Szenario spiegelt die intendierte Domänenabhängigkeit wieder, die mit der MDDM-Architektur aufgegriffen wird. Die Anforderungen der Domäne bestimmen die benötigten Eigenschaften für die Ausführung von Fähigkeiten. Werden diese in nicht ausreichendem Maße bereitgestellt, so dienen die Semantic Web-Mechanismen der Erweiterung der Wissensbasis. Dadurch profitieren nicht nur die multipotenten Systeme, sondern generell wird eine Dokumentation ermöglicht, die als Selbstbeschreibung eine fortlaufende Präzisierung unterstützt. In diesem Experiment können die Sensoren durch ein einfaches Stecksystem rekonfiguriert werden. Der Versuch einer Rekonfiguration des Systems *zur Laufzeit* bildet überdies die Grundlage für das folgende Experiment.

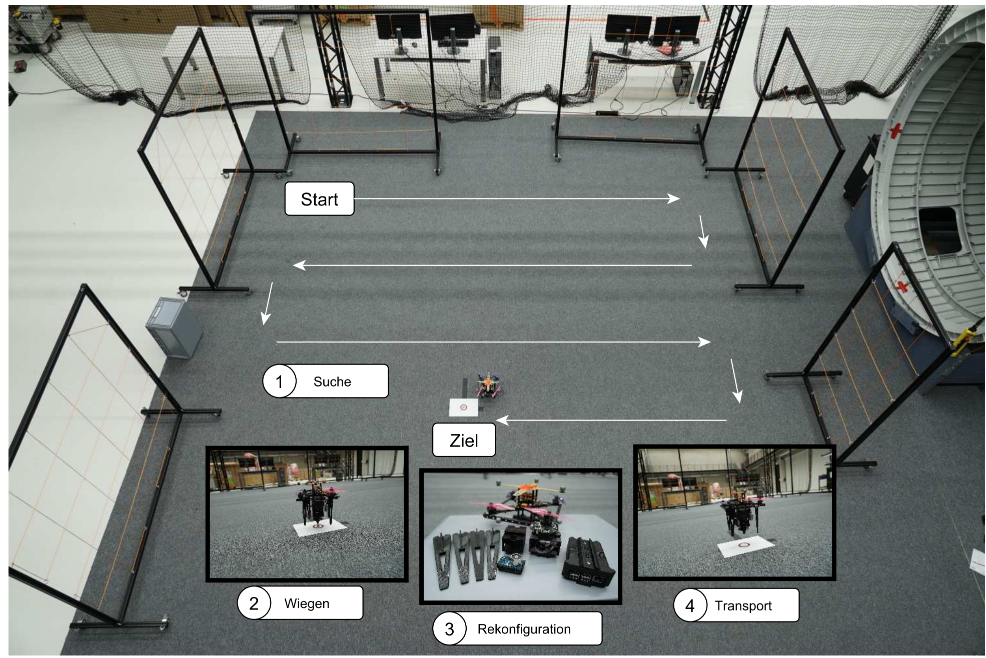

Abbildung 8.22. Versuchsaufbau in der großen Flugarena mit Vicon-System. Das Experiment ist in vier Abschnitte unterteilt, die sequentiell ausgeführt werden. (Eigenwerk)

8.6 Pick & Place

Für die Durchführung des Experiments wird ein abgesteckter Teilbereich der großen Indoor-Flugarena verwendet, wie in Abbildung 8.22 dargestellt. Das Vicon-Trackingsystem wird analog zum Experiment des sensorbasierten Flugs für die Positionssteuerung des Quadrocopters verwendet. Das Experiment wird in vier sequentiell ablaufende Phasen unterteilt:

1. **Suche**
   In der Suchphase wird ein Teilbereich der Flugarena abgeflogen und ein auf Papier ausgedruckte Farbmuster gesucht (siehe Abbildung 8.22). Dafür wird eine spezielle Kamera (Pixy2) eingesetzt, die Farbkodierungen erkennen und einer 2D-Position relativ zur Kamera zuordnen kann [2]. In der Mitte der Farbkodierung befindet sich das zu transportierende Werkstück, das mit einem Elektromagneten am Quadrocopter transportiert werden soll.
2. **Wiegen**
   Vor dem Transport wird das Werkstück gewogen, um eine mögliche Überlast zu identifizieren. Dabei werden die Ergebnisse aus dem Experiment des *sensorbasierten Flugs* verwendet, um eine Gewichtsabschätzung über den Verbrauch abzuleiten, die in Abbildung 8.21 skizziert ist. Bei Feststellung der Überlast wird die Position des Quadrocopters über dem Ziel in einer dynamischen Eigenschaft gespeichert und die Kamera kann demontiert werden, um die Nutzlast zu erhöhen.

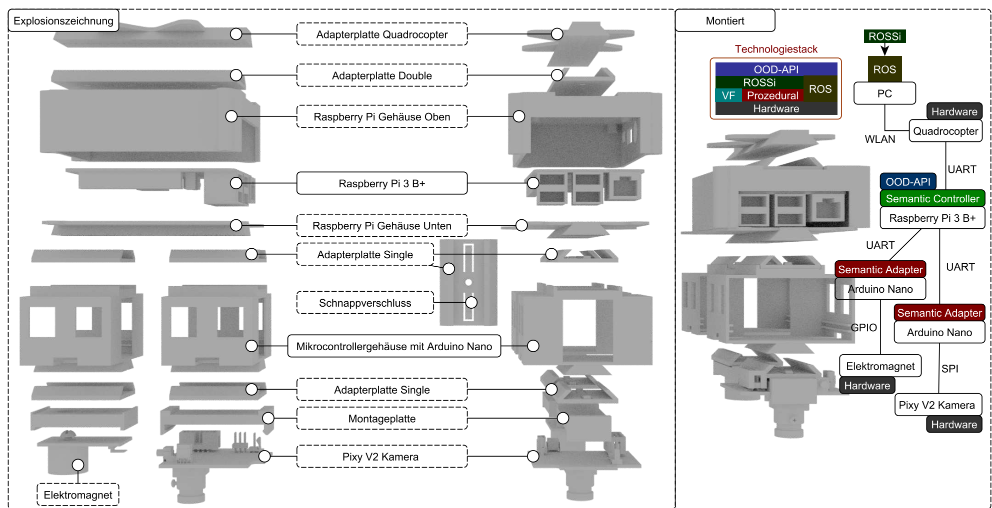

Abbildung 8.23. Abbildung des Montagesystems und aller im Experiment verwendeter modularer Bauteile in einer Explosionszeichnung. Die montierte Variante, wie sie auch in Abbildung 8.22 in natura abgebildet ist, befindet sich rechts im Bild. Der verwendete Technologiestack ist nicht nur auf die modularen Bauteile angewendet sondern impliziert auch das Trackingsystem, den Quadrocopter und einen PC. (Eigenwerk)

3. **Rekonfiguration**
   Sowohl der Elektromagnet als auch die Kamera sind in modular und können demontiert werden, wie in Abbildung 8.22 illustriert. Die Anwendung muss dafür nicht neu gestartet werden, womit innerhalb der Prozessdefinition eine Rekonfiguration realisiert werden kann. So fliegt der Quadrocopter zurück zur Startposition, landet und wartet, bis die Kamera demontiert ist.

4. **Transport**
   Nach erfolgter Rekonfiguration wird der Transport eingeleitet und der Quadrocopter fliegt von der Startposition zur gespeicherten Zielposition, um das Werkstück aufzuheben und an die Startposition zu transportieren.

Für dieses Experiment ist eine Neuentwicklung der modularen Plattform entstanden, die auf einen Kraft- und Formschlussmechanismus aufbaut.

Prototypen

Der in diesem Experiment entwickelte Prototyp umfasst ein Konzept für modulare Bauteile mit standardisierter Hardware wie dem Raspberry Pi oder einem Arduino Nano. Dabei sind die Module generisch gehalten, damit auch zukünftige Experimente auf dieser Plattform aufbauen können. So kann die Adapterplatte für den Quadrocopter aus Abbildung 8.23 auch für andere Modelle angepasst werden oder durch die Adapterplatten wird neue Hardware integriert. Die Adapterplatten der Gehäuse haben einen Schnappverschluss, der einen sicheren Halt bietet, wie in Abbildung 8.23 illustriert. Das Gehäuse für die Mikrocontroller verfügt über einen Einschub für Lochrasterplatinen, die für prototypische Aufbauten oftmals verwendet werden.

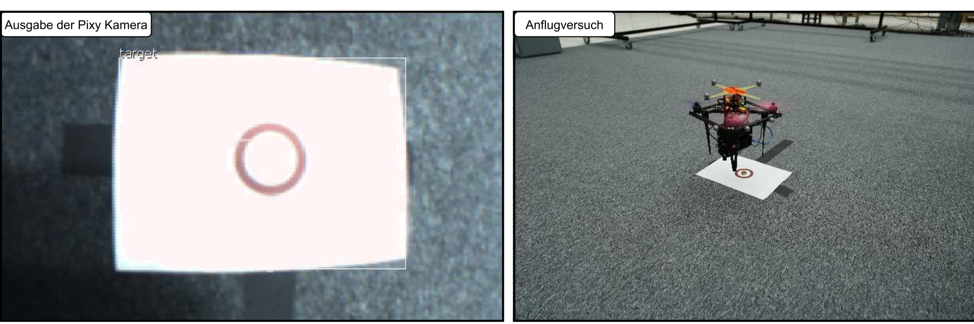

Abbildung 8.24. Screenshot der Pixy Bildauswertung [42], in der das Papier als Ziel erkannt und umrandet wird. Daneben ist ein Anflugversuch für das Wiegen des Werkstücks illustriert. (Das linke Bild ist aus der grafischen Oberfläche des Programms *Pixymon v2* entnommen.)

Der Semantic Controller ist auf einem Raspberry Pi 3 B+ installiert, die Semantic Adapter des Elektromagneten und der Pixy Kamera benutzen hingegen jeweils den Mikrocontroller *Arduino Nano*. Die Prozesslogik ist in der OOD-API programmiert und ebenfalls auf dem Raspberry Pi installiert. Der Quadrocopter erhält über WLAN die Ist-Positionen des Vicon-Systems für die Positionssteuerung, die über ROS auf einem PC eingespeist werden. Die ROS-Komponenten sind in ROSSi modelliert und in einen Dockercontainer gekapselt, können jedoch nicht automatisiert verteilt werden, da keine Komponenten von Semantic Plug & Play auf dem PC verwendet werden. Die Kommandierung der Soll-Positionen hingegen erfolgt innerhalb der OOD-API, indem eine virtuelle Fähigkeit verwendet wird, die mit Selbstbeschreibungsmechanismen ergänzt ist. So kann ein Device angelegt werden, das alle Fähigkeiten des Quadrocopters kapselt. Dazu gehören *Fliege zu Position*, das *Messen* der Position und das *Messen* des aktuellen Stromverbrauchs. Von der Kamera wird lediglich das *Messen* der Position des gesuchten Werkstücks und vom Elektromagneten das An- und Ausschalten benötigt.

Ergebnisse
Die Bewertung der Ergebnisse wird in die vier Phasen des Experiments unterteilt.

1. **Suche**
   Für den Suchalgorithmus wird der Suchraum in ein 2D-Raster unterteilt und in Linien abgesucht. Die Erkennung des Objekts findet über Farbkodierungen statt, die als Parameter in der Pixy angegeben wird. Wenn ein Objekt mit der passenden Farbe von der Pixy erkannt wird (siehe Abbildung 8.24), so werden die Positionsdaten der Umrandung über SPI an den Arduino Nano gesendet, der die Kommunikation über die Schnittstellen-API integriert und dem Semantic Controller bereitstellt. Da die Pixy eine *Axis Aligned Bounding Box* für das erkannte Objekt verwendet, ist das zu transportierende Werkstück mittig platziert. Der in der OOD-API implementierte Algorithmus besteht aus einer Kommandierung der Positionen aller Rasterelemente für den Quadrocopter, deren Größe sich am Ausfallwinkel und der Flughöhe der Kamera orientiert. Wird ein Objekt erkannt, so wird die 2D-Position mit der Position des Quadrocopters und der Montageposition

verrechnet und entsprechend korrigiert, bis die Anflugposition für den nächsten Schritt des Wiegens eingenommen ist. In Abbildung 8.24 ist ein Zielanflug zu sehen, wobei die Bodenturbulenzen und der relativ hohe Aufbau des Prototypen die Zielgenauigkeit beeinflussen, die im Durchschnitt bei etwa einem Radius von 5 Zentimetern lag.

2. **Wiegen**
Das Wiegen ist als virtuelle Fähigkeit mit eigenem Prozess modelliert, der als Vorbedingung einen gelandeten Quadrocopter fordert. Der Prozess ist rein sequentiell realisiert und beginnt mit dem Anschalten des Elektromagneten. Danach startet der Quadrocopter und fliegt eine parametrisierbare Höhe an und verweilt für 5 Sekunden, in denen der Stromverbrauch gemessen wird. Anschließend wird die Landung initiiert. Anhand der Gewichtsauswertung aus Abbildung 8.21 wird das Gesamtgewicht ermittelt und die Gewichte der aktuellen Konfiguration abgezogen, um die externe Last zu identifizieren.

   Für die Durchführung des Experiments wurden unterschiedliche Werkstücke für den Transport an dem mit 25 Newton Haltekraft angegebenen Magneten verwendet. Eine Metallbox mit einem Gewicht von 20g, ein Maulschlüssel mit 50g und letztlich drei 5-Cent-Münzen mit insgesamt 12g. Durch den Downwash des Quadrocopters wurde die runde Metallbox mit einem Durchmesser von 8cm bei einem Landeflug verschoben. Der Maulschlüssel blieb hingegen liegen und konnte vom Magneten aufgenommen werden, die Vibrationen des Quadrocopters in Kombination mit der Tragkraft des Magneten führten jedoch noch vor der Messung zu einem unerwünschten Abwurf. Für die Messung der drei aneinander geklebten 5-Cent-Münzen mit einem Durchmesser von 21,25mm ist die Zielgenauigkeit des Quadrocopters zu gering. Ferner ist die Genauigkeit des Stromverbrauchs in Relation zum Gesamtgewicht des Quadrocopters, wie in Abbildung 8.21 zu sehen, nicht aussagekräftig genug, um das Gewicht zu ermitteln. So sind erst Schätzungen ab ca. 100 Gramm mit dem Prototypen des Experiments möglich. Damit kann ein Quadrocopter als Anemometer verwendet werden, einen Messturm imitieren, eine präzise Waage jedoch nicht ersetzen. Diese Erkenntnis beweist einmal mehr, dass Eigenschaften aus den Dokumenten der Hersteller nicht für jede Domänenapplikation genügen und eine Rückführung der gewonnenen Informationen eine erweiterbare Wissensbasis erfordert, die in Semantic Plug & Play geboten wird. So kann eine maximale Traglast empirisch ermittelt und als Eigenschaft für den Magneten in der Domäne ergänzt werden. Für den nächsten Schritt wird daher die Annahme getroffen, dass die Traglast überschritten sei und eine Rekonfiguration durch das Entfernen der Kamera nötig ist.

3. **Rekonfiguration**
Als Vorbedingung für die Rekonfiguration wird der Quadrocopter an der Startposition gelandet und die Motoren werden abgestellt (disarmed). In der Prozessdefinition kann die Rekonfiguration auf unterschiedliche Weise realisiert werden. So kann explizit auf die Deregistrierung des Kameradevices gewartet oder das Gesamtgewicht zyklisch bis zur Unterschreitung der definierten Grenze abgefragt werden. Im Experiment wurde die zweite Variante angewendet und es wurde

anschließend gewartet, bis der Quadrocopter wieder in der Nähe des Startpunkts abgestellt wird. Das Montagesystem aus Abbildung 8.23 erlaubt eine schnelle und einfache Demontage der Kamera und trotzdem einen stabilen Halt während des Flugs. Der Code der OOD-API für die Berechnung des Gesamtgewichts ist bereits in Abschnitt 6.3 aufgeführt und in entsprechender Weise müssen lediglich die einzelnen Fähigkeiten sequentiell angeordnet und bei Bedarf synchron verwendet werden. Nach der Platzierung des Quadrocopters und nach Ablauf einer Minute, damit der Nutzer einen angemessenen Abstand einnehmen kann, wird der Transport initiiert.

4. **Transport**
   Analog zum Wiegen ist auch für den Transport die automatische Aufnahme des Werkstücks nicht möglich. Der Prozess besteht jedoch ebenfalls nur aus einer sequentiellen Abfolge von Fähigkeiten des Quadrocopters und des Elektromagneten.

Das Ziel des Experiments, einen automatisierten Pick & Place-Prozess mit einem Quadrocopter zu realisieren, ist gescheitert. Die unzureichende Haltekraft des Magneten in Kombination mit der mangelhaften Präzision des Zielanflugs führten zu manuellen Eingriffen in den Prozess selbst. Die Illustration der Stärken des Semantic Plug & Play-Ansatzes hingegen liegen nicht im Gelingen einzelner Experimente, sondern in der Wiederverwendbarkeit, einfachen Integration und der Wertschöpfung durch Erfahrungswerte. So können die Fähigkeiten des Quadrocopters, insbesondere die Messung des Stromverbrauchs, aus dem vorherigen Experiment wiederverwendet werden. Die Integration des Elektromagneten wird physisch durch das Montagesystem unterstützt, während die Schnittstellen-API eine einfache Migration der Implementierung ermöglicht. Die Relevanz spezifischer Domänenanforderungen, die in der MDDM-Architektur explizit behandelt werden, wird durch das Experiment untermauert. So sind die Anforderungen an den Elektromagneten, das Werkstück und die Genauigkeit der Kamera zur Designzeit nicht bekannt und werden erst durch das Experiment selbst eruiert. Die Trennung zwischen der implementierten Fähigkeit und der dazu assoziierten, erweiterbaren Selbstbeschreibung bietet die Möglichkeit, diese Erkenntnisse zurückzuführen und die Wissensbasis für weitere Experimente, auch von anderen Forschergruppen, zu erweitern. Die Semantic Web-Technologien bieten dafür eine Schnittstelle, die nicht nur lokal verwendet werden kann, sondern auch global einen Mehrwert erzeugt.

Einordnung in die Fallstudie
Die Fallstudie verbindet eine abstrakte Planung mit einer verteilten Ausführung in einem Multiagentensystem, in dem emergente Effekte genutzt und rekonfigurierbare Hardware in ScORe-Missionen generisch eingesetzt werden. Dass die Anforderungen dieser Domäne bereits im Vorfeld vollständig definiert und alle Eigenschaften für die Ausführung der einzelnen Phasen festgelegt werden können, ist illusorisch, wie das Experiment demonstriert. Semantic Plug & Play bietet eine Strategie für die Wiederverwendbarkeit von Fähigkeiten, die durch Eigenschaften an die Domäne angepasst werden können und den Entwicklungsprozess unterstützen. Das vorgestellte Montagesystem ermöglicht eine prototypische Realisierung mit standardisierter Hardware, ohne proprietäre Vorschriften für die Schnittstellen zu definieren. Die Variabilität der

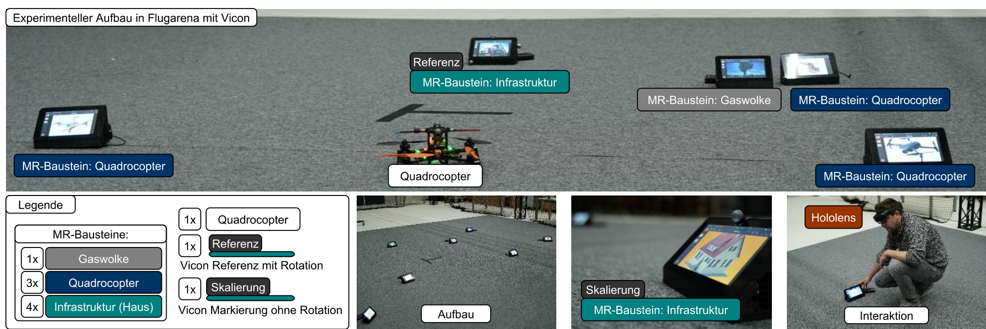

Abbildung 8.25. Aufbau des Experiments *GoLive* in der großen Flugarena mit insgesamt acht MR-Bausteinen und einem Quadrocopter. Zwei der MR-Bausteine sind mit Vicon-Markern ausgestattet, um ein Bezugskoordinatensystem zwischen der Hololens und dem Vicon-System herzustellen. (Eigenwerk)

Verteilung, die Unterstützung unterschiedlicher Betriebssysteme und Mikrocontroller, sowie ROS als Erweiterung bietet eine Basis für die Realisierung der Fallstudie, die sich aus der Summe einzelner Experimente ergibt. Um diese These weiter zu untermauern, wird in folgendem Abschnitt die modulare Simulation anhand des GoLive-Experiments demonstriert und eine schrittweise Überführung von der Simulation in die Realität vollzogen.

8.7 GoLive

Das Experiment *GoLive* wird in der großen Flugarena mit dem Vicon-Trackingsystem durchgeführt. In GoLive wird die modulare Simulation gezeigt, die durch ein Baukastensystem adaptive Szenarien ermöglicht. In Abbildung 8.25 ist der experimentelle Aufbau abgebildet. Insgesamt werden acht MR-Bausteine, ein Quadrocopter und die Hololens [137] von Microsoft verwendet. Die MR-Bausteine repräsentieren jeweils ein Simulationsobjekt, wie die Gaswolke, Quadrocopter oder Häuser. Zusammengefasst sollen die folgenden Aspekte durch das Experiment untersucht werden:

Selbstbeschreibung und Simulation
Die Selbstbeschreibung in Semantic Plug & Play ist von der Implementierung entkoppelt und kann beliebig erweitert werden. Durch GoLive wird eine Erweiterung vorgestellt, die mit 3D- und Simulationsobjekten eine Simulation adaptiver Szenarien präsentiert.

Modulare Umgebung
Für adaptive Szenarien wird eine physisch manipulierbare Umgebung verwendet, die durch MR-Bausteine eine Schnittstelle für Interaktionen bietet. Diese können, wie in Abbildung 8.25 illustriert, beliebig in der Flugarena positioniert werden.

Simulationsobjekte
Neben der Interaktion über MR-Bausteine für statische Infrastrukturen werden in der Simulation auch steuerbare Objekte mit Fähigkeiten betrachtet. So kann

ein MR-Baustein auch einen simulierten Quadrocopter repräsentierten, der über ROS anhand seiner Fähigkeiten gesteuert werden kann. Neben simulierten Quadrocoptern wird ebenfalls ein Simulationsobjekt vorgestellt, das die Simulationsumgebung beeinflusst, aber über keine Fähigkeiten steuerbar ist. So verbreitet bspw. die Gaswolke Gaspartikel, die von simulierten Sensoren wahrgenommen werden können, sie besitzt selbst aber keine Fähigkeiten. Durch die einfache Integration virtueller Fähigkeiten in Semantic Plug & Play können die Codefragmente der Simulationsobjekte gekapselt und in der OOD-API bereitgestellt werden. Geometrische Beschreibungen und Abhängigkeiten werden hingegen in den Eigenschaften beschrieben.

Schrittweise Überführung von der Simulation in die Realität
Die Schrittweise Überführung von der Simulation in die Realität basiert auf einem gemeinsamen Grundverständnis der Umgebung. Um dieses Grundverständnis herzustellen, wird das externe Trackinsystem *Vicon* verwendet, um die Positionen realer Elemente mit den simulierten Objekten der Hololens anhand eines Bezugskoordinatensystems zu verbinden. Für das Bezugskoordinatensystem werden zwei MR-Bausteine mit Vicon-Markern ausgestattet, wobei ein Referenz-MR-Baustein (siehe Abbildung 8.25) ein Muster aus Vicon-Markern für die Bestimmung der Position und Rotation (Frame) erhält und ein weiterer nur einen Vicon-Marker für die Position. Insgesamt werden zwei Interaktionen zwischen der virtuellen und der realen Welt untersucht. Erstens wird die simulierte Gaswolke mit einem echten Quadrocopter unter Verwendung eines simulierten Gassensors gemessen. Zweitens erhalten alle Simulationsobjekte, realen Objekte und die Infrastruktur (Häuser) eine Kollisionshülle für eine Kollisionserkennung.

Für die Realisierung des GoLive-Experiments wird nahezu der gesamte Technologiestack von Semantic Plug & Play verwendet, wie in Abbildung 8.26 abgebildet. Die simulierten und realen Quadrocopter werden mit ROS gesteuert, wobei die Modellierung der Launchfiles und Nodes in ROSSi erfolgt. Virtuelle Fähigkeiten werden für die Injektion der ROS-Nodes verwendet und in der OOD-API bereitgestellt, die Verteilungsmechanik von Semantic Plug & Play werden für ROS eingesetzt. Für die physikalischen MR-Bausteine wird folgend ein Prototyp vorgestellt, der auf einen Einplatinencomputer aufbaut.

Prototypen
Der Prototyp für den MR-Baustein besteht aus einem aus PLA gedruckten Gehäuse für einen Touchscreen und einen Raspberry Pi 4, wie in Abbildung 8.26 abgebildet. Die hintere Öffnung am Gehäuse dient als Durchführung für den Stromanschluss einer handelsüblichen Powerbank. Auf dem Raspberry Pi 4 ist ein Semantic Controller installiert, sowie ein Semantic Adapter für das zu repräsentierende Simulationsobjekt. Für die Bereitstellung der 3D-Objekte wird Collada in Kombination mit URDF analog zu den Beispielen in Abschnitt 7.3 verwendet. Die Kollisionshüllen sind in URDF eingetragen. Die 3D-Objekte können als Pfad zur Datei oder Referenz auf ein GIT-Repository in den RDF-Graphen eingetragen werden. Durch die Verwendung eines GIT-Repositories können die 3D-Objekte bei statischen Modellen einfacher ausgetauscht werden. QR-Codes hingegen unterstützen durch die sehr eingegrenzte Speichermöglichkeit innerhalb der eigenen Kodierung nur Referenzen auf GIT-Repositories. Für die Markererkennung der Hololens

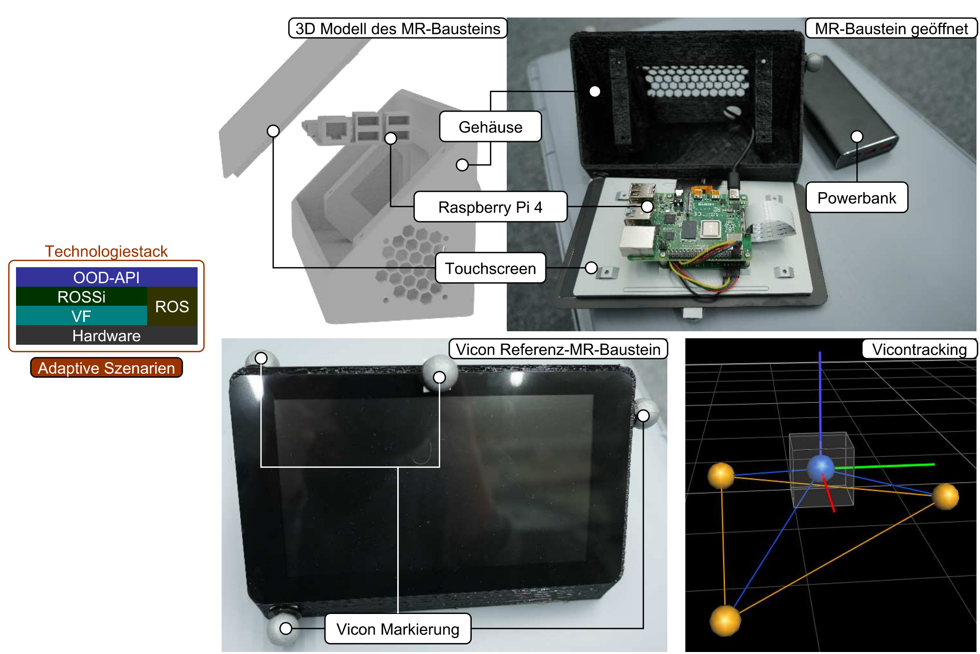

Abbildung 8.26. Das 3D-Modell für den MR-Baustein ist 3D-gedruckt und besteht aus einem Touchscreen und einem Raspberry Pi 4, das mit einer externen Powerbank betrieben werden kann. Der Vicon-Referenz-MR-Baustein ist mit vier Vicon-Markern ausgestattet und erlaubt eine Bestimmung der Position und Rotation (Frame) innerhalb des Vicon-Systems. (Eigenwerk)

wird ein Bild auf dem Touchscreen dargestellt, das ebenfalls Teil der Selbstbeschreibung ist. Der OOD-API-Code besteht aus einer einfachen Registrierung aller Devices und der automatischen Ausführung der ROS-Initialisierung bei Simulationsobjekten wie dem Quadrocopter. Die Simulation wird auf einem PC ausgeführt und kommuniziert über ROS mit den Nodes aller Simulationsobjekte. Der TF-Graph von ROS besitzt eine Schnittstelle zu dem Koordinatensystemen der Hololens und des Vicon-Systems. Damit ein Bezug zwischen beiden Koordinatensystemen hergestellt werden kann, wird ein Referenz-MR-Baustein mit vier Markern ausgestattet und im Vicon-System ein Objekt für das Tracking angelegt, wie in Abbildung 8.26 illustriert. Der in Abbildung 8.25 skizzierte MR-Baustein für die Skalierung ist mit nur einem Marker ausgestattet und dient primär dem Ausgleich der Skalierung zwischen dem Vicon-System und der Hololens. Erst nachdem die Hololens den Referenz- und Skalierungs-MR-Baustein erfasst hat, wird das Bezugskoordinatensystem erstellt und ein Skalierungsoffset bestimmt. Anhand des Bezugskoordinatensystems können die Positionen realer Objekte an die Position der Hololens in der Simulation angeglichen werden.

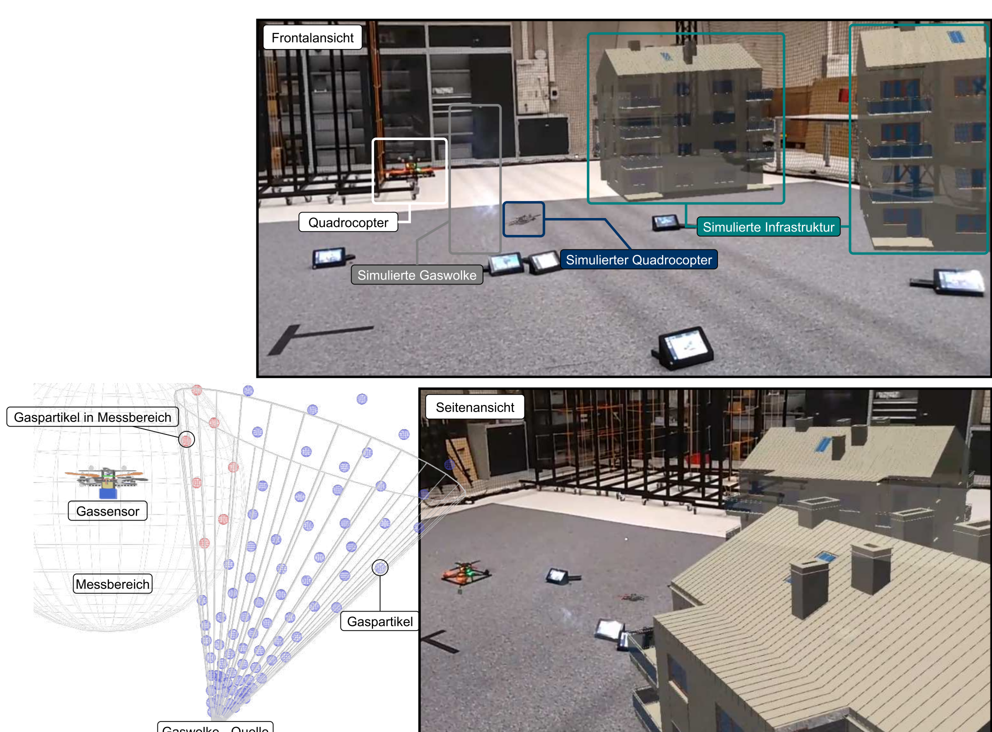

Abbildung 8.27. Die Gaswolkensimulation basiert auf einem Partikeleffekt mit Kegelausbreitung. Sobald ein simulierter Gaspartikel im Messbereich erkannt wird, ist im simulierten Sensor ein Messwert zu verzeichnen, dessen Intensität anhand der Anzahl erkannter Gaspartikel ab- und zunimmt. Sowohl der simulierte als auch der reale Quadrocopter sind mit simulierten Gassensoren ausgestattet. (Eigenwerk)

Ergebnisse

Das Experiment illustriert ein neues Konzept für die Simulation adaptiver Szenarien mit einem *Tangible Userinterface* (TUI) anhand modularer MR-Bausteine. Aufbauend auf dem Semantic Plug & Play-Technologiestack und unter Einsatz der entwickelten Prototypen werden folgende Aspekte ausgewertet:

Selbstbeschreibung und Simulation

In der Selbstbeschreibung wird ein Marker für die Projektion der 3D-Objekte verwendet. Der Marker wird über den Touchscreen angezeigt und von der Hololens registriert. Eine automatische Ausführung des generischen OOD-API-Codes führt zu einer schnellen Integration neuer MR-Bausteine, da lediglich die Selbstbeschreibung angepasst werden muss, also das 3D-Modell und der entsprechende Code. Die Applikation auf der Hololens kann nicht zur Laufzeit beliebig erweitert werden, sondern greift auf bekannte Referenzen der GIT-Repositories mit 3D-Objekten und Markern zurück. So ist ein Neustart der Applikation nötig, in dem die Referenzen aufgelöst und die 3D-Modelle geladen werden.

Die Simulation verwendet für reine Simulationsanwendungen ohne das Vicon-Trackingsystem das Koordinatensystemmodell der Hololens. So können rein simulative Anwendungen auch unabhängig des Vicon-Trackingsystems verwendet werden.

Modulare Umgebung
Durch die MR-Bausteine kann eine Simulationsumgebung mit physischen Elementen aufgebaut werden, wie in Abbildung 8.27 dargestellt. Das Tracking der Hololens (zweite Generation) ohne Unterstützung des Vicon-Systems funktioniert ohne Positionsänderungen zur Laufzeit stabil. Werden hingegen zur Laufzeit die MR-Bausteine umpositioniert, so wird das Hologramm nicht zwingend automatisch angepasst. Dieser Umstand ist unter anderem auf die kleinen Marker zurückzuführen, die erst bei einer Distanz zwischen 10 bis 20cm von der Brille erkannt werden. Unterschiedliche Markervarianten, wie etwa Bilder mit eindeutigen Konturen statt QR-Codes aus Abbildung 8.25 führten zu keinen messbaren Verbesserungen. Ein logischer Ansatzpunkt für dieses Problem ist der Einsatz größerer Bildschirme für die Anzeige der Marker.

Simulationsobjekte
Die steuerbaren Geräte werden im Experiment über ROS-Nodes realisiert, die für die Steuerung dieselbe Schnittstelle verwenden wie echte Quadrocopter. So ist eine Unterscheidung zwischen realem und simuliertem Quadrocopter in der OOD-API nur anhand der Eigenschaften möglich, die Fähigkeiten ändern sich nicht. Simulationsobjekte ohne expliziten MR-Baustein, wie etwa der in Abbildung 8.27 skizzierte Gassensor, werden durch einen Semantic Adapter auf dem MR-Baustein des Quadrocopters installiert und über die Simulation an das Koordinatensystem des Quadrocopters angehängt. Ein 3D-Objekt ist dafür nicht zwingend erforderlich. Im Fall der Kombination eines realen Quadrocopters mit einem simulierten Gassensor kann der Semantic Adapter des Gassensors auf einem beliebigen MR-Baustein aufgespielt werden. Die Fähigkeit *Messen* des simulierten Gassensors verändert die Schnittstelle der Fähigkeit nicht, intern werden jedoch Schnittstellen für den Messwert zur Simulation benötigt. Die Simulation verwendet Partikeleffekte für die Simulation der Gaswolke. Die Gaspartikel werden wie in Abbildung 8.27 an einer Quelle emittiert und breiten sich randomisiert in einem Kegel aus. Für den Gassensor wird anhand der euklidischen Distanz die Anzahl der Gaspartikel im definierten Messbereich gezählt und auf den Messbereich abgebildet. Die Visualisierungskomponente der Hololens verwendet Raucheffekte der Unity-Gameengine für die Gaswolke, die in den Abbildungen aus Abbildung 8.27 aufgrund des Hintergrunds nur schlecht zu sehen sind. Durch die Trennung zwischen Simulation und Visualisierung sind ebenfalls unterschiedliche Sichten möglich. Einerseits 3D-Objekte, die in der Visualisierung der Simulation, bspw. Rviz [236] in ROS, eine Nachvollziehbarkeit der Messwerte illustrieren, wie im linken Teil der Abbildung 8.27, und andererseits die illustrativen Effekte der Gameengine für eine realitätsnähere Darstellung des Szenarios.

Schrittweise Überführung von der Simulation in die Realität
Ein Referenz- und Skalierungs-MR-Baustein dient dem Bezugskoordinatensystem zwischen dem Vicontracking und dem Tracking der Hololens. In dem Experiment konnte ein virtueller Gassensor an einen realen Quadrocopter montiert und eine Messung in der simulierten Gaswolke vorgenommen werden. Da die Marker der MR-Bausteine zur Laufzeit aufgrund ihrer Größe nicht von der Hololens erkannt werden, wird lediglich die Odometrie der Hololens verwendet, um die Position zu bestimmen. Der Odometry-Shift der Hololens ohne kontinuierliche Markererkennung ist bereits ausführlich in mehreren Publikationen [79, 85] evaluiert und publiziert worden. Dieser Umstand führt zu einem Versatz zwischen den simulierten und realen Objekten, der sich nicht nur auf die Gaswolke, sondern auch die Kollisionserkennung auswirkt. Ein Lösungsansatz für diese Problemstellung bietet sich durch eine Kalibrierung der Hololens mit dem Vicon, wie in Debarba et al. [57] vorgeschlagen, oder durch die alleinige Nutzung des Vicons als Basis für Hologramme. Damit müssen allerdings sowohl die Hololens als auch alle simulierten MR-Bausteine mit Vicon-Markern ausgestattet werden. Zusammengefasst ist das Konzept für eine schrittweise Überführung der Simulation in die Realität mit Semantic Plug & Play realisiert, technische Hürden für einen präzisen und flüssigen Ablauf gilt es jedoch noch zu bewältigen.

Die Ergebnisse illustrieren die Einsatzmöglichkeiten für Semantic Plug & Play auch außerhalb realer Hardware. So bildet die Selbstbeschreibung eine Basis für digitale Zwillinge mit simulierten Fähigkeiten, die innerhalb der OOD-API nur anhand der Eigenschaften unterscheidbar sind. MR-Bausteine ermöglichen nicht nur die Erstellung adaptiver Szenarien, sondern ebenfalls einen ersten Schritt in die Realität, da ein Einplatinencomputer mit Semantic Controller eingesetzt wird. So kann die verteilte Ausführung und die Kommunikation bereits in der Simulation getestet werden. Diese Aspekte kommen vor allem in Multiagentensystemen zum Tragen, die in der Fallstudie verwendet werden.

Einordnung in die Fallstudie
In der Fallstudie wird das Schema der ScORe-Mission eingeführt und auf die einzelnen Experimente angewendet. Die Hypothese des ScORe-Schemas ist die Anwendbarkeit auf alle Ausprägungen von Katastrophenszenarien. Eine vollständige ScORe-Mission wird in der Fallstudie anhand des Chemieunfalls präsentiert, in der als Phasenübergang eine Rekonfiguration der Systeme benötigt wird. Weitere Faktoren der Fallstudie sind die Herausforderungen realer Einsätze der Feuerwehr, die in unterschiedlichen Umgebungen stattfinden. Die modulare Simulation, die anhand des Experiments *GoLive* illustriert ist, adressiert eben diese Aspekte mit einer Mischung aus realer Hardware, Selbstbeschreibung und einer Simulation mit getrennter MR-Visualisierung. So kann die Variabilität der ScORe-Missionen in unterschiedlichen Szenarien getestet und Rekonfigurationen mit realen Hardwareelementen erprobt werden. Die technischen Herausforderungen eines Quadrocopterschwarms können durch Visualisierungen in Szene gesetzt und durch den gezielten Einsatz echter Quadrocopter in der Realität demonstriert werden.

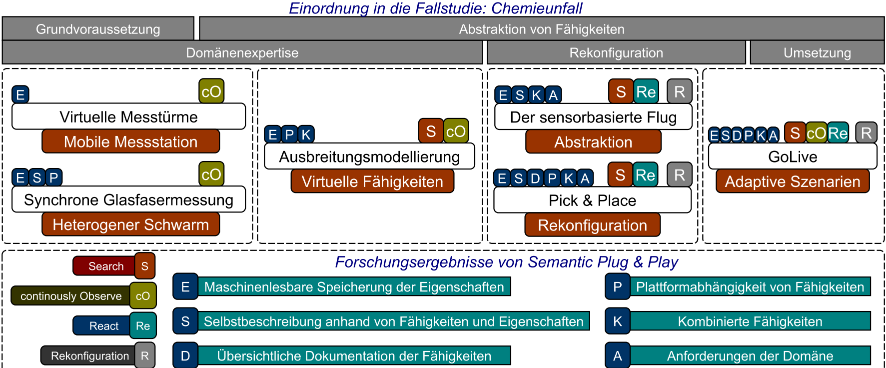

Abbildung 8.28. Einordnung der Experimente in die Themengebiete der Fallstudie, das ScORe-Schema und die Forschungsergebnisse von Semantic Plug & Play. Die Einordnung in die Themengebiete der Fallstudie erstrecken sich über mehrere Experimente, wobei die Legende im unteren Bereich Kürzel darstellt, die den einzelnen Experimenten konkret zugeordnet sind. (Eigenwerk)

8.8 Zusammenfassung der Experimente

In der Fallstudie des Chemieunfalls werden multipotente Systeme und ein generischer Ansatz für Katastrophenszenarien anhand des ScORe-Schemas vorgestellt. Die Unterteilung in sechs Experimente illustriert nicht nur eine Anwendbarkeit des ScORe-Schemas, sondern ebenfalls eine systematische Annäherung an die Herausforderungen der Fallstudie, die folgend kategorisiert und in Abbildung 8.28 illustriert sind:

Grundvoraussetzung
Für die Realisierung der Fallstudie müssen zwei Grundvoraussetzungen erfüllt werden. Einerseits die Verwendung eines Quadrocopters als mobiles Messinstrument und andererseits die Schnittstelle zu einem Multiagentensystem über abstrakte Fähigkeiten.

Das Experiment der *virtuellen Messtürme* wurde als Machbarkeitsnachweis für Temperatur- und Luftfeuchtemessungen durchgeführt und wurde anschließend um das Konzept einer mobilen Messtation mit austauschbaren Sensoren erweitert.

In dem Experiment der *synchronen Glasfasermessung* wird ein Multiagentensystem mit einem heterogenen Schwarm präsentiert. Über abstrakte Fähigkeiten und Eigenschaften können die einzelnen Schwarmteilnehmer gesteuert und synchronisiert werden.

Domänenexpertise
Die Domänenexpertise ist für die Realisierung der Fallstudie nötig. So ist nicht nur eine abschließende Bewertung der Messergebnisse aus den Experimenten der *synchronen Glasfasermessung* und den *virtuellen Messtürmen* nötig, sondern auch ein Nachweis für die Verarbeitung der Messdaten zur Laufzeit. Das Experiment der *Ausbreitungsmodellierung* verwendet die virtuellen Fähigkeiten für die Analyse der Windrichtung, die als Grundlage für die Quellensuche und eine Ausbreitungsanalyse zur Laufzeit erforderlich sind.

Abstraktion von Fähigkeiten
Die Abstraktion von Fähigkeiten und das Prinzip der - durch die Domäne bestimmten - Granularität zieht sich durch alle Experimente und bildet das Leitthema von Semantic Plug & Play. In der Fallstudie wird die Menge unterschiedlicher Sensoren nicht beschränkt, da konkrete Gassensoren nicht definiert sind. Damit bildet ein generisches Prinzip für die Abstraktion einen Grundpfeiler für die Realisierung.

Rekonfiguration
In allen Experimenten mit Ausnahme des Experiments der *Ausbreitungsmodellierung* werden modulare Sensoren anhand von Prototypen eingesetzt und untersucht. Speziell die Experimente *der sensorbasierte Flug* und *Pick & Place* thematisieren die Austauschbarkeit von Sensoren der gleichen Klasse und Rekonfigurationen für einen Prozess mit unterschiedlichen Phasen. Beide Faktoren werden in der Fallstudie aufgezeigt.

Umsetzung
Die Fallstudie behandelt Multiagentensysteme, die in unterschiedlichen Umgebungen reaktiv eingesetzt werden und Rekonfigurationen der Hardware adaptiv verwenden. Somit werden für die Umsetzung nicht nur eine Vielzahl an Quadrocoptern, sondern auch zahlreiche modulare Sensoren und Aktuatoren benötigt. Das Konzept der modularen Simulation unterstützt eine hybride Realisierung der Fallstudie, in der die Umgebung durch *MR-Bausteine* angepasst und eine Teilmenge des Schwarms simuliert werden kann. Das Experiment *GoLive* präsentiert in einem kleinen Umfang die Möglichkeiten der modularen Simulation und damit eine Strategie für die Umsetzung.

Semantic Plug & Play ist jedoch keinesfalls nur auf die Domäne der Fallstudie anwendbar, sondern zielt mit seinem umfassenden und unterschiedlich verwendbaren Technologiestack auf breitgefächerte Einsatzmöglichkeiten. So ist in den Experimenten, wie in Abbildung 8.29 zusammengefasst, nicht nur die Fallstudie adressiert, sondern es werden gezielt unterschiedliche Prototypen vorgestellt, um die Forschungsthemen von Semantic Plug & Play aufzufassen. Die Einordnung der Forschungsthemen von Semantic Plug & Play sind in Abbildung 8.28 skizziert und über die Kürzel an den Experimenten vermerkt. Folgend werden die Forschungsthemen kurz zusammengefasst und mit den Experimenten in Verbindung gebracht.

Maschinenlesbare Speicherung der Eigenschaften
Für die maschinenlesbare Speicherung der Eigenschaften werden die Technologien des Semantic Web verwendet, die als Teilgraph über den Semantic Adapter bereitgestellt werden. Alle Experimente verwenden diese Eigenschaften für die Beschreibung von Fähigkeiten oder zur Nachbereitung von Messergebnissen.

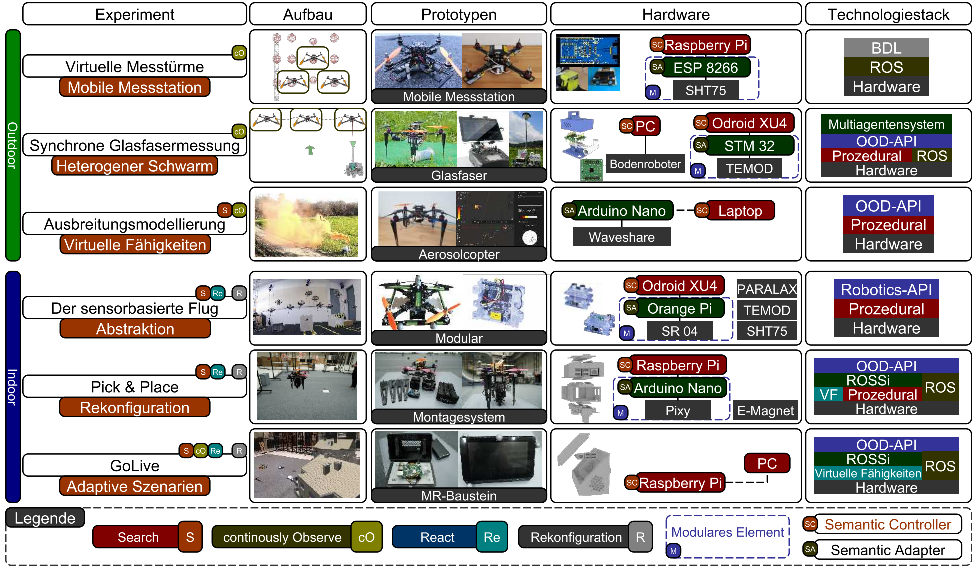

Abbildung 8.29. Zusammenfassung aller Experimente in einer Tabelle. Die Legende im unteren Bereich fasst die verwendeten Kürzel zusammen, die in den Spalten *Experiment* und *Hardware* verwendet werden. (Eigenwerk)

Selbstbeschreibung anhand von Fähigkeiten und Eigenschaften
Eine Speicherung der Eigenschaften und Fähigkeiten auf der Hardware als Selbstbeschreibung findet über einen Hardwareadapter in modularen Systemen statt, auf dem ein Semantic Adapter installiert ist. Die Experimente *virtuelle Messtürme* und *Ausbreitungsmodellierung* verwenden extern gespeicherte Eigenschaften für die Interpretation der Messdaten oder als Grundlage für die kombinierte Fähigkeit *Messen* des virtuellen Anemometers.

Übersichtliche Dokumentation der Fähigkeiten
Die Dokumentation ergibt sich aus den Eigenschaften der Selbstbeschreibung. In allen Experimenten werden die Eigenschaften festgehalten. Die Experimente *Pick & Place* und *GoLive* verwenden die Eigenschaften hingegen zur Laufzeit als Basis für Entscheidungen für eine Rekonfiguration.

Plattformabhängigkeit von Fähigkeiten
Die Experimente *virtuelle Messtürme* und *der sensorbasierte Flug* bauen auf der Robotics API und ROS auf und besitzen dadurch Einschränkungen bezüglich der verwendbaren Plattformen. So können bspw. keine Mikrocontroller mit der Robotics API als Hardwareadapter verwendet werden. In Abbildung 8.29 werden alle Hardwareelemente zusammengefasst vorgestellt und es wird erläutert welche Teile des Technologiestacks angewendet wurden.

Kombinierte Fähigkeiten
Für kombinierte Fähigkeiten wurden in unterschiedlichen Experimenten unter anderem ein Anemometer, der Gradientenflug und das Wiegen vorgestellt.

Anforderungen der Domäne
Die Integration der MDDM-Architektur aus Kapitel 6 mit dem Abgleich zwischen Anforderungen der Domäne und verfügbaren Fähigkeiten der aktuellen Hardwarekonfiguration ist im Experiment *Pick & Place* vorgestellt und in *GoLive* nutzbar. Das Experiment *der sensorbasierte Flug* behandelt das Thema, jedoch ist die MDDM-Architektur nicht verwendet worden.

Jedes Experiment umfasst mindestens einen Prototypen, der nicht nur in der eingesetzten Domäne verwendbar ist. So bietet die *mobile Messstation* aus Abbildung 8.29 ein modulares Stecksystem für in situ Messungen der Meteorologie [317] oder kann in Katastrophenszenarien durch die einfache Integration von Sensoren über den Hardwareadapter eingesetzt werden. Eine allgemeine Notfallsteuerung für Quadrocopterschwärme ist ein Teil des Prototypen *Glasfaser*. Unterschiedliche Montagesysteme für modulare Sensoren sind hingegen in den Prototypen *Modular* mit magnetischen Halterungen und in *Montagesystem* aus Abbildung 8.29 mit einem Stecksystem eingeflossen. Die *MR-Bausteine* bieten eine Basis für die Umsetzung der Ensembleanwendungen mit multipotenten Systemen, wobei der Einplatinencomputer ebenfalls in realen Szenarien eingesetzt werden kann.

Zusammengefasst wurde der Semantic Controller in den Experimenten auf typischen X86-Systemen, dem Raspberry Pi 3 und 4, sowie dem Odroid XU4 eingesetzt. Der Semantic Controller wurde zusätzlich zu den Plattformen des Semantic Controllers auf den Mikrocontrollern ESP 8266, STM 32 und Arduino Nano verwendet, wobei in den Experimenten *virtuelle Messtürme* und *synchrone Glasfasermessung* selbst entwickelte Platinen zum Einsatz kamen.

Die verschiedenen Möglichkeiten, wie der Technologiestack von Semantic Plug & Play eingesetzt werden kann, ist ebenfalls in den Experimenten illustriert. So wird keine Variante des Technologiestacks in den sechs Experimenten mehrfach verwendet, wie in Abbildung 8.29 zu sehen.

Zusammenfassung. In diesem Kapitel werden die Forschungsergebnisse, die im Rahmen der Dissertation entstanden sind, aufgegriffen und bewertet. Abschließend werden die Potentiale von Semantic Plug & Play erörtert und es wird ein Ausblick auf offene Forschungsthemen gegeben. Zudem wird auf ein Projekt, das bereits einige der Forschungsthemen behandelt, besonders eingegangen.

Fazit und Ausblick

9.1 Bewertung der erzielten Ergebnisse 193
9.2 Ausblick 196

Das einleitende und provokative Zitat: *„Leute, die es mit Software wirklich genau nehmen, sollten ihre eigene Hardware bauen"* von Alan Curtis Kay wurde in der vorliegenden Dissertation sowohl bestätigt als auch angegriffen. So wurden im Rahmen der Dissertation eigene Platinen, Forschungsplattformen und modularisierte Sensoren, sowie ein Technologiestack für deren Nutzung vorgestellt. Ergo ist das Zitat bestätigt und die eigene Hardware wurde zumindest in Teilen selbst realisiert. Das Ziel des Technologiestacks von Semantic Plug & Play ist die Verwendung der Hardware über abstrakte Fähigkeiten und über erweiterbare Selbstbeschreibungen anhand von Eigenschaften, also die Vermeidung selbst gebauter Hardware, um eine Wiederverwendbarkeit auch domänenübergreifend zu ermöglichen. Die erzielten Forschungsergebnisse von Semantic Plug & Play werden in Abschnitt 9.1 zusammengefasst und bewertet. Folgend wird das Potential von Semantic Plug & Play in anderen Domänen erörtert und ein Ausblick auf weitere Forschungsthemen gegeben, wie etwa die *autonome Inspektion großer Bauteile* in der Industrie.

9.1 Bewertung der erzielten Ergebnisse

Im Rahmen der Fallstudie wurde ein heterogener Quadrocopterschwarm mit rekonfigurierbarer Hardware über Selbstorganisationsmechanismen gesteuert. Die Integration der Rekonfiguration in die Planung und Ausführung ist als multipotentes System definiert, wobei sehr abstrakte Hardwarefähigkeiten als Schnittstelle zur Hardware verwendet werden. In Semantic Plug & Play ist die Lücke zwischen abstrakt definierten Fähigkeiten und realen Hardwaresystemen geschlossen worden. Anders als in vergleichbaren Projekten [87, 251, 284] ist jedoch kein an die Domäne angepasstes, proprietäres System entstanden. Vielmehr bietet Semantic Plug & Play einen ganzen Technologiestack für die

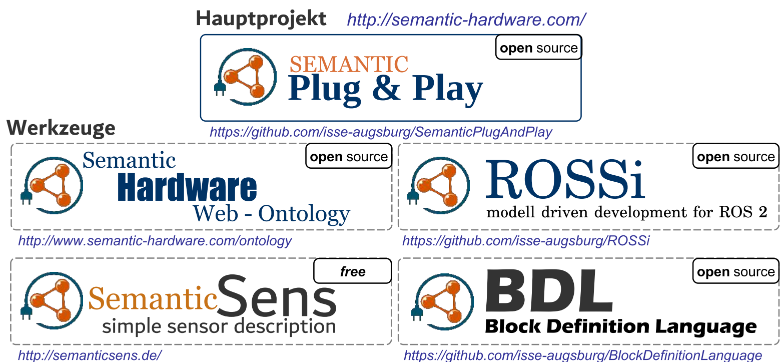

Abbildung 9.1. Übersicht über alle Projekte der vorliegenden Dissertation, die entweder Open-Source verfügbar, oder kostenlos nutzbar sind. In der Übersichtsseite *http://semantic-hardware.com/* sind alle Projekte verlinkt. (Eigenwerk)

Integration selbstbeschreibender Hardware, wie in Abbildung 9.1 abgebildet. Alle Projekte sind entweder kostenlos nutzbar oder stehen unter der MIT-Lizenz Open-Source zur Verfügung. Die Grundarchitektur von Semantic Plug & Play bietet eine Trennung zwischen der Domäne und einer Selbstbeschreibung, die mit Semantic Web-Mechanismen umgesetzt ist.

Die **maschinenlesbare Speicherung der Eigenschaften** wird auf der *Semantic Hardware Web* Ontologie aus Abschnitt 5.1 aufgebaut und kann über *SemanticSens* aus Abschnitt 7.1 instanziiert und veröffentlicht werden. Eine Trennung der beschreibenden Eigenschaften von den ausführbaren Fähigkeiten ermöglicht nicht nur eine Verteilung der Wissensbasis, sondern bietet vielmehr einen kollektiven Ansatz für die Selbstbeschreibung. So kann die Erkenntnis der Vibrationsempfindlichkeit des Lasersensors aus dem Experiment des *sensorbasierten Flugs* als Eigenschaft des Sensors hinzugefügt und veröffentlicht werden.

Eine **Selbstbeschreibung anhand von Fähigkeiten und Eigenschaften** ermöglicht die in Kapitel 6 vorgestellte Architektur von Semantic Plug & Play. Die konkrete Implementierung einer Fähigkeit erfolgt über die Schnittstellen-API aus Abschnitt 6.5.2 oder über virtuelle Fähigkeiten aus Abschnitt 6.4.2. Beide Varianten bilden in Kombination mit der Selbstbeschreibung einen Semantic Adapter, der in den Experimenten sowohl auf X86- und ARM-Plattformen, als auch auf Mikrocontrollern betrieben wird.

Die **übersichtliche Dokumentation der Fähigkeiten** basiert auf den Mechanismen des Semantic Web. So können unterschiedliche Ontologien verknüpft werden, um Inhalte gezielt mit passenden Vokabularien zu beschreiben. Diese Erweiterbarkeit wurde in Abschnitt 5.1.2 vorgestellt und ist ein integraler Bestandteil von *SemanticSens* aus Abschnitt 7.1. Damit ist nicht nur die Grundlage für eine automatisierte Dokumentationsgenerierung, sondern auch die direkte Verwendung im Code selbst möglich. So kann im Experiment des *sensorbasierten Flugs* aus Abschnitt 8.5 die Fähigkeit *Gradientenflug* anhand ihrer Eigenschaften entscheiden, ob eine temperatur- oder distanzgesteuerte Regelung die Position des Quadrocopters beeinflusst.

Eine **Plattformabhängigkeit von Fähigkeiten** kann in Semantic Plug & Play nicht vollständig aufgehoben werden. So bedingen komplexe Hardwareelemente einen vom Betriebssystem abhängigen Treiber, oder Hardware für Mikrocontroller benötigen entsprechende Bibliotheken. In Semantic Plug & Play werden Treiber durch virtuelle Fähigkeiten aus Abschnitt 6.4.2 an einen Semantic Adapter gekoppelt, über Dockercontainer gekapselt und mit GIT-Repositories bereitgestellt. Bei Mikrocontrollern bietet die Schnittstellen-API aus Abschnitt 6.5.2 eine Möglichkeit externe Bibliotheken an das *Self Descriptive Protocol* (SDP) aus Abschnitt 6.5.1 anzubinden. Für die Integration der häufig verwendeten Middleware ROS für (mobile) Roboter ist in Abschnitt 7.2 sowohl ein modellgetriebener Ansatz mit ROSSi vorgestellt worden, als auch die Block Definition Language (BDL) für die Abbildung von Prozessen, die ebenfalls in Abbildung 9.1 aufgeführt sind.

Die Kombination von Fähigkeiten dient der Formung eines Prozesses, kann aber auch in einer wiederverwendbaren Form als **Kombinierte Fähigkeit** abgebildet werden. Kombinierte Fähigkeiten können in virtuellen Fähigkeiten aus Abschnitt 6.4.2 umgesetzt werden. Zur Designzeit sind virtuelle und physische Fähigkeiten in der OOD-API aus Abschnitt 6.3 nicht unterscheidbar. Für die Referenz auf andere Fähigkeiten werden die Anforderungen der Domäne analog zur OOD-API verwendet und ein *BottomUp-Ansatz* für die Bildung der kombinierten Fähigkeiten anhand der vorhandenen Hardware in Abschnitt 6.4.2 vorgestellt. Ein *TopDown-Ansatz* wird hingegen für die Instanziierung und Auflösung der Abhängigkeiten zur Laufzeit eingesetzt. In dem Experiment *Pick & Place* aus Abschnitt 8.6 wurde für die kombinierte Fähigkeit *Wiegen* eine Kompatibilität zur ROS-Integration illustriert.

Anforderungen der Domäne Die Hypothese von Semantic Plug & Play ist, dass jede Domäne eigene Anforderungen an die Hardware hat. So benötigt das Experiment *virtueller Messturm* von dem Quadrocopter nur die Fähigkeiten *Fliege zu Position* und das *Messen* der Position, in dem Experiment des *sensorbasierten Fluges* wird auch der Stromverbrauch für die Berechnung der Flugzeit verwendet. Die MDDM-Architektur aus Kapitel 6 separiert die Anforderungen von der verteilten Wissensbasis und bietet ein auf JSON basiertes Austauschformat. Anforderungen werden im Domainmodel durch eine in Abschnitt 6.4 vorgestellte Graphensuche überprüft und als JSON-Datei der OOD-API zur Verfügung gestellt. Die Graphensuche verwendet neben SPARQL-Queris die Levenshtein-Distanz, um auch unpräzise Formulierungen der Anforderungen zu bewältigen. Dieses Prinzip kapselt die Semantic Web-Mechanismen in das Domainmodel und ermöglicht dem Entwickler textbasierte Anfragen in einer objektorientierten Umgebung, der OOD-API aus Abschnitt 6.3.

Zusammengefasst nutzt Semantic Plug & Play die Vorteile des Semantic Web für eine präzise und global verfügbare Wissensbasis, um eine Selbstbeschreibung der Hardware zu ermöglichen. Die MDDM-Architektur bildet eine Schnittstelle für objektorientierte Sprachen und verknüpft die Selbstbeschreibung mit implementierten Fähigkeiten. Die Experimente haben gezeigt, dass durch diese Mechanismen in Kombination mit den in Abbildung 9.1 abgebildeten Werkzeugen ein Quadrocopterschwarm mit abstrakten Fähigkeiten realisierbar ist und die Lücke zwischen den Anforderungen der Domäne und realer Hardware geschlossen werden kann. Die durch den Entwicklungsprozess

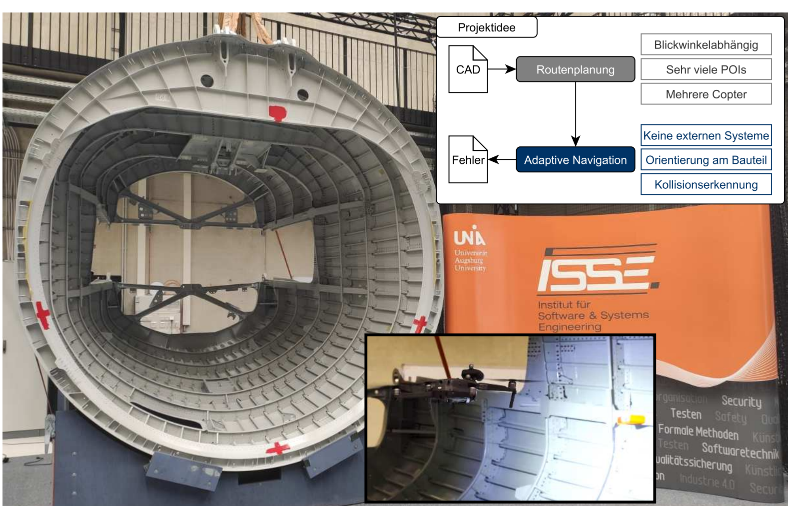

Abbildung 9.2. Halbschale eines Flugzeughecks, das von Premium Aerotec für das Projekt *Quality Assurance with Drones* (QuAD) in der großen Flugarena bereitgestellt wurde. Das Projekt behandelt eine autonome Drohneninspektion von Bauteilen ohne externe Trackingsysteme. (Eigenwerk)

entstehende, domänenübergreifende Erfahrung sollte dem Semantic Hardware Web zurückgeführt werden. So wird das Zitat von Alan Curtis Kay als Résumé der Dissertation folgend präzisiert:

> *„Leute, die es mit Software wirklich genau nehmen, sollten ihre Anforderungen kennen und das Domänenwissen teilen."* – *Constantin Wanninger*

9.2 Ausblick

Semantic Plug & Play bietet bereits einen umfangreichen Technologiestack, der in den Experimenten aus Kapitel 8 unterschiedlich angewendet wurde. Die Verwendung der Semantic Web-Technologie als Instrument für die Selbstbeschreibung und auch für die Bereitstellung der Messdaten gewährt jedoch nur einen kleinen Einblick in das Potential dieser Verfahrensweise. So bietet die vorgestellte Ontologie des *Semantic Hardware Web* in Kombination mit dem Werkzeug *SemanticSens* eine Möglichkeit, die Dokumente der Hardwarehersteller in eine maschinenlesbare Form zu überführen. Die resultierenden RDF-Graphen können nicht nur in Semantic Plug & Play genutzt, sondern auch als Grundlage für die Filter- und Suchmechanismen bspw. für Online-Shop-Systeme verwendet, als *Rich-Snippets* [188] visualisiert oder von Recommendersystemen für die Generierung von Vorschlägen eingesetzt werden.

Die Vielfalt der Anwendungsmöglichkeiten ist bereits in die Ausgestaltung weiterer Forschungsvorhaben am Institut eingeflossen, wie bspw. in das Projekt *Quality Assurance with Drones* (QuAD). In QuAD sollen Flugzeugbauteile, wie in Abbildung 9.2 abgebildet, ohne externes Tracking inspiziert und Fehler festgestellt werden. Die Routenplanung

wird anhand einer computer-aided design (CAD) Datei automatisiert erstellt, wobei die zahlreichen zu inspizierenden Punkte (POIs) und die Blickwinkelabhängigkeit zu beachten sind. In den Publikationen [267, 310] sind bereits Lösungsansätze dafür vorgestellt. Für Semantic Plug & Play bieten sich neue Herausforderung in der Verknüpfung einer Routenplanung mit CAD-Daten und der adaptiven Navigation mehrerer Quadrocopter. Die adaptive Navigation basiert auf der Odomoetrie des Quadrocopters in Kombination mit einer Kamera, deren Bild für eine Überlagerung mit der CAD-Datei verwendet wird. Anhand der Überlagerung kann einerseits die Position relativ zum Bauteil als Orientierung für den Quadrocopter verwendet und andererseits eine Fehlersuche durchgeführt werden. Die theoretischen Grundlagen des Verfahrens sind bereits in einer Publikation [192] vorgestellt. Als virtuelle Fähigkeit benötigt die adaptive Navigation nicht nur die Anbindung an die lokale Sensorik für eine Kollisionserkennung, sondern auch eine CAD-Datei für den Abgleich und Filtermechanismen für einen stabilen Flug ohne die genaue Positionserkennung externer Trackingsysteme. Somit bietet sich für die *adaptive Navigation* eine sehr komplexe, aber wiederverwendbare Fähigkeit an, die es in dem Projekt QuAD zu realisieren gilt und Gegenstand der Dissertation meines Kollegen *Martin Schörner* ist.

Die durchgeführten Experimente für die Illustration des Semantic Plug & Play-Technologiestacks haben nicht nur den käuflich erwerbbaren Forschungsquadrocopter, unterschiedliche Montagesysteme und Platinenlayouts hervorgebracht, sondern auch neue Konzepte, wie die modulare Simulation in dem Experiment *GoLive*, vorgestellt. Eine Erweiterung der Interaktionsmöglichkeiten, kooperative Aspekte, Browsermechaniken und vieles mehr sind Forschungsinhalte der Dissertation meines Kollegen *Michael Filipenko*.

Eine Adaption der Semantic Plug & Play-Mechanismen im Forschungsfeld der industriellen Roboter mit einem Schwerpunkt für Echtzeitsysteme wird hingegen von meinem Kollegen *Christian Eymüller* als Forschungsziel definiert. Dabei werden vor allem Reasoner eingesetzt, um logische Schlussfolgerungen aus den RDF-Graphen zu ziehen, die in dieser Dissertation nicht fokussiert wurden.

Abseits der bereits laufenden Forschung am Institut soll Semantic Plug & Play als eine Möglichkeit betrachtet werden, das Konzept eines maschinenlesbaren Internets von Tim Berners Lee auf Hardware und deren Verwendung zu übertragen. Dabei steht nicht nur die Suche und Vernetzung im Fokus, sondern die Vorteile der Maschinenlesbarkeit können in Semantic Plug & Play direkt im Programmcode verwendet werden. Die MDDM-Architektur bietet eine Abkehr der Implementierung gegen Hardwareschnittstellen hin zu einer auf Anforderungen basierenden Prozessdefinition. Die freie Erweiterbarkeit der Beschreibung soll zu einer kollektiven Wissensbasis führen, in der unterschiedliches Domänenwissen, Erfahrungen und Fehlschläge dokumentiert werden können. Damit soll nicht nur die Zusammenarbeit unterschiedlicher Forschergruppen unterstützt, sondern allgemein eine Wiederverwendbarkeit der Ergebnisse hergestellt werden. Die dafür nötige Basis bilden die Open-Source-Projekte von Semantic Plug & Play.

Literaturverzeichnis

[1] ACME-Systems. Pinlayout - sht75. online, 2022. URL `https://www.acmesystems.it/89`. (abgerufen: 20.05.2022).

[2] M. Ahmad, H. Rong, S. Alhady, W. Rahiman, and W. Othman. Colour tracking technique by using pixy cmucam5 for wheelchair luggage follower. In *2017 7th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*, pages 186–191. IEEE, 2017.

[3] S. C. Akkaladevi, A. Pichler, M. Plasch, M. Ikeda, and M. Hofmann. Skill-based programming of complex robotic assembly tasks for industrial application. *e & i Elektrotechnik und Informationstechnik*, 136(7):326–333, 2019.

[4] T. Alby. Web 2.0. *Konzepte, Anwendungen, Technologien*, 2, 2007.

[5] M. Amarasinghe, S. Kottegoda, A. L. Arachchi, S. Muramudalige, H. D. Bandara, and A. Azeez. Cloud-based driver monitoring and vehicle diagnostic with obd2 telematics. In *2015 Fifteenth International Conference on Advances in ICT for Emerging Regions (ICTer)*, pages 243–249. IEEE, 2015.

[6] C. Anderson. Docker [software engineering]. *Ieee Software*, 32(3):102–c3, 2015.

[7] A. Andrade. Game engines: A survey. *EAI Endorsed Trans. Serious Games*, 2(6):e8, 2015.

[8] A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif. Robotics api: Object-oriented software development for industrial robots. *Journ. of Software Engineering for Robotics*, 4 (1):1–22, 2013.

[9] M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages with round-trip engineering. In *International Conference on Model Driven Engineering Languages and Systems*, pages 692–706. Springer, 2006.

[10] S. Arachchi and I. Perera. Continuous integration and continuous delivery pipeline automation for agile software project management. In *2018 Moratuwa Engineering Research Conference (MERCon)*, pages 156–161. IEEE, 2018.

[11] Arduino Team. Arduino-Übersichtsseite. online, 2022. URL `https://www.arduino.cc/`. (abgerufen: 20.05.2022).

[12] R. Arnaud and M. C. Barnes. *COLLADA: sailing the gulf of 3D digital content creation*. CRC Press, 2006.

[13] M. Asadullah and A. Raza. An overview of home automation systems. In *2016 2nd international conference on robotics and artificial intelligence (ICRAI)*, pages 27–31. IEEE, 2016.

[14] A. Barciś, M. Barciś, and C. Bettstetter. Robots that sync and swarm: A proof of concept in ros 2. In *2019 International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, pages 98–104. IEEE, 2019.

[15] P. Barnaghi, W. Wang, L. Dong, and C. Wang. A linked-data model for semantic sensor streams. In *2013 IEEE Intern. Conf. on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pages 468–475, 2013. doi: 10.1109/GreenCom-iThings-CPSCom.2013.95.

[16] T. Baur, M. Bastuck, C. Schultealbert, T. Sauerwald, and A. Schütze. Random gas mixtures for efficient gas sensor calibration. *Journal of Sensors and Sensor Systems*, 9(2):411–424, 2020.

[17] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 356–367. IEEE, 2017.

[18] V. Bense, T. Read, and A. Verhoef. Using distributed temperature sensing to monitor field scale dynamics of ground surface temperature and related substrate heat flux. *Agricultural and Forest Meteorology*, 220:207–215, 2016.

[19] T. Berners-Lee. The original proposal of the www. online, 1989. URL `https://www.w3.org/History/1989/proposal.html`. (abgerufen: 20.05.2022).

[20] T. Berners-Lee. Uniform resource identifier (uri). *RFC3986*, 2005.

[21] T. Berners-Lee. Linked open data. *See also: http://www. w3. org/2008/Talks/0617-lod-tbl*, 281:29, 2008.

[22] T. Berners-Lee, R. Fielding, and L. Masinter. Rfc2396: Uniform resource identifiers (uri): generic syntax, 1998.

[23] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific american*, 284(5): 34–43, 2001.

[24] T. Berners-Lee, J. Hendler, and O. Lassila. A new form of web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific american*, 2002.

[25] L. Bing, M. Qing-Hao, W. Jia-Ying, S. Biao, and W. Ying. Three-dimensional gas distribution mapping with a micro-drone. In *2015 34th Chinese Control Conference (CCC)*, pages 6011–6015. IEEE, 2015.

[26] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillere. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th annual computer software and applications conference*, pages 303–312. IEEE, 2013.

[27] M. I. Board. Mars climate orbiter mishap investigation board phase i report november 10, 1999, 1999.

[28] H.-J. Böhme. *Serviceroboter und intuitive Mensch-Roboter-Interaktion.* Techn. Univ., Fachgebiet Neuroinformatik, 2002.

[29] S. Borgo, A. Cesta, A. Orlandini, R. Rasconi, M. Suriano, and A. Umbrico. Towards a cooperative knowledge-based control agent for a reconfigurable manufacturing plant. In *Proc. of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8. IEEE, Sept 2014. doi: 10.1109/ETFA.2014.7005226.

[30] S. Boschert, C. Heinrich, and R. Rosen. Next generation digital twin. In *Proc. tmce*, volume 2018, pages 7–11. Las Palmas de Gran Canaria, Spain, 2018.

[31] M. Brambilla, C. Pinciroli, M. Birattari, and M. Dorigo. Property-driven design for swarm robotics. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 139–146, 2012.

[32] L. Braubach, A. Pokahr, and W. Lamersdorf. Extending the capability concept for flexible bdi agent modularization. In *Int. Workshop on Programming Multi-Agent Systems*, pages 139–155. Springer, 2005.

[33] J. Broekstra, M. Klein, S. Decker, D. Fensel, F. Van Harmelen, and I. Horrocks. Enabling knowledge representation on the web by extending rdf schema. In *Proceedings of the 10th international conference on World Wide Web*, pages 467–478, 2001.

[34] T. E. Brook and R. Narayanaswamy. Polymeric films in optical gas sensors. *Sensors and Actuators B: Chemical*, 51(1-3):77–83, 1998.

[35] A. Bröring, P. Maué, K. Janowicz, D. Nüst, and C. Malewski. Semantically-enabled sensor plug & play for the sensor web. *Sensors*, 11(8):7568–7605, 2011.

[36] M. Broy. *Cyber-physical systems: Innovation durch softwareintensive eingebettete Systeme*. Springer-Verlag, 2011.

[37] J. Burgués, V. Hernández, A. J. Lilienthal, and S. Marco. Gas distribution mapping and source localization using a 3d grid of metal oxide semiconductor sensors. *Sensors and Actuators B: Chemical*, 304:127309, 2020.

[38] J. Busse, B. Humm, C. Lübbert, F. Moelter, A. Reibold, M. Rewald, V. Schlüter, B. Seiler, E. Tegtmeier, and T. Zeh. Was bedeutet eigentlich ontologie? *Informatik-Spektrum*, 37(4): 286–297, 2014.

[39] Y. Cao, Y. Leng, J. Sun, Y. Zhang, and W. Ge. 360botg2 - an improved unit of mobile self-assembling modular robotic system aiming at exploration in real world. In *Industrial Electronics Society, IECON 2015-41st Annual Conf. of the IEEE*, pages 001716–001722. IEEE, 2015.

[40] J. Časar, J. Fischer, and V. Starỳ. Measuring of copter motor power characteristics. *Cybern. Lett*, 2015.

[41] S. Chacon and B. Straub. Git branching. In *Pro Git*, pages 43–77. Springer, 2014.

[42] Charmedlabs. Produktseite: Pixycam. online, 2022. URL `https://pixycam.com/`. (abgerufen: 20.05.2022).

[43] A. Chatterjee and A. Maltz. Microsoft directshow: A new media architecture. *SMPTE journal*, 106(12):865–871, 1997.

[44] J. Christensen and J. Ekstedt. Evaluation of plugin frameworks for the jenkins continuous integration build server, 2012.

[45] CNBC. BASF Faces Prolonged Shut-Down After Chemical Site Explosion. online, 2018. URL `https://www.bloomberg.com/news/articles/2016-10-17/basf-reports-explosion-at-its-biggest-site-in-ludwigshafen`. (abgerufen: 20.05.2022).

[46] CNBC. Chemical plant explosion thrusts Arkema into spotlight. online, 2018. URL `https://www.cnbc.com/2017/09/02/chemical-plant-explosion-thrusts-arkema-into-spotlight.html`. (abgerufen: 20.05.2022).

[47] B. Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72. Berkeley, CA, USA, 2003.

[48] B. Cohen and K. Pietrzak. The chia network blockchain, 2019.

[49] R. Components. Produkseite: Sht75. online, 2022. URL `https://de.rs-online.com/web/p/temperatur-und-luftfeuchtigkeit-sensoren/6675271`. (abgerufen: 20.05.2022).

[50] M. Compton, C. Henson, L. Lefort, H. Neuhaus, and A. Sheth. A survey of the semantic specification of sensors. In *Proc. of the 2nd Int. Conf. on Semantic Sensor Networks-Volume 522*, pages 17–32. CEUR-WS. org, 2009.

[51] M. Compton, P. Barnaghi, L. Bermudez, R. Garcia-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, et al. The ssn ontology of the w3c semantic sensor network incubator group. *Journal of Web Semantics*, 17:25–32, 2012.

[52] J. Craighead, R. Murphy, J. Burke, and B. Goldiez. A survey of commercial & open source unmanned vehicle simulators. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 852–857. IEEE, 2007.

[53] D. Crockford. The application/json media type for javascript object notation (json). *RFC 4627*, 2006.

[54] H. Cui. Design and research on automotive controller area network bus analyzer. *Sensors & Transducers*, 166(3):91, 2014.

[55] K. Daniel, B. Dusza, A. Lewandowski, and C. Wietfelds. Airshield: A system-of-systems muav remote sensing architecture for disaster response. In *Proc. 3rd Annual IEEE Systems Conf. (SysCon)*, 2009.

[56] G. De Cubber, D. Serrano, K. Berns, K. Chintamani, R. Sabino, S. Ourevitch, D. Doroftei, C. Armbrust, T. Flamma, and Y. Baudoin. Search and rescue robots developed by the european icarus project. In *7th Int. Workshop on Robotics for Risky Environments*. Citeseer, 2013.

[57] H. G. Debarba, M. E. de Oliveira, A. Lädermann, S. Chaqué, and C. Charbonnier. Tracking a consumer hmd with a third party motion capture system. In *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pages 539–540. IEEE, 2018.

[58] F. Dellaert, D. Fox, W. Burgard, and S. Thrun. Monte carlo localization for mobile robots. In *Proceedings 1999 IEEE international conference on robotics and automation (Cat. No. 99CH36288C)*, volume 2, pages 1322–1328. IEEE, 1999.

[59] M. Diab, M. Pomarlan, D. Beßler, A. Akbari, J. Rosell, J. Bateman, and M. Beetz. Skillman—a skill-based robotic manipulation framework based on perception and reasoning. *Robotics and Autonomous Systems*, 134:103653, 2020.

[60] R. Diankov, R. Ueda, K. Okada, and H. Saito. Collada: An open standard for robot file formats. In *Proceedings of the 29th Annual Conference of the Robotics Society of Japan, AC2Q1–5*, 2011.

[61] M. Dibley, H. Li, Y. Rezgui, and J. Miles. An integrated framework utilising software agent reasoning and ontology models for sensor based building monitoring. *Journ. of Civil Engineering and Management*, 21(3):356–375, 2015.

[62] V. DiLuoffo, W. R. Michalson, and B. Sunar. Robot operating system 2: The need for a holistic security approach to robotic architectures. *International Journal of Advanced Robotic Systems*, 15(3):1729881418770011, 2018.

[63] L. Ding and T. Finin. Characterizing the semantic web on the web. In *International Semantic Web Conference*, pages 242–257. Springer, 2006.

[64] M. Dorigo, D. Floreano, L. M. Gambardella, and more. Swarmanoid: A novel concept for the study of heterogeneous robotic swarms. *IEEE RAM*, 20(4):60–71, 2013. ISSN 1070-9932. doi: 10.1109/MRA.2013.2252996.

[65] M. Dorigo, M. Birattari, and M. Brambilla. Swarm robotics. *Scholarpedia*, 9(1):1463, 2014.

[66] K. Dorofeev. Skill-based engineering in industrial automation domain: skills modeling and orchestration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pages 158–161, 2020.

[67] B. Dorsemaine, J.-P. Gaulier, J.-P. Wary, N. Kheir, and P. Urien. Internet of things: a definition & taxonomy. In *2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies*, pages 72–77. IEEE, 2015.

[68] N. Dousse, G. Heitz, and D. Floreano. Extension of a ground control interface for swarms of small drones. *Artificial Life and Robotics*, 21(3):308–316, 2016.

[69] R. Drath. *Datenaustausch in der Anlagenplanung mit AutomationML: Integration von CAEX, PLCopen XML und COLLADA*. Springer-Verlag, 2009.

[70] M. Duarte, V. Costa, J. Gomes, T. Rodrigues, F. Silva, S. M. Oliveira, and A. L. Christensen. Evolution of collective behaviors for a real swarm of aquatic surface robots. *PLoS ONE*, 11 (3):1–25, 03 2016. doi: 10.1371/journal.pone.0151834. URL http://dx.doi.org/10.1371%2Fjournal.pone.0151834.

[71] G. Ducard and R. D'Andrea. Autonomous quadrotor flight using a vision system and accommodating frames misalignment. In *2009 IEEE International Symposium on Industrial Embedded Systems*, pages 261–264. IEEE, 2009.

[72] Duden. Definition: Aerosol. online, 2022. URL https://www.duden.de/rechtschreibung/Aerosol. (abgerufen: 20.05.2022).

[73] Duden. Definition: Eigenschaft. online, 2022. URL https://www.duden.de/rechtschreibung/Eigenschaft. (abgerufen: 20.05.2022).

[74] Duden. Definition: Fähigkeit. online, 2022. URL https://www.duden.de/rechtschreibung/Faehigkeit. (abgerufen: 20.05.2022).

[75] G. Echeverria, S. Lemaignan, A. Degroote, S. Lacroix, M. Karg, P. Koch, C. Lesire, and S. Stinckwich. Simulating complex robotic scenarios with morse. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 197–208. Springer, 2012.

[76] S. Z. Ed Durfree. *Multiagent Planning, Control, and Execution*, chapter 11, pages 485–545. MIT press, 2013.

[77] B. Edwards. Die ursprünge der sozialen netzwerke. online, 2022. URL https://www.pcwelt.de/ratgeber/Ratgeber-Soziale-Netzwerke-Die-Wurzeln-der-sozialen-Netzwerke-6504543.html#:~:text=DieerstesozialeNetzwerkseitewar,MySpaceundFacebookgegründetwurden. (abgerufen: 20.05.2022).

[78] M. Electronics. Produktseite: Sht75. online, 2022. URL https://www.mouser.de/ProductDetail/Sensirion/SHT75?qs=wWC4CIiyLaMWBKOsYD%2Bk5A==. (abgerufen: 20.05.2022).

[79] J. Elsdon. Shared control for hand-held robots. *Electrical and Electronic Engineering PhD theses - Imperial College London*, 2019.

[80] K. Erol, J. Hendler, and D. S. Nau. Htn planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128, 1994.

[81] C. P. et al. Planetwatch whitepaper. online, 2021. URL https://www.planetwatch.io/white-paper/pdf/white-paper.pdf. (abgerufen: 20.05.2022).

[82] Euronews. Eight injured in German oil refinery explosion. *Available at https://www.euronews.com/2018/09/01/eight-injured-in-german-oil-refinery-explosion, accessed on 2018-11-15*, 2018. URL https://www.euronews.com/2018/09/01/eight-injured-in-german-oil-refinery-explosion. (abgerufen: 20.05.2022).

[83] C. Eymüller, C. Wanninger, A. Hoffmann, et al. Semantic Plug and Play – Self-Descriptive Modular Hardware for Robotic Applications. In *International Journal of Semantic Computing (IJSC)*, 2018.

[84] C. Eymüller, C. Wanninger, A. Hoffmann, and W. Reif. Semantic plug and play—self-descriptive modular hardware for robotic applications. *International Journal of Semantic Computing*, 12(04):559–577, 2018.

[85] T. Feigl, A. Porada, S. Steiner, C. Löffler, C. Mutschler, and M. Philippsen. Localization limitations of arcore, arkit, and hololens in dynamic large-scale industry environments. In *VISIGRAPP (1: GRAPP)*, pages 307–318, 2020.

[86] S. Ferdoush and X. Li. Wireless sensor network system design using raspberry pi and arduino for environmental monitoring applications. *Procedia Computer Science*, 34:103 – 110, 2014. ISSN 1877-0509. doi: http://dx.doi.org/10.1016/j.procs.2014.07.059. URL `http://www.sciencedirect.com/science/article/pii/S1877050914009144`.

[87] E. F. Flushing, L. M. Gambardella, and G. A. D. Caro. A mathematical programming approach to collaborative missions with heterogeneous teams. In *2014 IEEE/RSJ Int. Conf. on Intell. Robots and Systems*, pages 396–403, 2014. doi: 10.1109/IROS.2014.6942590.

[88] O. S. R. Foundation. Gazebo. online, 2018. URL `http://gazebosim.org/`. (abgerufen: 20.05.2022).

[89] M. Frasheri, B. Cürüklü, M. Esktröm, and A. V. Papadopoulos. Adaptive autonomy in a search and rescue scenario. In *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 150–155. IEEE, 2018.

[90] T. M. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[91] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart. Rotors—a modular gazebo mav simulator framework. In *Robot operating system (ROS)*, pages 595–625. Springer, 2016.

[92] J. R. Garratt. The atmospheric boundary layer. *Earth-Science Reviews*, 37(1-2):89–134, 1994.

[93] I. Georgievski and M. Aiello. An overview of hierarchical task network planning. *CoRR*, abs/1403.7426, 2014. URL `http://arxiv.org/abs/1403.7426`.

[94] L. Gerencsér. The integration of dogs into collaborative human-robot teams-an applied ethological approach. online, 2015. URL `https://core.ac.uk/download/pdf/51326012.pdf`. (abgerufen: 20.05.2022).

[95] B. P. Gerkey and M. J. Matarić. A formal analysis and taxonomy of task allocation in multi-robot systems. *The Int. Journ. of Robotics Res.*, 23(9):939–954, 2004.

[96] A. Ghosh, H. Roy, and S. Dhar. Arduino quadcopter. In *2018 Fourth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)*, pages 280–283. IEEE, 2018.

[97] J. K. Gillan, J. W. Karl, A. Elaksher, and M. C. Duniway. Fine-resolution repeat topographic surveying of dryland landscapes using uas-based structure-from-motion photogrammetry: Assessing accuracy and precision against traditional ground-based erosion measurements. *Remote Sensing*, 9(5):437, 2017.

[98] G. I. GmbH. Temod. online, 2018. URL `https://www.mikrocontroller.net/attachment/46744/I2C_Temperaturmodul_DBD.pdf`. (abgerufen: 20.05.2022).

[99] J. Goldstein. Emergence as a construct: History and issues. *Emergence*, 1(1):49–72, 1999. doi: 10.1207/s15327000em0101_4. URL `http://dx.doi.org/10.1207/s15327000em0101_4`.

[100] L. M. Golston, L. Tao, C. Brosy, K. Schäfer, B. Wolf, J. McSpiritt, B. Buchholz, D. R. Caulton, D. Pan, M. A. Zondlo, et al. Lightweight mid-infrared methane sensor for unmanned aerial systems. *Applied Physics B*, 123(6):1–9, 2017.

[101] L. Gomes, L. Leal, T. Oliveira, J. Cunha, and T. Revoredo. Unmanned quadcopter control using a motion capture system. *IEEE Latin America Transactions*, 14(8):3606–3613, 2016.

[102] A. Gómez-Pérez. Ontological engineering: A state of the art. *Expert Update: Knowledge Based Systems and Applied Artificial Intelligence*, 2(3):33–43, 1999.

[103] R. S. Gonçalves, S. Bail, E. Jiménez-Ruiz, N. Matentzoglu, B. Parsia, B. Glimm, and Y. Kazakov. Owl reasoner evaluation (ore) workshop 2013 results. In *ORE*, pages 1–18. Citeseer, 2013.

[104] GoodRelations. The web vocabulary for e-commerce. online, 2022. URL `http://www.heppnetz.de/ontologies/goodrelations/v1.html#uml`. (abgerufen: 20.05.2022).

[105] C. Goodwin and D. J. Russomanno. An ontology-based sensor network prototype environment. In *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks*, pages 1–2, 2006.

[106] Google Inc. Dienste und smart-home-geräte, die mit google assistant kompatibel sind. online, 2022. URL `https://support.google.com/googlenest/answer/7639952?hl=de`. (abgerufen: 20.05.2022).

[107] V. Govindarajan, S. Bhattacharya, and V. Kumar. Human-robot collaborative topological exploration for search and rescue applications. In *Distributed Autonomous Robotic Systems*, pages 17–32. Springer, 2016.

[108] M. Graves, A. Constabaris, and D. Brickley. Foaf: Connecting people on the semantic web. *Cataloging & classification quarterly*, 43(3-4):191–202, 2007.

[109] B. R. Greene, A. R. Segales, T. M. Bell, E. A. Pillar-Little, and P. B. Chilson. Environmental and sensor integration influences on temperature measurements by rotary-wing unmanned aircraft systems. *Sensors*, 19(6):1470, 2019.

[110] M. D. Gross and C. Veitch. Beyond top down: Designing with cubelets. *Tecnologias, Sociedade e Conhecimento*, 1(1):150–164, 2013.

[111] R. Gross and M. Dorigo. Towards group transport by swarms of robots. *Intern. Journ. of Bio-Inspired Computation*, 1(1-2):1–13, 2009.

[112] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[113] K. Guo, Z. Qiu, C. Miao, A. H. Zaini, C.-L. Chen, W. Meng, and L. Xie. Ultra-wideband-based localization for quadcopter navigation. *Unmanned Systems*, 4(01):23–34, 2016.

[114] J. Haas. A history of the unity game engine. *Diss. WORCESTER POLYTECHNIC INSTITUTE*, 2014.

[115] S. Haller. The things in the internet of things. *Poster at the (IoT 2010). Tokyo, Japan, November*, 5(8):26–30, 2010.

[116] T. Hänisch. Grundlagen industrie 4.0. In *Industrie 4.0*, pages 9–31. Springer, 2017.

[117] Z. U. Haq, G. F. Khan, and T. Hussain. A comprehensive analysis of xml and json web technologies. *New Developments in Circuits, Systems, Signal Processing, Communications and Computers*, pages 102–109, 2013.

[118] A. Harth, M. Janik, and S. Staab. Semantic web architecture. In *Handbook of Semantic Web Technologies*. Springer, 2011.

[119] W. Hasselbring. Software-architektur. *Informatik-Spektrum*, 29(1):48–52, 2006.

[120] J.-E. Haugen, O. Tomic, and K. Kvaal. A calibration method for handling the temporal drift of solid state gas-sensors. *Analytica chimica acta*, 407(1-2):23–39, 2000.

[121] M. Hehn and R. D'Andrea. Quadrocopter trajectory generation and control. *IFAC proceedings Volumes*, 44(1):1485–1491, 2011.

[122] T. Heikkilä and J. M. Ahola. Robot skills-modeling and control aspects. In *2018 14th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA)*, pages 1–6. IEEE, 2018.

[123] F. Heimgaertner, S. Hettich, O. Kohlbacher, and M. Menth. Scaling home automation to public buildings: A distributed multiuser setup for openhab 2. In *2017 Global Internet of Things Summit (GIoTS)*, pages 1–6. IEEE, 2017.

[124] R. K. Heintz. Bibliotheksdaten im semantic web:'linked open data'-projekte im vergleich. online, 2011. URL `https://hdms.bsz-bw.de/frontdoor/deliver/index/docId/625/file/Bachelorarbeit_BIB_Rebecca_Heintz.pdf`. (abgerufen: 20.05.2022).

[125] R. HELGESSON. 29. universal serial bus. *Student Papers in Computer Architecture, 2004*, page 131, 2005.

[126] M. Hepp. The product types ontology. online, 2022. URL `http://www.productontology.org/`. (abgerufen: 20.05.2022).

[127] P. Hitzler, M. Krötzsch, S. Rudolph, and Y. Sure. *Semantic Web: Grundlagen*. Springer-Verlag, 2007.

[128] J.-T. Huang, L.-Y. Chang, and H.-C. Lin. Implementation of iot, wearable devices, google assistant and google cloud platform for elderly home care system. *7th International Conference on Information and Communication Technologies for Ageing Well and e-Health (ICT4AWE 2021), pages 203-212*, 2021.

[129] P.-C. Huang, Y.-H. Hsieh, and A. K. Mok. A skill-based programming system for robotic furniture assembly. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, pages 355–361. IEEE, 2018.

[130] A. Hunt and I. Stewart. Calibration of an ingestible temperature sensor. *Physiological Measurement*, 29(11):N71, 2008.

[131] J. Hunt. Gang of four design patterns. In *Scala design patterns*, pages 135–136. Springer, 2013.

[132] J. Jackson. Microsoft robotics studio: A technical introduction. *IEEE robotics & automation magazine*, 14(4):82–87, 2007.

[133] L. Jean-Baptiste. Annotations, multilingual texts, and full-text search. In *Ontologies with Python*, pages 187–205. Springer, 2021.

[134] M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt). Technical report, Internet Engineering Task Force (IETF), 2015.

[135] A. Kaba, A. Ermeydan, and E. Kiyak. Model derivation, attitude control and kalman filter estimation of a quadcopter. In *2017 4th International Conference on Electrical and Electronic Engineering (ICEEE)*, pages 210–214. IEEE, 2017.

[136] J. Kaiser. Source code dokumentation. online, 2022. URL `https://schegge.de/2019/09/source-code-dokumentation/`. (abgerufen: 20.05.2022).

[137] S. Karthika, P. Praveena, and M. GokilaMani. Hololens. *International Journal of Computer Science and Mobile Computing*, 6(2):41–50, 2017.

[138] U. Kastens and H. K. Büning. *Modellierung: Grundlagen und formale Methoden.* Carl Hanser Verlag GmbH Co KG, 2014.

[139] A. Katifori, C. Halatsis, G. Lepouras, C. Vassilakis, and E. Giannopoulou. Ontology visualization methods—a survey. *ACM Computing Surveys (CSUR)*, 39(4):10–es, 2007.

[140] A. Kawakami, A. Torii, K. Motomura, and S. Hirose. Smc rover: Planetary rover with transformable wheels. In *Experimental Robotics VIII*, pages 498–506. Springer, 2003.

[141] I. Keller, A. Lehmann, M. Franke, and T. Schlegel. Towards an interaction concept for efficient control of cyber-physical systems. In *Int. Conf. on Virtual, Augmented and Mixed Reality*, pages 149–158. Springer, 2014.

[142] G. Kellogg, P.-A. Champin, and D. Longley. *JSON-LD 1.1–A JSON-based Serialization for Linked Data.* PhD thesis, W3C, 2019.

[143] M. Kerres. Potenziale von web 2.0 nutzen. *Handbuch E-Learning. München: DWD*, 17: 1–16, 2006.

[144] F. Klinker. Exponential moving average versus moving exponential average. *Mathematische Semesterberichte*, 58(1):97–107, 2011.

[145] G. Kobilarov, T. Scott, Y. Raimond, S. Oliver, C. Sizemore, M. Smethurst, C. Bizer, and R. Lee. Media meets semantic web–how the bbc uses dbpedia and linked data to make connections. In *European semantic web conference*, pages 723–737. Springer, 2009.

[146] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004.

[147] J. K. Korpela. *Unicode explained.* O'Reilly Media, Inc., 2006.

[148] O. Kosak. A decentralised swarm approach for mobile robot-systems. In *Organic Computing: Doctoral Dissertation Colloquium 2015*, volume 7, page 53. kassel university press GmbH, 2015.

[149] O. Kosak. A decentralised swarm approach for mobile robot-systems. In *Organic Computing: Doctoral Dissertation Colloquium*, volume 7, page 53, 2015.

[150] O. Kosak. *Mission programming for flying ensembles: combining planning with self-organization.* doctoralthesis, Universität Augsburg, 2021.

[151] O. Kosak, C. Wanninger, A. Angerer, A. Hoffmann, A. Schiendorfer, and H. Seebach. Towards self-organizing swarms of reconfigurable self-aware robots. In *Found. and Applications of Self* Systems, IEEE Int. Workshops on*, pages 204–209. IEEE, 2016.

[152] O. Kosak, C. Wanninger, A. Angerer, A. Hoffmann, A. Schiendorfer, and H. Seebach. Towards self-organizing swarms of reconfigurable self-aware robots. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 204–209. IEEE, 2016.

[153] O. Kosak, C. Wanninger, A. Hoffmann, H. Ponsar, and W. Reif. Multipotent systems: Combining planning, self-organization, and reconfiguration in modular robot ensembles. *Sensors*, 19(1):17, 2019.

[154] O. Kosak, F. Bohn, L. Eing, D. Rall, C. Wanninger, A. Hoffmann, and W. Reif. Swarm and collective capabilities for multipotent robot ensembles. In *International Symposium on Leveraging Applications of Formal Methods*, pages 525–540. Springer, 2020.

[155] O. Kosak, L. Huhn, F. Bohn, C. Wanninger, A. Hoffmann, and W. Reif. Maple-swarm: programming collective behavior for ensembles by extending htn-planning. In *International Symposium on Leveraging Applications of Formal Methods*, pages 507–524. Springer, 2020.

[156] A. Koubâa, A. Allouch, M. Alajlan, Y. Javed, A. Belghith, and M. Khalgui. Micro air vehicle link (mavlink) in a nutshell: A survey. *IEEE Access*, 7:87658–87680, 2019.

[157] A. Koubâa et al. *Robot Operating System (ROS).*, volume 1. Springer, 2017.

[158] A. Kovari and P. Dukan. Kvm & openvz virtualization based iaas open source cloud virtualization platforms: Opennode, proxmox ve. In *2012 IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics*, pages 335–339. IEEE, 2012.

[159] N. Kshetri and D. Rojas-Torres. The 2018 winter olympics: A showcase of technological advancement. *IT Prof.*, 20(2):19–25, 2018.

[160] KT-elektronic. Ultraschall messmodul hc-sr04. online, 2022. URL `https://www.mikrocontroller.net/attachment/218122/HC-SR04_ultraschallmodul_beschreibung_3.pdf`. (abgerufen: 20.05.2022).

[161] E. Kuantama, R. Tarca, S. Dzitac, I. Dzitac, T. Vesselenyi, and I. Tarca. The design and experimental development of air scanning using a sniffer quadcopter. *Sensors*, 19(18):3849, 2019.

[162] J. Kucera, D. Chlapek, J. Klímek, and M. Necaskỳ. Methodologies and best practices for open data publication. In *DATESO*, pages 52–64, 2015.

[163] K. Kumar, S. I. Azid, A. Fagiolini, and M. Cirrincione. Erle-copter simulation using ros and gazebo. In *2020 IEEE 20th Mediterranean Electrotechnical Conference (MELECON)*, pages 259–263. IEEE, 2020.

[164] P. S. Kumar, W. Emfinger, G. Karsai, D. Watkins, B. Gasser, and A. Anilkumar. Rosmod: a toolsuite for modeling, generating, deploying, and managing distributed real-time component-based software using ros. *Electronics*, 5(3):53, 2016.

[165] A. Kyriakidis, K. Maniatis, and E. You. *The majesty of Vue. js.* Packt Publishing, 2016.

[166] I. Labbus. Prototypische umsetzung und exemplarische anwendung in der automobilen komponentenproduktion. In *Cyber-physische Produktionssysteme für die energieeffiziente Komponentenproduktion*, pages 163–200. Springer, 2021.

[167] M. Ladeira, Y. Ouhammou, and E. Grolleau. Robmex: Ros-based modelling framework for end-users and experts. *Journal of Systems Architecture*, 117:102089, 2021.

[168] C. Larman and V. R. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, 2003.

[169] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann. Industrie 4.0. *Wirtschaftsinformatik*, 56(4):261–264, 2014.

[170] J. A. Ledin. Hardware-in-the-loop simulation. *Embedded Systems Programming*, 12:42–62, 1999.

[171] C. Lee, M. Kim, Y. J. Kim, N. Hong, S. Ryu, H. J. Kim, and S. Kim. Soft robot review. *International Journal of Control, Automation and Systems*, 15(1):3–15, 2017.

[172] A. Leff and J. T. Rayfield. Web-application development using the model/view/controller design pattern. In *Proceedings fifth ieee international enterprise distributed object computing conference*, pages 118–127. IEEE, 2001.

[173] S.-H. Leitner and W. Mahnke. Opc ua–service-oriented architecture for industrial applications. *ABB Corporate Research Center*, 48(61-66):22, 2006.

[174] V. I. Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

[175] J.-G. Li, Q.-H. Meng, F. Li, M. Zeng, and D. Popescu. Mobile robot based odor source localization via particle filter. In *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, pages 2984–2989. IEEE, 2009.

[176] Y. Liang, C. Wu, S. Jiang, Y. J. Li, D. Wu, M. Li, P. Cheng, W. Yang, C. Cheng, L. Li, et al. Field comparison of electrochemical gas sensor data correction algorithms for ambient air measurements. *Sensors and Actuators B: Chemical*, 327:128897, 2021.

[177] A. Lilienthal and T. Duckett. Building gas concentration gridmaps with a mobile robot. *Robotics and Autonomous Systems*, 48(1):3–16, 2004.

[178] A. Lingayat, R. R. Badre, and A. K. Gupta. Performance evaluation for deploying docker containers on baremetal and virtual machine. In *2018 3rd International Conference on Communication and Electronics Systems (ICCES)*, pages 1019–1023. IEEE, 2018.

[179] T. Lochmatter. Bio-inspired and probabilistic algorithms for distributed odor source localization using mobile robots. online, 2010. URL `https://www.researchgate.net/profile/Thomas-Lochmatter/publication/40868110_Bio-inspired_and_probabilistic_algorithms_for_distributed_odor_source_localization_using_mobile_robots/links/00b4951c32e5f9c0b4000000/Bio-inspired-and-probabilistic-algorithms-for-distributed-odor-source-localization-using-mobile-robots.pdf`. (abgerufen: 20.05.2022).

[180] T. Locke. To elon musk, web3 seems more like a 'marketing buzzword' than a reality. online, 2021. URL `https://www.cnbc.com/2021/12/20/elon-musk-web3-seems-more-marketing-buzzword-than-reality-right-now.html`. (abgerufen: 20.05.2022).

[181] J. Loeliger and M. McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc., 2012.

[182] S. Luo, J. Bimbo, R. Dahiya, and H. Liu. Robotic tactile perception of object properties: A review. *Mechatronics*, 48:54–67, 2017.

[183] T. Luukkonen. Modelling and control of quadcopter. *Independent research project in applied mathematics, Espoo*, 22:22, 2011.

[184] W. Maalej and M. P. Robillard. Patterns of knowledge in api reference documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, 2013.

[185] K. Maeda. Performance evaluation of object serialization libraries in xml, json and binary formats. In *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*, pages 177–182. IEEE, 2012.

[186] S. Magnenat, M. Voelkle, and F. Mondada. Planner9, a HTN planner distributed on groups of miniature mobile robots. In *Intell. Robotics and Applications, Proceedings of the Second*

International Conference on Intelligent Robotics and Application, volume 5928 of *Lecture Notes in Computer Science*, pages 1013–1022. Springer, 2009. doi: 10.1007/978-3-642-10817-4. URL http://icira2009.robotics.sg/.

[187] Z. Makhataeva and H. A. Varol. Augmented reality for robotics: A review. *Robotics*, 9(2): 21, 2020.

[188] M.-C. Marcos, F. Gavin, and I. Arapakis. Effect of snippets on user experience in web search. In *Proceedings of the XVI International Conference on Human Computer Interaction*, pages 1–8, 2015.

[189] K. V. Mardia, P. E. Jupp, and K. Mardia. *Directional statistics*, volume 2. Wiley Online Library, 2000.

[190] J. Markoff. online, 2006. URL https://www.nytimes.com/2006/11/12/business/12web.html. (abgerufen: 20.05.2022).

[191] A. Martin and M. R. Emami. An architecture for robotic hardware-in-the-loop simulation. In *2006 International Conference on Mechatronics and Automation*, pages 2162–2167. IEEE, 2006.

[192] C. W. Martin Schoerner, Michelle Bettendorf. Uav inspection of large components: Indoor navigation relative to structures. 18th International Conference on Informatics in Control, Automation and Robotics, IEEE, 2021. doi: 10.5220/0010556300002994.

[193] C. Martinez, C. Sampedro, A. Chauhan, and P. Campoy. Towards autonomous detection and tracking of electric towers for aerial power line inspection. In *2014 Intern. Conf. on Unmanned Aircraft Systems (ICUAS)*, pages 284–295, 2014. doi: 10.1109/ICUAS.2014.6842267.

[194] M. Masse. *REST API design rulebook: designing consistent RESTful web service interfaces*. O'Reilly Media, Inc., 2011.

[195] L. Matikainen, M. Lehtomäki, E. Ahokas, J. Hyyppä, M. Karjalainen, A. Jaakkola, A. Kukko, and T. Heinonen. Remote sensing methods for power line corridor surveys. *ISPRS Journ. of Photogrammetry and Remote Sensing*, 119(Supplement C):10 – 31, 2016. ISSN 0924-2716. doi: https://doi.org/10.1016/j.isprsjprs.2016.04.011. URL http://www.sciencedirect.com/science/article/pii/S0924271616300697.

[196] J. P. McCrae. The lod-cloud. online, 2022. URL https://lod-cloud.net/. (abgerufen: 20.05.2022).

[197] D. L. McGuinness, F. Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[198] M. Mehta. Esp 8266: A breakthrough in wireless sensor networks and internet of things. *International Journal of Electronics and Communication Engineering & Technology*, 6(8): 7–11, 2015.

[199] D. Mellinger, M. Shomin, and V. Kumar. Control of quadrotors for robust perching and landing. In *Proceedings of the International Powered Lift Conference*, pages 205–225, 2010.

[200] P. N. Mendes, M. Jakob, and C. Bizer. *DBpedia: A multilingual cross-domain knowledge base*. European Language Resources Association (ELRA), 2012.

[201] P. Merriaux, Y. Dupuis, R. Boutteau, P. Vasseur, and X. Savatier. A study of vicon system positioning performance. *Sensors*, 17(7):1591, 2017.

[202] P. Milgram and F. Kishino. A taxonomy of mixed reality visual displays. *IEICE TRANSACTIONS on Information and Systems*, 77(12):1321–1329, 1994.

[203] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, and J. Timmis. Automatic property checking of robotic applications. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3869–3876. IEEE, 2017.

[204] S. Mokaram, J. M. Aitken, U. Martinez-Hernandez, I. Eimontaite, D. Cameron, J. Rolph, I. Gwilt, O. McAree, and J. Law. A ros-integrated api for the kuka lbr iiwa collaborative robot. *IFAC-PapersOnLine*, 50(1):15859–15864, 2017.

[205] S. Monk. *Raspberry Pi cookbook: Software and hardware problems and solutions*. O'Reilly Media, Inc., 2016.

[206] R. Murphy and A. P. Family. Usb 101: An introduction to universal serial bus 2.0. *Cypress Application Note AN57294*, 2011.

[207] R. R. Murphy, S. Tadokoro, D. Nardi, A. Jacoff, P. Fiorini, H. Choset, and A. M. Erkmen. *Search and Rescue Robotics*, pages 1151–1173. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-30301-5. doi: 10.1007/978-3-540-30301-5_51. URL `https://doi.org/10.1007/978-3-540-30301-5_51`.

[208] M. A. Musen. The protégé project: a look back and a look forward. *AI matters*, 1(4):4–12, 2015.

[209] G. T. Navajas and S. R. Prada. Building your own quadrotor: A mechatronics system design case study. In *2014 III International Congress of Engineering Mechatronics and Automation (CIIMA)*, pages 1–5. IEEE, 2014.

[210] H. Neuhaus and M. Compton. The semantic sensor network ontology. In *AGILE workshop on challenges in geospatial data harmonisation, Hannover, Germany*, pages 1–33, 2009.

[211] P. P. Neumann. *Gas source localization and gas distribution mapping with a micro-drone*. Bundesanstalt für Materialforschung und-prüfung (BAM), 2013.

[212] K. Nitta and I. Savnik. Survey of rdf storage managers. *DBKDA 2014*, page 156, 2014.

[213] A. Nordmann, A. Tuleu, and S. Wrede. A domain-specific language and simulation architecture for the oncilla robot. In *ICRA 2013 Workshop on Developments of Simulation Tools for Robotics & Biomechanics*, 2013.

[214] J. M. O'Kane. A gentle introduction to ros. online, 2014. URL `http://www.cs.rpi.edu/~trink/Courses/AlgorithmicRobotics/fall2017/agitr-letter.pdf`. (abgerufen: 20.05.2022).

[215] E. Olson. Apriltag: A robust and flexible visual fiducial system. In *2011 IEEE international conference on robotics and automation*, pages 3400–3407. IEEE, 2011.

[216] O. M. G. (OMG). online, 2017. URL `http://www.uml.org/`. (abgerufen: 20.05.2022).

[217] M. Ostanin and A. Klimchik. Interactive robot programing using mixed reality. *IFAC-PapersOnLine*, 51(22):50–55, 2018.

[218] F. Palm and U. Epple. openAAS - Die offene Entwicklung der Verwaltungsschale. In *Automation 2017 : technology networks processes*, volume 2293, pages 103–104. VDI Verlag GmbH, 2017. URL `https://publications.rwth-aachen.de/record/691900`.

[219] R. T. Palomaki, N. T. Rose, M. van den Bossche, T. J. Sherman, and S. F. De Wekker. Wind estimation in the lower atmosphere using multirotor aircraft. *Journal of Atmospheric and Oceanic Technology*, 34(5):1183–1191, 2017.

[220] A. Pamir. How to set up ci/cd for python builds on gitlab. online, 2022. URL `https://www.activestate.com/blog/how-to-set-up-ci-cd-for-python-on-gitlab/`. (abgerufen: 20.05.2022).

[221] I. Panić, J. Ćelić, M. Bistrović, and A. Škrobonja. Drone as a part of maritime search and rescue operations. *Technologies, Techniques and Applications Across PNT*, page 63, 2021.

[222] Y. S. Parihar. Internet of things and nodemcu. *Journal of Emerging Technologies and Innovative Research*, 6(6):1085, 2019.

[223] J. Park, F. Jumu, J. Power, M. Richard, Y. Elsahli, M. A. Jarkas, A. Ruan, A. Luican-Mayer, and J.-M. Ménard. Drone-mountable gas sensing platform using graphene chemiresistors for remote in-field monitoring. *Sensors*, 22(6):2383, 2022.

[224] B. Parsia, N. Matentzoglu, R. S. Gonçalves, B. Glimm, and A. Steigmiller. The owl reasoner evaluation (ore) 2015 competition report. *Journal of Automated Reasoning*, 59(4):455–482, 2017.

[225] K. K. Patel, S. M. Patel, et al. Internet of things-iot: definition, characteristics, architecture, enabling technologies, application & future challenges. *International journal of engineering science and computing*, 6(5), 2016.

[226] M. R. Pedersen, L. Nalpantidis, R. S. Andersen, C. Schou, S. Bøgh, V. Krüger, and O. Madsen. Robot skills for manufacturing: From concept to industrial deployment. *Robotics and Computer-Integrated Manufacturing*, 37:282–291, 2016.

[227] A. Perzylo, J. Grothoff, L. Lucio, M. Weser, S. Malakuti, P. Venet, V. Aravantinos, and T. Deppe. Capability-based semantic interoperability of manufacturing resources: A basys 4.0 perspective. *IFAC-PapersOnLine*, 52(13):1590–1596, 2019.

[228] J. Pfrommer, D. Stogl, K. Aleksandrov, S. E. Navarro, B. Hein, and J. Beyerer. Plug & produce by modelling skills and service-oriented orchestration of reconfigurable manufacturing systems. *at-Automatisierungstechnik*, 63(10):790–800, 2015.

[229] W. D. Pietruszka. *MATLAB® und Simulink® in der Ingenieurpraxis*. Springer, 2012.

[230] Plattform Industrie 4.0. Struktur der verwaltungsschale. online, 2016. URL `www.zvei.org/Publikationen/Struktur-der-Verwaltungsschale.pdf`. (abgerufen: 20.05.2022).

[231] A. M. Potdar, D. Narayan, S. Kengond, and M. M. Mulla. Performance evaluation of docker container and virtual machine. *Procedia Computer Science*, 171:1419–1428, 2020.

[232] D. Potter. Smart plug and play sensors. *IEEE Instrumentation & Measurement Magazine*, 2002.

[233] V. Praveen, S. Pillai, et al. Modeling and simulation of quadcopter using pid controller. *International Journal of Control Theory and Applications*, 9(15):7151–7158, 2016.

[234] S. Profanter, A. Breitkreuz, M. Rickert, and A. Knoll. A hardware-agnostic opc ua skill model for robot manipulators and tools. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1061–1068. IEEE, 2019.

[235] J. Quade and E.-K. Kunst. *Linux-Treiber entwickeln: eine systematische Einführung in die Gerätetreiber-und Kernelprogrammierung-jetzt auch für Raspberry Pi*. Dpunkt. verlag, 2015.

[236] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, 2009.

[237] S. Rabah, A. Assila, E. Khouri, F. Maier, F. Ababsa, P. Maier, F. Mérienne, et al. Towards improving the future of manufacturing through digital twin and augmented reality technologies. *Procedia Manufacturing*, 17:460–467, 2018.

[238] C. Ramirez-Atencia and D. Camacho. Extending qgroundcontrol for automated mission planning of uavs. *Sensors*, 18(7):2339, 2018.

[239] M. Reckhaus, N. Hochgeschwender, J. Paulus, A. Shakhimardanov, and G. K. Kraetzschmar. An overview about simulation and emulation in robotics. *Proceedings of SIMPAR*, pages 365–374, 2010.

[240] M. Reggente and A. J. Lilienthal. Using local wind information for gas distribution mapping in outdoor environments with a mobile robot. In *SENSORS, 2009 IEEE*, pages 1715–1720. IEEE, 2009.

[241] M. Rehbein. Ontologien. In *Digital Humanities*, pages 162–176. Springer, 2017.

[242] W. Reinhold. Elektronische schaltungstechnik. *Leipzig: Fachbuchverlag im Hanser Verlag*, 2010.

[243] D. Reis, B. Piedade, F. F. Correia, J. P. Dias, and A. Aguiar. Developing docker and docker-compose specifications: A developers' survey. *IEEE Access*, 10:2318–2329, 2021.

[244] E. Rescorla and A. Schiffman. Rfc2660: The secure hypertext transfer protocol, 1999.

[245] R. Revetria, F. Tonelli, L. Damiani, M. Demartini, F. Bisio, and N. Peruzzo. A real-time mechanical structures monitoring system based on digital twin, iot and augmented reality. In *2019 Spring Simulation Conference (SpringSim)*, pages 1–10. IEEE, 2019.

[246] F. Richter. *Radio technologies for Smart Homes: ZigBee or EnOcean.* na, 2012.

[247] J.-P. Richter, H. Haller, and P. Schrey. Serviceorientierte architektur. *Informatik-Spektrum*, 28(5):413–416, 2005.

[248] D. M. RICKERT and L. D. F. B. FORTISS. Syntax, semantik und informationsmodelle. online, 2022. URL `https://zentrum-digitalisierung.bayern/wp-content/uploads/190819_Whitepaper_Kapitel8.pdf`. (abgerufen: 20.05.2022).

[249] H. Rijgersberg and J. Top. *UnitDim: an ontology of physical units and quantities.* Agrotechnology & Food Innovations, 2004.

[250] A. Robots. Produktseite: Parallax laser range finder. online, 2022. URL `https://www.active-robots.com/parallax-laser-range-finder.html`. (abgerufen: 20.05.2022).

[251] J. W. Romanishin, K. Gilpin, and D. Rus. M-blocks: Momentum-driven, magnetic modular robots. In *2013 IEEE/RSJ Int. Conf. on Intell. Robots and Systems*, pages 4288–4295, Nov 2013. doi: 10.1109/IROS.2013.6696971.

[252] ROS. Target platforms. online, 2022. URL `https://www.ros.org/reps/rep-0003.html#noetic-ninjemys-may-2020-may-2025`. (abgerufen: 20.05.2022).

[253] M. Rossi and D. Brunelli. Gas sensing on unmanned vehicles: Challenges and opportunities. In *2017 New Generation of CAS (NGCAS)*, pages 117–120. IEEE, 2017.

[254] S. W. Ruehl, C. Parlitz, G. Heppner, A. Hermann, A. Roennau, and R. Dillmann. Experimental evaluation of the schunk 5-finger gripping hand for grasping tasks. In *2014 IEEE International Conference on Robotics and Biomimetics (ROBIO 2014)*, pages 2465–2470. IEEE, 2014.

[255] RWTH Aachen. Openaas: Erster meilenstein gesetzt. online, 2016. URL http://www.openautomation.de/detailseite/openaas-erster-meilenstein-gesetzt.html. (abgerufen: 20.05.2022).

[256] L. Ryzhyk. The arm architecture. *Chicago University, Illinois, EUA*, 2006.

[257] M. Saari, A. M. bin Baharudin, and S. Hyrynsalmi. Survey of prototyping solutions utilizing raspberry pi. In *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 991–994. IEEE, 2017.

[258] K. Sahlmann, A. Lindemann, and B. Schnor. Binary representation of device descriptions: Cbor versus rdf hdt. *Technische Universität Braunschw: Braunschweig, Germany*, 2018.

[259] N. Sariff and N. Buniyamin. An overview of autonomous mobile robot path planning algorithms. In *2006 4th student conference on research and development*, pages 183–188. IEEE, 2006.

[260] J. Scherer, S. Yahyanejad, S. Hayat, E. Yanmaz, T. Andre, A. Khan, V. Vukadinovic, C. Bettstetter, H. Hellwagner, and B. Rinner. An autonomous multi-uav system for search and rescue. In *Proc. of the First Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use*, DroNet '15, pages 33–38, Florence, Italy, 2015. ACM. ISBN 978-1-4503-3501-0. doi: 10.1145/2750675.2750683. URL http://doi.acm.org/10.1145/2750675.2750683.

[261] J. Scherer, S. Yahyanejad, S. Hayat, E. Yanmaz, T. Andre, A. Khan, V. Vukadinovic, C. Bettstetter, H. Hellwagner, and B. Rinner. An autonomous multi-uav system for search and rescue. In *Proceedings of the First Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use*, pages 33–38, 2015.

[262] C. Schierle. Evaluierung und implementierung einer verwaltungsschale für industrie 4.0 komponenten. B.S. thesis, Universität Stuttgart, 2017.

[263] T. Schmickl, R. Thenius, C. Moslinger, J. Timmis, A. Tyrrell, M. Read, J. Hilder, J. Halloy, A. Campo, C. Stefanini, L. Manfredi, S. Orofino, S. Kernbach, T. Dipper, and D. Sutantyo. Cocoro – the self-aware underwater swarm. In *2011 Fifth IEEE Conf. on Self-Adaptive and Self-Organizing Systems Workshops*, pages 120–126, 2011. doi: 10.1109/SASOW.2011.11.

[264] W. Schmidt. Modeling and implementing of industrie 4.0 scenarios. In *Modelling to Program: Second International Workshop, M2P 2020, Lappeenranta, Finland, March 10–12, 2020, Revised Selected Papers*, volume 1401, page 90. Springer Nature, 2021.

[265] J. Scholtz. Theory and evaluation of human robot interactions. In *System Sciences, 2003. Proc. of the 36th Annual Hawaii Int. Conf. on*, pages 10 pp.–, Jan 2003. doi: 10.1109/HICSS.2003.1174284.

[266] M. Schörner, R. Katschinsky, C. Wanninger, A. Hoffmann, and W. Reif. Towards fully automated inspection of large components with uavs: Offline path planning and view angle dependent optimization strategies. In *International Conference on Informatics in Control, Automation and Robotics*, pages 105–123. Springer, 2020.

[267] M. Schörner, R. Katschinsky, C. Wanninger, A. Hoffmann, and W. Reif. Towards fully automated inspection of large components with uavs: Offline path planning and view angle dependent optimization strategies. In *International Conference on Informatics in Control, Automation and Robotics*, pages 105–123. Springer, 2020.

[268] M. Schörner, C. Wanninger, A. Hoffmann, O. Kosak, H. Ponsar, and W. Reif. Modeling and execution of coordinated missions in reconfigurable robot ensembles. In *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, pages 290–293. IEEE, 2020.

[269] M. Schörner, C. Wanninger, A. Hoffmann, O. Kosak, and W. Reif. Architecture for emergency control of autonomous uav ensembles** this work is partly funded by the german research foundation (dfg) under the combo grant. In *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*, pages 41–46. IEEE, 2021.

[270] G. Schroeder, C. Steinmetz, C. E. Pereira, I. Muller, N. Garcia, D. Espindola, and R. Rodrigues. Visualising the digital twin using web services and augmented reality. In *2016 IEEE 14th international conference on industrial informatics (INDIN)*, pages 522–527. IEEE, 2016.

[271] Sensirion. Produktseite: Sht75. online, 2022. URL `https://sensirion.com/de/produkte/katalog/SHT75/`. (abgerufen: 20.05.2022).

[272] T. Shanley. *Plug and play system architecture.* Addison-Wesley Professional, 1995.

[273] R. Shearer, B. Motik, and I. Horrocks. Hermit: A highly-efficient owl reasoner. In *Owled*, volume 432, page 91, 2008.

[274] C. Siepmann and P. Kowalczuk. Understanding continued smartwatch usage: the role of emotional as well as health and fitness factors. *Electronic Markets*, 31(4):795–809, 2021.

[275] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

[276] E. Song, D. Westbrook, and K. B. Lee. A prototype ieee 1451.4 smart transducer interface for sensors and actuators. In *IEEE 2011 10th International Conference on Electronic Measurement Instruments*, volume 1, pages 1–6, 2011. doi: 10.1109/ICEMI.2011.6037665.

[277] E. Sorensen and M. Mikailesc. Model-view-viewmodel (mvvm) design pattern using windows presentation foundation (wpf) technology. *MegaByte Journal*, 9(4):1–19, 2010.

[278] L. Stanisic, A. Legrand, and V. Danjean. An effective git and org-mode based workflow for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):61–70, 2015.

[279] G. Starke. *Effektive Softwarearchitekturen: Ein praktischer Leitfaden.* Carl Hanser Verlag GmbH Co KG, 2015.

[280] J. Südekum. Digitalisierung und die zukunft der arbeit. *Wirtschaftspolitisches Zentrum*, page 1, 2018.

[281] X. Tan, M. Zhou, and B. Fitzgerald. Scaling open source communities: An empirical study of the linux kernel. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1222–1234. IEEE, 2020.

[282] N. Tatarchuk, J. Barczak, and B. Bilodeau. Programming for real-time tessellation on gpu. *AMD whitepaper*, 5(3), 2010.

[283] M. Tenorth and M. Beetz. Knowrob - knowledge processing for autonomous personal robots. In *2009 IEEE/RSJ Int. Conf. on Intell. Robots and Systems*, pages 4261–4266, 2009. doi: 10.1109/IROS.2009.5354602.

[284] R. Thenius, D. Moser, S. Kernbach, I. Kuksin, O. Kernbach, E. Elena Kuksina, N. Mišković, S. Bogdan, T. Petrović, A. Babić, F. Boyer, V. Lebastard, S. Bazeille, G. W. Ferrari, E. Donati, R. Pelliccia, D. Romano, C. Stefanini, M. Morgantin, A. Campo, and T. Schmickl. subclutron: a learning, self-regulating, self-sustaining underwater society/culture of robots. *Art. Life and Intell. Agents Symposium, 2016*, 2016.

[285] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann. A new skill based robot programming language using uml/p statecharts. In *2013 IEEE International Conference on Robotics and Automation*, pages 461–466. IEEE, 2013.

[286] S. Thrun. Probabilistic robotics. *Communications of the ACM*, 45(3):52–57, 2002.

[287] Tinkerforge. Hauptseite. online, 2022. URL https://www.tinkerforge.com/de/. (abgerufen: 20.05.2022).

[288] M. Tölgyessy, M. Dekan, L. Chovanec, and P. Hubinskỳ. Evaluation of the azure kinect and its comparison to kinect v1 and kinect v2. *Sensors*, 21(2):413, 2021.

[289] T. F. Trautner. *Agile Automatisierung von Fertigungszellen: Semiotische Interoperabilität zustandsbehafteter, deterministischer Anlagenkomponenten.* PhD thesis, Wien, 2021.

[290] J. Turnbull. *The Docker Book: Containerization is the new virtualization.* James Turnbull, 2014.

[291] B. Ullmer and H. Ishii. The metadesk: models and prototypes for tangible user interfaces. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 223–232, 1997.

[292] S. B. Vaghani. Virtual machine file system. *ACM SIGOPS Operating Systems Review*, 44(4): 57–70, 2010.

[293] J. Vainikka. Full-stack web development using django rest framework and react. online, 2018. URL https://www.theseus.fi/bitstream/handle/10024/146578/joel_vainikka.pdf?sequence=1. (abgerufen: 20.05.2022).

[294] N. Valcasara. *Unreal engine game development blueprints.* Packt Publishing Ltd, 2015.

[295] A. R. Van Cam Pham, S. Gérard, and S. Li. Complete code generation from uml state machine. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*, volume 1, pages 208–219, 2017.

[296] J. Virtanen. Comparing different ci/cd pipelines. online, 2021. URL https://www.theseus.fi/bitstream/handle/10024/511026/Opinnaytetyo_Joni_Virtanen.pdf?sequence=2. (abgerufen: 20.05.2022).

[297] O. V. Virtualbox. Oracle vm virtualbox. *Change*, 107:1–287, 2011.

[298] E. Vonach. Robot supported virtual and augmented reality. In *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pages 1–2. IEEE, 2018.

[299] S. Voshmgir. *Token Economy: How the Web3 reinvents the Internet*, volume 2. Token Kitchen, 2020.

[300] W3C. Semantic Sensor Network Ontology. online, 2017. URL https://www.w3.org/TR/vocab-ssn/. (abgerufen: 20.05.2022).

[301] W3C. Json-ld 1.1. online, 2022. URL https://www.w3.org/TR/json-ld11/. (abgerufen: 20.05.2022).

[302] W3C. Swol versus wol. online, 2022. URL https://lists.w3.org/Archives/Public/www-webont-wg/2001Dec/0169.html. (abgerufen: 20.05.2022).

[303] W3C. Resource description framework (rdf). online, 2022. URL https://www.w3.org/TR/rdf-concepts/. (abgerufen: 20.05.2022).

[304] W3C. Rdf schema 1.1. online, 2022. URL https://www.w3.org/TR/rdf-schema/. (abgerufen: 20.05.2022).

[305] W3C. Sparql 1.1 query language. online, 2022. URL https://www.w3.org/TR/sparql11-query/. (abgerufen: 20.05.2022).

[306] M. Waibel, B. Keays, and F. Augugliaro. Drone shows: Creative potential and best practices. Technical report, ETH Zurich, 2017.

[307] M. W. Walker and D. E. Orin. Efficient dynamic computer simulation of robotic mechanisms. *Journal of Dynamic Systems, Measurement and Control*, 1982.

[308] M. Wang and J. Cao. A review of collision detection for deformable objects. *Computer Animation and Virtual Worlds*, 32(5):e1987, 2021.

[309] C. Wanninger, C. Eymüller, A. Hoffmann, O. Kosak, and W. Reif. Synthesizing capabilities for collective adaptive systems from self-descriptive hardware devices bridging the reality gap. In *International Symposium on Leveraging Applications of Formal Methods*, pages 94–108. Springer, 2018.

[310] C. Wanninger, R. Katschinsky, A. Hoffmann, M. Schörner, and W. Reif. Towards fully automated inspection of large components with uavs: Offline path planning. In *ICINCO*, pages 71–80, 2020.

[311] C. Wanninger, L. Alfano, M. Schömer, A. Hoffmann, O. Kosak, and W. Reif. Semantic plug and play: an architecture combining linked data and reconfigurable hardware. In *2021 IEEE 15th International Conference on Semantic Computing (ICSC)*, pages 203–206. IEEE, 2021.

[312] C. Wanninger, S. Rossi, M. Schörner, A. Hoffmann, A. Poeppel, C. Eymueller, and W. Reif. Rossi a graphical programming interface for ros 2. In *2021 21st International Conference on Control, Automation and Systems (ICCAS)*, pages 255–262. IEEE, 2021.

[313] J.-D. Warren, J. Adams, and H. Molle. Arduino for robotics. In *Arduino robotics*, pages 51–82. Springer, 2011.

[314] J. Weaver and P. Tarjan. Facebook linked data via the graph api. *Semantic Web*, 4(3): 245–250, 2013.

[315] S. Wijethilaka and M. Liyanage. Survey on network slicing for internet of things realization in 5g networks. *IEEE Communications Surveys & Tutorials*, 23(2):957–994, 2021.

[316] E. Williams. A note on regression methods in calibration. *Technometrics*, 11(1):189–192, 1969.

[317] B. Wolf, C. Chwala, B. Fersch, et al. The scalex campaign: Scale-crossing land surface and boundary layer processes in the tereno-prealpine observatory. *Bulletin of the American Meteorological Society*, 98(6):1217–1234, 2017. doi: 10.1175/BAMS-D-15-00277.1. URL `https://doi.org/10.1175/BAMS-D-15-00277.1`.

[318] D. Wood, M. Zaidman, and L. Ruth. Structured data on the web. *Manning Publications*, 2014.

[319] A. Woolf, A. Haller, L. Lefort, and K. Taylor. Publishing high-quality climate data on the semantic web. In *EGU General Assembly Conference Abstracts*, pages EGU2013–7958, 2013.

[320] P. Wright. Threading. *Beginning Visual C# 2005 Express Edition: From Novice to Professional*, pages 427–440, 2006.

[321] N. Wulf and S. Betz. Daten-ökosysteme wider willen: Herausforderungen des pay-as-you-live-geschäftsmodells im kontext deutscher krankenversicherungen. *HMD Praxis der Wirtschaftsinformatik*, 58(3):494–506, 2021.

[322] X. Xiang, Z. Wang, Z. Mo, G. Chen, K. Pham, and E. Blasch. Wind field estimation through autonomous quadcopter avionics. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–6. IEEE, 2016.

[323] C. Xiao and Z. Lifeng. Implementation of mobile augmented reality based on vuforia and rawajali. In *2014 IEEE 5th International Conference on Software Engineering and Service Science*, pages 912–915. IEEE, 2014.

[324] L. Xue, Y. Liu, P. Zeng, H. Yu, and Z. Shi. An ontology based scheme for sensor description in context awareness system. In *Inform. and Automation, 2015 IEEE Int. Conf. on*, pages 817–820, 2015. doi: 10.1109/ICInfA.2015.7279397.

[325] X. Ye and S. H. Hong. Toward industry 4.0 components: Insights into and implementation of asset administration shells. *IEEE Industrial Electronics Magazine*, 13(1):13–25, 2019.

[326] H. Zhu, P. A. Hall, and J. H. May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.

[327] Y. Zhu, T. Zhang, and J. Song. An improved wall following method for escaping from local minimum in artificial potential field based path planning. In *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, pages 6017–6022. IEEE, 2009.

[328] Z. Zhu, C. Liu, and X. Xu. Visualisation of the digital twin data in manufacturing by using augmented reality. *Procedia Cirp*, 81:898–903, 2019.

Abbildungsverzeichnis

1.1 Übersicht über die simplifizierte Verwaltung von Treibern innerhalb eines Betriebssystems mit exemplarischem Ablauf für das Laden und die Verwendung eines Treibers innerhalb des Programms. (Eigenwerk) 3

1.2 Übersicht der Säulen von Semantic Plug & Play und der Unterteilung der Themenbereiche, sowie die Zuordnung in die Kapitel der Dissertation. (Eigenwerk) 11

2.1 Beispiel einer Roboterzelle mit Verwaltungsschale modelliert in AML (links) und einer exemplarischen Modellierung in einem Graphen (rechts) als abstrakte Veranschaulichung des Semantic Web. Regelwerke für Abhängigkeiten in AML existieren nicht, so kann eine Instanz einer Zelle unter einer Roboterklasse eingefügt werden. (Eigenwerk) 14

2.2 Abbildung unterschiedlicher grafischer Modellierungen für dynamische Abläufe. (1) zeigt eine abstrakte Modellierung in Matlab Simulink mit Hierarchieebenen [229]. In (2) ist ein Blueprint aus der Unrealengine abgebildet mit hervorgehobener Kante für die Visualisierung einer Datenübermittlung zur Laufzeit [294]. Ein abstraktes Aktivitätsdiagramm aus der UML ist in (3) illustriert, während in (4) eine konkrete Missionsplanung in einer Erweiterung der QGroundControl [238] dargestellt wird. (Die Bilder 1,2 und 4 sind aus den Referenzen der Bildbeschreibung entnommen.) 17

2.3 Abbildung eines KUKA Roboters in Blender mit kinematischer Kette in den Achsen und einem letzten Glied am Endeffektor für die inverse Kinematik. Der Verbund aus Modell und kinematischer Ketten kann in Collada-Dateien gespeichert werden. (Rendering und Kinematisierung sind Eigenwerke) 19

2.4 Abbildung drei unterschiedlicher modularer Robotersysteme. Cubelets (links) dienen als Tangible User Interface für einfache Robotersysteme [110]. 360G2 sind selbst assemblierende Würfelsysteme mit lokomotorischen Fähigkeiten [39] und der SMC-Rover besteht aus unterschiedlichen modularen Bauteilen mit eigenen Fähigkeiten [140]. Tinkerforge [287] und Arduino [11] dienen zur Integration und Steuerung von Sensoren und Aktuatoren. (Die Bilder sind den entsprechenden Quellen der Bildbeschreibung entnommen.) 20

3.1 Exemplarischer Chemieunfall innerhalb von Augsburg mit austretender Gaswolke. Die einzelnen Schritte (1)-(3) skizzieren das Vorgehen der Berufsfeuerwehr in solchen Situationen. (Bild stammt aus Google Maps) 24

3.2 Untergliederung der Schritte bei einem Gasunfall in eine exemplarische ScORe-Mission mit zehn konfigurierbaren Quadrocoptern, sowie modularen Sensoren und Aktuatoren. Für die einzelnen Schritte werden jeweils Konfigurationsvorschläge sowie die nötige Rekonfigurationsphasen (R) skizziert. (Bild aus der kooperativen Publikation [153] entnommen.) 25
3.3 Mehrschichtige Systemarchitektur mit Fokus auf Roboter a_1 eines Multipotenten Ensembles A und dessen Interaktionen (durchgezogene Pfeile) mit dem Benutzer sowie anderen Robotern $\{a_2, ..., a_n\}$. Gestrichelte Pfeile symbolisieren eine Instanziierung auf der Ensemble-, Agenten- und Fähigkeits-Schicht. Unterschiedliche Hintergrundfarben stellen Änderungen in den Technologien dar, um Situationen zu bewältigen, auf die das ScORe-Aufgabenmuster angewendet werden kann. (Die Abbildung der Architektur ist aus der kooperativen Publikation [153] entnommen.) 27
3.4 Links ist als Referenz ein statischer Messturm mit drei Messpunkten skizziert. Rechts werden drei virtuelle Messtürme durch auf und absteigende Quadrocopter illustriert, die an den Messpunkten stehenbleiben, um die Temperatur und Luftfeuchte zu messen. (Eigenwerk) 29
3.5 Dieses Experiment verwendet ein über mehrere Quadrocopter gespanntes Lichtwellenleiterkabel, welches an einem Auswertungsgerät an einem mobilen Bodenroboter montiert ist. Durch Synchronisierungsmechanismen kann sich das heterogene Ensemble zur Laufzeit in einer Achse bewegen. (Eigenwerk) 30
3.6 In diesem Experiment wird ein Partikelemitter eingesetzt, um eine Partikelwolke zu erzeugen. Um externe Einflüsse wie die Windstärke und -geschwindigkeit zu messen, werden ein Anemometer und ein Partikelsensor an den Quadrocopter angebracht. (Rendering und Simulation der Gaswolke sind Eigenwerke) 31
3.7 Ein Quadrocopter ist mit einem modularen Distanzsensor ausgestattet und soll über mehrere Hindernisse fliegen. Der Distanzsensor misst dabei den Abstand über Boden, der durch die Hindernisse variiert. (Eigenwerk) 32
3.8 In diesem Experiment soll ein Quadrocopter mit einer Kamera ein Werkstück suchen (1). Nach erfolgreicher Suche wird versucht, über einen Elektromagnet das Werkstück aufzuheben (2). Durch das zusätzliche Gewicht muss die Kamera demontiert werden (3), bevor das Werkstück zum Ausgangspunkt transportiert werden kann (4). (Eigenwerk) 33
3.9 Dieses Experiment hat das Ziel, reale und virtuelle Repräsentation in einer Simulationsumgebung zu berücksichtigen. So sollen Katastrophenszenarien wie ein Chemieunfall simuliert und schrittweise mit realer Hardware getestet werden können. (Eigenwerk) 34

4.1 Übersicht der behandelten Themen in diesem Kapitel mit entsprechenden Unterkategorien und deren jeweiligem Schwerpunkt. So steht beispielsweise im Unterkapitel der Versionierung das Thema *Vorgehensmodelle der Entwicklung* im Fokus. (Eigenwerk) 38

4.2 Entwicklungsstufen des Webs mit einer groben Zeiteinteilung durch maßgebliche Entwicklungen. Das Datum des Web 1.0 bezieht sich auf den Vorschlag von Tim Berners Lee 1989 [19] und der tatsächlichen Implementierung. Die jeweiligen Stufen des Webs ergänzen vorhergehende und sind weiterhin in Verwendung. (Eigenwerk) 39

4.3 Ausschnitt der Schichten des Semantic Web nach Tim Berners Lee, angelehnt an die vier Versionen der Architektur [118]. Additiv befindet sich auf der Syntax Ebene ebenfalls JSON, das von JSON-LD 1.1 verwendet [142] und seit 2020 von der W3C empfohlen wird [301]. (Eigenwerk in Anlehnung an [118]) 39

4.4 Aufbau einer URI nach RFC3986 [20] mit den drei wichtigsten Fragmenten (Schema, Zuständigkeit, Pfad). Das obere Beispiel steht für eine PDF-Datei und das untere für ein GIT-Repository. (Eigenwerk) 40

4.5 Objektdiagramm mit einer Instanz der Klasse Sensor und den Attributen Typ und Genauigkeit. Daneben ist einmal die Serialisierung der Instanz mittels XML und JSON. (Eigenwerk) 41

4.6 Einfache HTML-Seite mit einem Link für die PDF-Datei mit den Spezifikationen des SHT75 Sensors. Über das *script* Tag kann eine JSON-LD Beschreibung eingebettet werden. (Eigenwerk) 41

4.7 Darstellung des Aufbaus eines RDF-Triples in einer gerichteten Graphendarstellung. Darunter ist ein Fragment aus dem JSON-LD-Beispiel exemplarisch aufgeführt. (Eigenwerk) 42

4.8 Beispiel aus Abschnitt 4.1.2 mit Ausschnitten der verwendeten Ontologien. Prinzipiell werden zwei Visualisierungen verwendet, entweder als Klassendiagramm oder in Form eines gerichteten Graphen. Die Beispielinstanz verwendet die Ontologien anhand der in JSON-LD definierten Kürzel gr und pto. (Eigenwerk) 44

4.9 Klassifizierung von Ontologien nach Rehbein et. al. [241] in Abhängigkeit des Bezuges (x-Achse) und der Abstraktion (y-Achse). Die Assoziationen zwischen den Klassifizierungen stehen für eine strukturelle Abhängigkeit. (Eigenwerk in Anlehnung an [241]) 45

4.10 Beispiel für die Realisierung der Fähigkeit *Fliege zu Position* mit einem Quadrocopter. Über die fünf mit einem *(S)* gekennzeichneten Sensoren können sowohl die Lage, als auch die Position bestimmt werden. Die Controller *(C)* verarbeiten die Sensorwerte und Stellwerte und steuern dadurch die 4 Motoren *(A)* (Eigenwerk) 48

4.11 Grobe Struktur des Ablaufs (links) und des Aufbaus (rechts) des in Abschnitt 3.3.1 skizzierten Experiments der mobilen Messtürme. Die Pfeile signalisieren auf der linken Seite des Aufbaus den Ablauf Missionsverteilung und auf der rechten Seite die kontinuierlichen Messdaten des Quadrocopters und SHT75 Sensors. (Eigenwerk) 49

4.12 Aufbau eines nicht modularen Quadrocopters mit einem SHT75 Temperatur- und Luftfeuchtesensors. Die Elemente sind unterteilt in Platinen mit eigenem Mikrocontroller oder Prozessor (ATmega328, ARM) und Platinen ohne programmierbare Schnittstelle wie die Power Platine oder die Breakout Platine. Die linken Assoziationen stehen für elektrische Verbindungen und die rechten für physikalische Schnittstellen. (Eigenwerk) . . . 50
4.13 Pinlayout des SHT7x Sensors von Sensirion mit einem Vorschlag für die Verkabelung an einen Arduino Uno [1]. Da die Sensoren SHT75 und SHT71 einen identischen Aufbau haben, wird hier der Begriff SHT7x verwendet. (Bildquelle [1]) . . . 51
4.14 Kombination eines UML-Verteilungsdiagramms und eines UML-Komponentendiagramms mit Abhängigkeitsbeziehungen. Das verwendete Beispiel korreliert mit dem Aufbau aus Abbildung 4.12. Die konkreten Artefakte sind in exemplarische Komponenten aufgeteilt. Die Komponenten des Controllerartefaktes veranschaulichen eine einfache Schichtenarchitektur. (Eigenwerk) . . . 54
4.15 Kombination des Model View Controller Musters mit dem Model View Viewmodel Muster. Der Zugriff auf das Modell erfolgt über das Viewmodel und die Assoziationen zwischen View und Viewmodel werden mit Data Bindings realisiert. (Eigenwerk) . . . 55
4.16 Exemplarischer Aufbau eines ROS-Systems innerhalb eines UML-Verteilungsdiagramms mit abweichender Syntax. Die auf den Knoten verteilten ROS-Nodes kommunizieren über Topics und Services miteinander, während der ROS-Master die Orchestrierung übernimmt. (Eigenwerk) . 56
4.17 Skizzierung üblicher Verfahrensweisen für GIT-Branches in Kombination mit einer *Continous Integration (CI) Pipeline* in Anlehnung an die Skizze aus einem Blog von Aleks Pamir [220]. Die vier Linien mit release, main, dev und feature zeigen exemplarische Branches mit einem Versionsverlauf, der durch die jeweiligen Kreise symbolisiert wird. Die CI Pipeline bezieht sich dabei nicht auf einen einzelnen Branch, sondern kann beliebig angewendet werden. (Eigenwerk) . . . 59
4.18 Skizze der Kernkomponenten einer Virtualisierung über Virtuelle Maschinen (links) und einer Containervirtualisierung (rechts). (Eigenwerk in Anlehnung an Potdar et al. [231]) . . . 60
4.19 Zusammenfassung der behandelten Themengebiete und Einordnung in die einzelnen Disziplinen. Die Struktur ist nicht als strikte Abfolge der Themengebiete zu betrachten, da sich bspw. die Versionierung über alle Themengebiete erstreckt und sowohl die Annotation als auch die Verteilung auf Versionierungsmechanismen aufbauen. (Eigenwerk) . . 61

5.1 Übersicht der vier Schwerpunkte in diesem Kapitel. Beginnend mit der Definition von Eigenschaften und Fähigkeiten, die mit Code Bausteinen assoziiert werden, über die Modellierung von Prozessen, bis zu einer modularen Simulation mit MR-Bausteinen. (Eigenwerk) . . . 64

5.2 Konzeptionierung des Experiments: *Ausbreitungsmodellierung* mit einer Unterteilung des Gebietes in ein Raster für Messpunkte, die zeitabhängig sind. Die ausgefüllten Rasterelemente sind bereits gemessene Positionen mit einer Angabe der Intensität (links). Das Konzept umfasst die Unterteilung des Quadrocopters und des Sensors in Eigenschaften (E), Fähigkeiten (F) und Eigenschaften von Fähigkeiten (FE). (Eigenwerk) . 66

5.3 Darstellung der Eigenschaften und Fähigkeiten im Experiment der Ausbreitungsmodellierung um das Gewicht zu definieren, sowie dynamischen Eigenschaften, in denen das Kalibrieroffset und die MAC Adresse des Quadrocopters persistiert sind. Der Zusammenschluss aus einem Quadrocopter und einem Sensor wird in ein virtuelles Device gekapselt. (Eigenwerk) 68

5.4 Das Konzeptmodell (links) illustriert die Abhängigkeiten zwischen Device und Property. Die Multiplizitäten der Assoziationen sind als 1 zu * zu interpretieren. Eine Beispielinstanz (rechts) zeigt den Quadrocopter mit Sensor aus Abbildung 5.3. (Eigenwerk) 70

5.5 Erweiterung des internen Aufbaus aus Abbildung 5.3. Eigenschaften ohne Relevanz sind ausgelassen, um den Fokus auf die zusätzlichen Fähigkeiten zu legen. Das Prinzip der Kapselung in Devices ist auf das virtuelle Anemometer erweitert und Abhängigkeiten (A,B,B*,C) sind zwischen einzelnen Fähigkeiten hinzugefügt worden. (Eigenwerk) . . . 72

5.6 Konzept der Fähigkeiten in Kombination mit Eigenschaften (links). Auf der rechten Seite ist eine Beispielinstanz der *Fliege zu Position* Fähigkeit skizziert. Die Parameter werden über Eigenschaften ausgedrückt. Die Unterteilungen (W,S,A) stehen für die Zuordnung der Konzepte zur Struktur (S), Ausführung (A) und der Wissensbasis (W). (Eigenwerk) . 74

5.7 Ontologie des Semantic Hardware Web (SHW) mit der Grundstruktur des Konzeptes und zusätzlichen Konzepten für die Einschränkungen der Verwendung. Das Strukturelement der Devices wird ebenfalls verwendet, um auf GIT-Repositories zu verweisen. Die komplette Ontologie ist online verfügbar. (Eigenwerk) 75

5.8 Trennung des Konzeptmodells in eine objektorientierte Struktur, die anhand instanziierter Klassen Anforderungen (Requirements) definiert, und eines semantischen Netzes, das die Wissensbasis darstellt. Die konkrete Instanz in der Mitte wird anhand der Requirements und den Informationen des semantischen Netzes gebildet. (Eigenwerk) 76

5.9 Verortung der Semantic Hardware Web (SHW) Ontologie in das Schema von Rehbein et al. [241] aus Abbildung 4.9. Durch die Schnittstellen zu den Verknüpfungen zur Domäne sowie der Integration von Aufgaben durch Fähigkeiten befindet sich die SHW Ontologie im Zentrum. Über die SHW und die Konzepte von Semantic Plug & Play müssen Anwendungen keine eigene Ontologie besitzen. (Eigenwerk in Anlehnung an Rehbein et al. [241]) 76

5.10 Zugriffshierarchie über adaptierte Hardware (rechts). Treiber bilden die Basis für Fähigkeiten, die im Programmcode verwendet werden können. Teile des semantischen Netzes können auf dem Adapter bereitgestellt und durch Referenzen im Internet komplettiert werden (links). (Eigenwerk) 77

5.11 Abhängigkeiten zwischen Modell und Original mit Änderungen nach Kastens et al. [138], sowie exemplarische Modellierungswerkzeuge. Die roten, gestrichelten Linien sind Ergänzungen für die modellgetriebene Entwicklung. (Eigenwerk in Anlehnung an Kastens et al. [138]) 79

5.12 Links ist das Modell der Abhängigkeiten von Skills nach Pfrommer et al. [228] dargestellt. Rechts findet sich eine exemplarische Zuordnung der Modelleinordnung. (Eigenwerk in Anlehnung an Pfrommer et al. [228]) 80

5.13 Aufbau und Ablauf des Experiments der synchronen Glasfasermessung. Der Ablauf wird aus der Vogelperspektive betrachtet und die Bewegung in einzelne Interpolationspunkte unterteilt (1,2,n). Die Fähigkeiten werden weiter abstrahiert in *Bewege* statt *Fliege zu Position*, damit die Fähigkeit auch für den Bodenroboter eingesetzt werden kann. Der Hardwareaufbau dient einer Verortung der Softwareartefakte. (Eigenwerk) . 81

5.14 UML-Aktivitätsdiagramm für den Kontrollfluss eines Quadrocopters im Experiment auf abstrakter Ebene. Der Unterbrechungsbereich (gestrichelt) kann durch das Ereignis *Abbruch* extern verlassen werden. Ereignisse sind globale Aktionen und können von einer Kontrollstation an alle Quadrocopter gesendet werden. Das *Aufsteigen* und *Absteigen* ist bewusst als Aktion modelliert, da ein detaillierteres Modell nicht vorgesehen ist. (Eigenwerk) 81

5.15 Die Schichtenarchitektur aus Abschnitt 3.2 angewendet auf das Experiment, mit einfacher Schnittstelle der für Agenten relevanten Fähigkeiten. Die internen Abläufe mit separaten Fähigkeiten sind im rechten Teil des Diagramms angedeutet. (Eigenwerk) 82

5.16 Technologiestack für die Implementierung und Modellierung von Fähigkeiten. Prozesse können sowohl in objektorientierten Programmiersprachen, als auch mit ROS oder den Erweiterungen ROSSi und der Block Definition Language (BDL) implementiert werden. Der Stack mit seinen optionalen Schichten ist von oben nach unten zu lesen, so kann ein durchgängiger Ansatz bspw. rein objektorientiert erfolgen. (Eigenwerk) 83

5.17 Erweiterung des Konzeptmodells aus Abbildung 5.6 um Commands für die Ausführung von Fähigkeiten (links). In der Mitte ist Pseudocode für die Implementierung eines Sensors auf der Hardwareseite und die dazu assoziierten Graphen aus Eigenschaften (rechts) skizziert. (Eigenwerk) 84

5.18 UML-Deploymentdiagramm in Kombination mit einem Komponentendiagramm für die Verteilung der ROS-Nodes. Der Umfang beschränkt sich auf die Realisierung der Fähigkeit *Bewegen*, das über MAVRos an den Flugcontroller der Quadrocopter geschickt wird. (Eigenwerk) . . . 85

5.19 Prozessmodellierung eines Quadrocopters mit Sensor. Dabei wird ein UML-Aktivitätsdiagramm durch Stereotypen erweitert, um die unterschiedlichen Aktionen zu verdeutlichen. Die Zeitschranke ist für das Verständnis zusammengefasst, kann jedoch auch mit der UML-Syntax modelliert werden. (Eigenwerk) 87
5.20 Der Launchfile-Editor (links) bietet neben der Infrastruktur für ROS-Nodes auch eine Anzeige aktueller Werte der Topics zur Laufzeit, das durch den *Bewege* Befehl angedeutet ist. Der Node-Editor hingegen stellt Codefragmente für die Werkzeuge von ROS bereit, wie bspw. eine Action. (Eigenwerk) 88
5.21 Verteilung der Softwarebausteine zwischen den physisch zusammenhängenden Geräten. Die Ensembleschicht aus Abbildung 3.3 steuert die Agenten, welche über den Semantic Controller die Fähigkeiten der Semantic Adapter kommandieren. (Eigenwerk) 89
5.22 Unterteilung komplexer und einfacher Hardware anhand der verwendeten Adapter und exemplarische Schnittstellen und Programmiersprachen. Als Beispiel für einfache Hardware wird der SHT75 [271] und für komplexe Hardware die Microsoft Kinect [288] verwendet. (Eigenwerk) 89
5.23 Aufbau des physischen Messcopters aus Abbildung 5.21 mit nur einem Temperatursensor und den Softwareartefakten *Semantic Controller* und *Semantic Adapter*. Die schwarzen Punkte symbolisieren den Übergang eine logischen in eine physische Schnittstelle und vice versa. Die mit GIT gekennzeichneten Artefakte werden in den Teilgraphen referenziert. (Eigenwerk) 90
5.24 Definition der Mixed Reality nach Milgram et al. [202] durch das *Virtuality Continuum*. Der bekannte Begriff Augmented Reality im Zusammenhang mit 3D Brillen wird durch den Begriff Mixed Reality subsumiert. (Eigenwerk in Anlehnung an Milgram et al. [202]) 92
5.25 Digitaler Zwilling eines Quadrocopters mit einem 3D Modell das als Eigenschaft bereitgestellt wird, sowie der Fähigkeit *Bewege* für die Simulation. (Eigenwerk) 93
5.26 Konzept der MR-Bausteine in Kombination mit echten Quadrocoptern und dem Nutzer. Der Nutzer trägt eine 3D-Brille, um die Hologramme der digitalen Zwillinge zu sehen. Das Bezugskoordinatensystem gilt für alle physischen und visualisierten Objekte. (Eigenwerk) 94
5.27 Exemplarischer Aufbau einer gemischten Simulation, bestehend aus einem Messcopter mit realem Quadrocopter und zwei simulierten TEMOD Sensoren. Das Postfix DT steht für Digitaler Zwilling. (Eigenwerk) . . . 95
5.28 Methodologie für die Bereitstellung selbstbeschreibender Hardware innerhalb des Semantic Plug & Play beginnend bei der Annotation von Eigenschaften mittels des Semantic Hardware Web über die Integration der Fähigkeiten bis hin zu dedizierten APIs, der Modellierung von Prozessen im Code oder anhand abstrakter Modelle, sowie die schrittweise Realisierung in einer modularen Mixed Reality Simulation. (Eigenwerk) 96

6.1 Kapitelübersicht der Architektur, die sich über den Semantic Adapter und Controller bis zur OOD-API erstreckt. Für eine Einordnung ist im rechten Teil der Abbildung der Stack aus Abbildung 5.16 skizziert. (Eigenwerk) 98
6.2 Das Experiment Pick & Place wird in vier Abschnitte unterteilt, die von links nach rechts aufgeführt sind. Das Areal ist für die Suche (1) in einzelne Felder unterteilt, die mit der Kamera abgesucht werden. Nach dem Wiegen (2) des Werkstückes wird eine Überschreitung der zulässigen Gesamtlast festgestellt und die Kamera muss demontiert werden (3), damit der Transport (4) gelingt. (Eigenwerk) 98
6.3 Das im Experiment verwendete Device besteht aus einem Quadrocopter mit einer Kamera und einem Elektromagneten. Die virtuelle Fähigkeit des Devices besteht aus der Berechnung des Gesamtgewichtes aus der Summe aller Einzelgewichte physischer Elemente sowie dem gemessenem Gewicht der Waage (W), einer kombinierte Fähigkeit. Die Kamera hat eine integrierte Markererkennung, wobei Marker für die Identifikation von Objekten in einer dynamischen Eigenschaft gespeichert sind. (Eigenwerk) 99
6.4 Die Model-Domain-Domainmodel (MDDM)-Architektur orientiert sich an der MVVM Architektur aus Abschnitt 4.2.3. In der Domain werden die Requirements definiert, die im Domainmodel mit Commands gebunden werden. Als Austauschformat für die RDF-Graphen im Model ist JSON-LD vorgesehen und der Austausch findet über ein eigenes Protokoll statt: das Self Descriptive Protocol (SDP). Die Instanz mit den gebundenen Commands wird in einer JSON Serialisierung über das Commands Protocol (ComP) der Domain zur Verfügung gestellt. (Eigenwerk) . . . 100
6.5 Auszug aus dem Klassendiagramm der OOD-API mit dem Model, bestehend aus den Strukturelementen Device, Capability und Property sowie der Schnittstellenimplementierung für das *Commands Protocol* in Form des ThinClients. (Eigenwerk) 102
6.6 In der Semantic Hardware Web Ontologie finden sich exemplarische Individuals, die konkrete Hardware beschreiben. Ein Auszug ist in dem Graphen zu sehen, wobei nur die Knoten mit dicker Umrandung näher beschrieben sind. Eckige Boxen stellen Literale dar. (Eigenwerk) 103
6.7 UML-Aktivitätsdiagramm für die Visualisierung des Kontrollflusses zwischen OOD-API, Semantic Controller und Adapter. Protokollspezifische Nachrichten werden als Signale mit einer gestrichelten Linie zwischen Sende und Empfangssignalen modelliert. Es wird nur ein exemplarischer Ablauf beschrieben, der beim Senden des ersten Kommandos endet. (Eigenwerk) 105
6.8 Ausschnitt aus der Semantic Hardware Web Ontology mit dem Fokus auf virtuelle Fähigkeiten und der Möglichkeit, Fähigkeiten zu kombinieren. Das Prädikat *hasSource* beschreibt den Typ des *xsd_string* über *sourceContent*. (Eigenwerk) 108

6.9 Designklassendiagramm für abstrakte Fähigkeiten nach dem Erzeugungsmuster *Abstract Factory*. Die Klasse *UCController* symbolisiert den Anschluss an die restlichen Klassen des Gesamtsystems. Vererbte Methoden des Interfaces sind nicht skizziert. (Eigenwerk in Anlehnung an die Ergebnisse der betreuten Abschlussarbeit von Moritz Hofer, Student ISSE.) . . . 109

6.10 Beispiel für die Fähigkeit der Waage des Pick & Place-Experiments. Auf der untersten Ebene befinden sich Hardwarefähigkeiten, die über zwei Iterationsschritte zur virtuellen Fähigkeit *Messen* mit der Eigenschaft *Gewicht* zusammengesetzt werden. (Eigenwerk) . . . 110

6.11 Verteilung der Softwareartefakte auf einen Mikrocontroller und einen Einplatinencomputer, die über UART kommunizieren. Die jeweils verwendeten Protokolle sind einerseits das Self Descriptive Protocol (SDP) und das Commands Protocol (ComP). Der Elektromagnet (MA) wird von einem Mikrocontroller adaptiert, währen der Quadrocopter mit einem Einplatinencomputer verbunden ist. (Eigenwerk) . . . 112

6.12 Beispiel eines Protokollablaufs anhand eines UML-Sequenzdiagramms. Die Softwareartefakte Semantic Controller und Adapter sind auf einen Einplatinencomputer und einen Mikrocontroller verteilt. (Eigenwerk) . 113

6.13 Übersicht der MDDM-Architektur mit Verortung der APIs für die Integration der Selbstbeschreibung. Das Model besteht aus verteilten Teilgraphen, die im Domainmodel mit den Anforderungen der Domain anhand einer Blaupause gemappt werden. Unten im Bild sind die verwendeten Protokolle für die Kommunikation zwischen den Artefakten skizziert. (Eigenwerk) . . . 116

7.1 Methodologie von Semantic Plug & Play mit einer Zuordnung der Werkzeuge. Rechts abgebildet ist ein Ausschnitt des Technologiestacks aus Abbildung 5.16, der in diesem Kapitel eingehender vorgestellt wird. (Eigenwerk) . . . 119

7.2 Iterativer Ablauf für die semantische Annotation in Anlehnung an das Wasserfallmodell [168]. Das Modell ist keine strikte Vorgabe der Phasen, sondern soll als Übersicht für den allgemeinen Ablauf dienen. (Eigenwerk) 121

7.3 Abbildung des in SemanticSens umgesetzten iterativen Ablaufes mit Ausschnitten aus dem Frontend der Applikation. (Das Frontend stammt aus der betreuten Abschlussarbeit von Florian Timter, Student am ISSE) 122

7.4 Exemplarischer Aufbau zusammenhängender ROS-Nodes, die über den Semantic Adapter gekapselt und mit einer Selbstbeschreibung (SB) ergänzt sind. Die Kommunikation der Nodes findet über ROS statt. (Eigenwerk) . . . 124

7.5 Aufbau des Experiments mit erwarteter Trajektorie, die durch die Hindernisse beeinflusst wird. Die Positionsabweichung während des Fluges wird mit Δ x und Δ z ausgedrückt unter der Bedingung, dass der relative Abstand von einem halben Meter zum Boden während des Fluges eingehalten wird. Das externe Trackingsystem registriert die Marker und bestimmt die Position für die Auswertung. (Eigenwerk) 125
7.6 Übersicht der eingesetzten Einplatinencomputer und Mikrocontroller für das Experiment. Die ROS-Nodes befinden sich auf einem PC, der über WLAN mit dem Quadrocopter kommuniziert. Intern kommunizieren die Nodes über TCP-Sockets. (Eigenwerk) 126
7.7 Aufbau der Kernfunktionalität von ROSSi durch die abstrakte Klasse *InteractiveGraphicsView*, von der die Typen *RosNodeEditor*, *LaunchFileEditor* und das *LiveDiagram* erben. (Eigenwerk) 126
7.8 Aufbau des *ROS-Node-Editor* mit unterschiedlichen Entitäten. Die Entitäten bilden sowohl grafische Elemente als auch ein Codeskelett für die Codegenerierung. (Eigenwerk) 127
7.9 Nachbildung eines Screenshots des Node-Editors in einer Vektorgrafik für eine bessere Auflösung. Der Node-Editor zeigt eine MAVRos-Node mit einer Subscriber und zwei Publisher Entitäten. Über einen Doppelklick kann der Code einer Entität in externen IDEs bearbeitet werden. (Eigenwerk) 128
7.10 Ausschnitt einer in ROSSi erstellten Launchfile im Launchfile-Editor in Vektorgrafik. Das Modell zeigt eine nicht finalisierte Integration des Quadrocopters mit einer Steuerung über eine MAVLINK-Bridge. Der Flugcontroller des Quadrocopters heißt *Autoquad.* (Eigenwerk) 129
7.11 Die ROS-Node *processdef* bietet über zwei Publisher die Topics */pos* und */dgn* an, wobei die Node *autoquad_pos_vel_target* nur das */pos* Topic über einen Subscriber abonniert hat. (Eigenwerk) 130
7.12 Die unterschiedlichen Elemente der DMDE mit einer grafischen Repräsentation durch an die UML angelehnte Aktivitätsdiagramme. Konkret werden vier Fälle vorgestellt: Die parallele Ausführung, Fehlerbehandlungen, Alternierungen und Erweiterungen über Plugins. (Eigenwerk) 132
7.13 Auszug aus der Modellierung eines Sensortausches für das Experiment des sensorbasierten Fluges. Die Launchfile für die Verteilung und der Flug der Drohne ist mit ROSSi modelliert, der Konfigurationsprozess hingegen ist in DMDE verankert. (Eigenwerk) 135
7.14 Einfache Modellierung einer Mission, die eine in einzelne Positionen unterteilte Trajektorie über Barrieren zwischen verschiedenen Elementen synchronisiert. (Eigenwerk) 136
7.15 Struktureller Aufbau der Simulation als zentrales Element mit unabhängiger Visualisierung, sowie dezentrale Beschreibung aller 3D-Modelle über die Selbstbeschreibung. (Eigenwerk) 138
7.16 Struktureller Aufbau der Simulation als zentrales Element mit unabhängigen Visualisierungen, sowie dezentrale Beschreibung aller 3D-Modelle über die Selbstbeschreibung. (Eigenwerk) 139

7.17 3D-Modell eines Quadrocopters, das sich aus mehreren Meshes zusammensetzt. Die *Base* bildet das statische Grundmodell, die jeweils 2 schwarzen und orangen Rotoren rotieren um die nach oben gerichtete Z-Achse. (Eigenwerk) 140
7.18 Zwei simulierte Quadrocopter mit jeweils einer Sphere als Kollisionshülle. Der euklidische Abstand wird anhand der jeweiligen Rotationsachsen (Base) gemessen. (Eigenwerk) 142
7.19 Bild einer Überlagerung der Realität mit einer Microsoft Hololens und dem digitalen Zwilling des Quadrocopters und zwei Häusern (Infrastruktur). (Eigenwerk) 143
7.20 Struktureller Aufbau der Überlagerung aus Abbildung 7.19. Ein Bezugskoordinatensystem dient als gemeinsame Basis für die Transformationen zwischen den MR-Bausteinen, der Hololens und dem Trackingsystem. (Eigenwerk) 144
7.21 QR-Code, der als Vuforia-Tag verwendet werden kann und eine Referenz auf die Selbstbeschreibung des MR-Bausteins kodiert. In diesem Beispiel ist die Übersichtsseite von Semantic Plug & Play kodiert. (Eigenwerk) . 145
7.22 Einordnung der Werkzeuge in die Methodologie von Semantic Plug & Play. (Eigenwerk) 146

8.1 Eingliederung der Experimente in das ScORe-Schema, die Umgebung, den verwendeten Technologiestack und den Hauptfokus. (Eigenwerk) . 150
8.2 Explosionszeichnung des Forschungsquadrocopters mit der Grundausstattung für Indooranwendungen. Das von EMCOTEC gefertigte Signalboard ist rechts detailliert abgebildet. Das ISSE symbolisiert, dass das Institut entweder an der Entwicklung beteiligt war (Signal-, Powerboard), oder dass jeweilige Teile gänzlich im Rahmen der Dissertation entstanden ist (Topplate, Viconmount). (Eigenwerk) 151
8.3 Fotos der Indoor- und Outdoorvariante des Forschungsquadrocopters. Die Indoorvariante nutzt Vicon-Marker für die Positionssteuerung, während die Outdoorvariante eine GPS-Antenne verwendet. Verschiedene aus CFK gefertigte Adapterplatten unterhalb des Quadrocopters erlauben bspw. die Montage eines Odroid XU4-Einplatinencomputers. (Eigenwerk) 153
8.4 In der Abbildung links ist der strukturelle Aufbau der Flugarena skizziert, mit 16 Kameras, die an einem Traversensystem angebracht sind. Das rechte Bild zeigt die möglichen Markerpositionen der Topplate und Seitenteile des Indoor-Forschungsquadrocopters. (Eigenwerk) 154
8.5 Abbildung des Konzepts für den Prototypen und die tatsächliche Realisierung mit Erweiterungen. Dieser Prototyp ist in fünffacher Ausfertigung realisiert, um eine Redundanz für Ausfälle während der ScaleX Messkampagne zu gewährleisten. (Eigenwerk) 156
8.6 Aufbau des SHT75 Sensors in einem weißen Rohr, das unter den Rotoren des Quadrocopters befestigt wurde. Die Messfläche des Sensors befindet sich innerhalb des Rohres. (Eigenwerk) 157

8.7 Auswertung eines synchronen Messfluges mit drei Quadrocoptern in der ScaleX-Kampagne. Die Messwerte beziehen sich auf die zwei SHT75-Sensoren mit mehreren Messungen in den jeweiligen Höhen. (Bild entnommen aus der kooperativen Publikation [153]) 158
8.8 Abbildung der mobilen Messstation mit modularem Hardwareadapter über die USB-C Schnittstelle. Die Platine führt die Anschlüsse eines ESP-Moduls nach außen und besitzt einen Anschluss für einen Informationsbildschirm, der oberhalb des USB-C-Anschlusses montiert werden kann. (Fotos aufgenommen von Martin Schörner, ISSE) 159
8.9 Mission der einzelnen Quadrocopter in der BDL modelliert. Synchronisationspunkte sind durch Barrieren mit einer Timeoutfunktion realisiert. Der verwendete Technologiestack der Anwendung ist rechts abgebildet. (Eigenwerk) 160
8.10 Abbildung der Verknüpfung zwischen der SSN und der Semantic Hardware Web Ontology. (Eigenwerk) 160
8.11 Aufnahmen des Experiments während der Messkampagne ScaleX. Der Prototyp *Glasfaser* startet auf einer Bodenmatte und das LWL-Kabel wird für den Start über die Ablagen zum Prototyp *DTS* geführt. (Die Fotos der Prototypen wurden von Stefanie Seubert, Universität Augsburg, aufgenommen.) 162
8.12 Aufnahme und Beschreibung des Prototypen *DTS* mit einer Kalibrierwanne, in der für den Versuch die DTS-Spindel gelagert ist. (Das Foto des Prototypen wurde von Stefanie Seubert, Universität Augsburg aufgenommen.) 163
8.13 Die Aufnahme zeigt den Prototypen *Glasfaser*, der das LWL-Kabel in einer Halterung trägt. Als Referenzsensoren für die spätere Analyse finden sich zwei Steckplätze an den Rotorstegen für modulare Temperatursensoren. Das in den Temperatursensoren verwendete STM-Verteilerboard bietet sechs Anschlüsse für UART und jeweils einen SPI- und I2C-Anschluss. (Das Foto des Prototypen wurden von Stefanie Seubert, Universität Augsburg, aufgenommen. Renderings und Beschreibung sind Eigenwerke.) 164
8.14 Die linke Grafik zeigt die Temperaturmessungen (z) der DTS Auswertungseinheit entlang des LWL-Kabels (x) im Verlauf der Zeit (y) während des Experiments. Die farbliche Markierung symbolisiert zusätzlich den Temperaturgradienten. Die rechte Abbildung illustriert den Aufstieg aus einer anderen Perspektive, in der die Temperaturkurve zwischen dem Start und der Landung der Mission visualisiert ist. (Die Grafiken sind der kooperativen Publikation [153] entnommen.) 165
8.15 Die linke Grafik zeigt die grafische Benutzeroberfläche der Schwarmkontrollstation auf einem im mittleren Bild dargestellten DGX-Funkgerät an einer 3D-gedruckten Adapterplatte. Rechts sind RGB-LEDs für ein visuelles Feedback an den Drohnen montiert (Die Fotos wurden von Martin Schörner, ISSE, aufgenommen.) 166

8.16 Feldexperiment in der Nähe der Universität Augsburg mit einem Rauchemitter (Mr Smoke 4) an einer mobilen Roboterplattform (INNOK Basis) und einem Quadrocopter mit Aerosolsensor (Aerosolcopter). (Eigenwerk) 168
8.17 Aufbau des Aerosolcopter-Prototypen mit einem ESP 8266-Modul als Adapter für den Waveshare Aerosolsensor und den Autoquad M4. Das Bild rechts ist eine grafische Darstellung des Sensormesswertes zur Laufzeit mit einem adaptiven Schwellwert. (Das Foto stammt aus der betreuten Abschlussarbeit von Timo Peters, Student am ISSE) 169
8.18 Screenshot der grafischen Benutzeroberfläche für die Windmessung (A, AL), Ausbreitungsmodellierung (CL), Gasquellenlokalisierung (B, BL) mit unterschiedlicher Parametrisierung, und Messwerte (M). Die verwendeten Parameter der Gasquellenlokalisierung: (B): Adaptilität: 0,05; Gradient: 1 für kein Aerosol entdeckt (BL): Adaptilität: 0,4; Gradient: 10 für kein Aerosol entdeckt (Die Benutzeroberfläche stammt aus der betreuten Abschlussarbeit von Timo Peters, Student am ISSE) 171
8.19 Zeitraffer des sensorbasierten Fluges in einer Indoor-Arena mit dem Vicon-Trackingsystem. Die Hindernisse bestehen aus Kisten. (Das Foto stammt aus der Abschlussarbeit von Christian Eymüller, ISSE) 172
8.20 Abbildung der modularen Sensorik für den Prototypen *Modular*, der eine Montageplatte für zwei Sensoren bietet. Als Basis für die Sensoren wird ein Einplatinencomputer verwendet, auf dem die vier unterschiedlichen Sensoren angebracht und somit die Selbstbeschreibungen lokal an den Sensoren verfügbar gemacht werden können. (Die Fotos sind von Christian Eymüller im Rahmen einer Abschlussarbeit aufgenommen worden, ISSE) 174
8.21 Die Graphen zeigen die Auswertungen der Versuche des Gradientenflugs, sowie die Abschätzung der Flugzeit. Für die Positionsdaten wurde das Vicon-Trackingsystem verwendet. Für die Messung des Stromverbrauchs befindet sich ein spezieller SMD IC-Baustein auf der Powerplatine des Forschungsquadrocopters. (Die Auswertungen der Versuche stammen aus der betreuten Abschlussarbeit von Christian Eymüller, ISSE) 175
8.22 Versuchsaufbau in der großen Flugarena mit Vicon-System. Das Experiment ist in vier Abschnitte unterteilt, die sequentiell ausgeführt werden. (Eigenwerk) 178
8.23 Abbildung des Montagesystems und aller im Experiment verwendeter modularer Bauteile in einer Explosionszeichnung. Die montierte Variante, wie sie auch in Abbildung 8.22 in natura abgebildet ist, befindet sich rechts im Bild. Der verwendete Technologiestack ist nicht nur auf die modularen Bauteile angewendet sondern impliziert auch das Trackingsystem, den Quadrocopter und einen PC. (Eigenwerk) 179
8.24 Screenshot der Pixy Bildauswertung [42], in der das Papier als Ziel erkannt und umrandet wird. Daneben ist ein Anflugversuch für das Wiegen des Werkstücks illustriert. (Das linke Bild ist aus der grafischen Oberfläche des Programms *Pixymon v2* entnommen.) 180

8.25 Aufbau des Experiments *GoLive* in der großen Flugarena mit insgesamt acht MR-Bausteinen und einem Quadrocopter. Zwei der MR-Bausteine sind mit Vicon-Markern ausgestattet, um ein Bezugskoordinatensystem zwischen der Hololens und dem Vicon-System herzustellen. (Eigenwerk) 183
8.26 Das 3D-Modell für den MR-Baustein ist 3D-gedruckt und besteht aus einem Touchscreen und einem Raspberry Pi 4, das mit einer externen Powerbank betrieben werden kann. Der Vicon-Referenz-MR-Baustein ist mit vier Vicon-Markern ausgestattet und erlaubt eine Bestimmung der Position und Rotation (Frame) innerhalb des Vicon-Systems. (Eigenwerk) 185
8.27 Die Gaswolkensimulation basiert auf einem Partikeleffekt mit Kegelausbreitung. Sobald ein simulierter Gaspartikel im Messbereich erkannt wird, ist im simulierten Sensor ein Messwert zu verzeichnen, dessen Intensität anhand der Anzahl erkannter Gaspartikel ab- und zunimmt. Sowohl der simulierte als auch der reale Quadrocopter sind mit simulierten Gassensoren ausgestattet. (Eigenwerk) 186
8.28 Einordnung der Experimente in die Themengebiete der Fallstudie, das ScORe-Schema und die Forschungsergebnisse von Semantic Plug & Play. Die Einordnung in die Themengebiete der Fallstudie erstrecken sich über mehrere Experimente, wobei die Legende im unteren Bereich Kürzel darstellt, die den einzelnen Experimenten konkret zugeordnet sind. (Eigenwerk) 189
8.29 Zusammenfassung aller Experimente in einer Tabelle. Die Legende im unteren Bereich fasst die verwendeten Kürzel zusammen, die in den Spalten *Experiment* und *Hardware* verwendet werden. (Eigenwerk) . . 191

9.1 Übersicht über alle Projekte der vorliegenden Dissertation, die entweder Open-Source verfügbar, oder kostenlos nutzbar sind. In der Übersichtsseite *http://semantic-hardware.com/* sind alle Projekte verlinkt. (Eigenwerk) 194
9.2 Halbschale eines Flugzeughecks, das von Premium Aerotec für das Projekt *Quality Assurance with Drones* (QuAD) in der großen Flugarena bereitgestellt wurde. Das Projekt behandelt eine autonome Drohneninspektion von Bauteilen ohne externe Trackingsysteme. (Eigenwerk) . . 196

Betreute Abschlussarbeiten

- Julian Hanke. „Aggregation von Sensordaten zur Objektlokalisierung mit einem Quadrocopter“. Bachelorarbeit. Universität Augsburg, 2016
- Martin Schörner. „Konzept eines Systems zur Überwachung und Steuerung mobiler Roboterschwärme mit prototypischer Evaluation“. Bachelorarbeit. Universität Augsburg, 2017
- Jakob Scholz. „Lokale Hinderniserkennung und Kollisionsvermeidungsstrategien bei Quadrocoptern“. Bachelorarbeit. Universität Augsburg, 2017
- Fritz Luber. „Aufbau und Evaluation eines Multicopters mit Pixhawk-Flugsteuerung“. Bachelorarbeit. Universität Augsburg, 2017
- Christian Eymüller. „Semantische Annotation von Sensoren und Aktuatoren für die automatische Interpretation und Reaktion adaptiver Systeme“. Masterarbeit. Universität Augsburg, 2017
- Lukas Lodes. „Maschinelles Lernen von Regelungsparametern in Multikoptern“. Bachelorarbeit. Universität Augsburg, 2018
- Raphael Katschinsky. „Drohnen Inspektion großer Bauteile - Trajektorienplanung mit vollständiger Sichtabdeckung“. Bachelorarbeit. Universität Augsburg, 2019
- Moritz Vogelsang. „Drohnen-Inspektion großer Bauteile - Relative Orientierung zur Laufzeit“. Bachelorarbeit. Universität Augsburg, 2019
- Martin Schörner. „Smart Component Framework - A ROS-based toolkit for modular hardware in flying robot ensembles“. Masterarbeit. Universität Augsburg, 2019
- Luca Alfano. „OOD Framework für selbstbeschreibende, modulare Hardware mit dezentralen Semantic Web Mechanismen“. Bachelorarbeit. Universität Augsburg, 2020
- Timo Peter. „Teilautonome Gasquellenlokalisierung und Ausbreitungsanalyse mit Multicoptern“. Bachelorarbeit. Universität Augsburg, 2020
- Sebastian Rossi. „ROSSi - Eine Graphische Programmierschnittstelle für ROS 2“. Bachelorarbeit. Universität Augsburg, 2021
- Thomas Badem. „Go Live: Inkrementelle Überführung simulierter Roboteranwendungen in die Realität mit Mixed Reality“. Bachelorarbeit. Universität Augsburg, 2021
- Moritz Hofer. „Combined Capabilities - Softwarebausteine für hardwareübergreifende Fähigkeiten“. Bachelorarbeit. Universität Augsburg, 2021
- Lukas Schlachter. „Rescuemender - proaktive Empfehlung zur Geräteauswahl bei Feuerwehr Einsätzen“. Bachelorarbeit. Universität Augsburg, 2021
- Florian Timter. „SemanticSens - Eine webbasierte Anwendung für semantische Annotationen“. Bachelorarbeit. Universität Augsburg, 2021

Eigene Publikationen

1. Constantin Wanninger, Christian Eymueller, Alwin Hoffmann and Wolfgang Reif. „Synthesizing Capabilities for Collective Adaptive Systems from Self-descriptive Hardware Devices - Bridging the Reality Gap“. In book: 2018 Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems DOI: 10.1007/978-3-030-03424-5_7
2. Christian Eymueller, Constantin Wanninger, Alwin Hoffmann and Wolfgang Reif. „Semantic Plug and Play - Self-Descriptive Modular Hardware for Robotic Applications“. Conference: 2018 International Journal of Semantic Computing 12(04) DOI: 10.1142/S1793351X18500058
3. Oliver Kosak*, Constantin Wanninger*, Alwin Hoffmann, Hella Ponsar and Wolfgang Reif. „Multipotent Systems: Combining Planning, Self-Organization, and Reconfiguration in Modular Robot Ensembles“. MDPI: 2018 Sensors DOI: 10.3390/s19010017 *equal contribution
4. Constantin Wanninger, Luca Alfano, Martin Schörner, Alwin Hoffmann, Oliver Kosak and Wolfgang Reif. „Semantic Plug and Play: An Architecture Combining Linked Data and Reconfigurable Hardware“. Conference: 2021 IEEE 15th International Conference on Semantic Computing (ICSC) 2021 DOI: 10.1109/ICSC50631.2021.00043
5. Constantin Wanninger, Sebastian Rossi, Martin Schörner, Alwin Hoffmann, Alexander Poeppel, Christian Eymueller and Wolfgang Reif. „ROSSi A Graphical Programming Interface for ROS 2“. Conference: 2021 21st International Conference on Control, Automation and Systems (ICCAS)
DOI: 10.23919/ICCAS52745.2021.9649736
6. Martin Schörner, Constantin Wanninger, Alwin Hoffmann, Oliver Kosak, Hella Ponsar and Wolfgang Reif. „Modeling and Execution of Coordinated Missions in Reconfigurable Robot Ensembles“. Conference: 2020 Fourth IEEE International Conference on Robotic Computing (IRC) DOI: 10.1109/IRC.2020.00053
7. Martin Schörner, Constantin Wanninger, Alwin Hoffmann, Oliver Kosak and Wolfgang Reif. „Architecture for Emergency Control of Autonomous UAV Ensembles“. Conference: 2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE) DOI: 10.1109/RoSE52553.2021.00014
8. Benjamin Wolf, (...), Constantin Wanninger and 40 other. „The SCALEX Campaign: Scale-Crossing Land Surface and Boundary Layer Processes in the TERENO-preAlpine Observatory“. In journal: 2017 Bulletin of the American Meteorological Society 98(6):1217–1234 DOI: 10.1175/BAMS-D-15-00277.1

9. Oliver Kosak, Constantin Wanninger, Andreas Angerer, Alwin Hoffmann, Andreas Schierl, Hella Ponsar and Wolfgang Reif. „Decentralized Coordination of Heterogeneous Ensembles Using Jadex“. Conference: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W) DOI: 10.1109/FAS-W.2016.65
10. Oliver Kosak, Constantin Wanninger, Andreas Angerer, Alwin Hoffmann, Alexander Schiendorfer, Hella Ponsar and Wolfgang Reif. „Towards Self-Organizing Swarms of Reconfigurable Self-Aware Robots“. Conference: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W) DOI: 10.1109/FAS-W.2016.52
11. Oliver Kosak, Lukas Huhn, Felix Bohn, Constantin Wanninger, Alwin Hoffmann and Wolfgang Reif. „Maple-Swarm: Programming Collective Behavior for Ensembles by Extending HTN-Planning“. In book: 2020 Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles DOI: 10.1007/978-3-030-61470-6_31
12. Oliver Kosak, Felix Bohn, Lennard Eing, Dennis Rall, Constantin Wanninger, Alwin Hoffmann and Wolfgang Reif. „Swarm and Collective Capabilities for Multipotent Robot Ensembles“. Conference: 2020 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation DOI: 10.1007/978-3-030-61470-6_30
13. Christian Eymueller, Julian Hanke, Alwin Hoffmann, Alexander Poeppel, Constantin Wanninger and Wolfgang Reif. „Towards a Real-Time Capable Plug & Produce Environment for Adaptable Factories“. Conference: 2021 IEEE 26th International Conference on Emerging Technologies and Factory Automation (ETFA) DOI: 10.1109/ETFA45728.2021.9613729
14. Martin Schörner, Michelle Bettendorf, Constantin Wanninger, Alwin Hoffmann and Wolfgang Reif. „UAV Inspection of Large Components: Indoor Navigation Relative to Structures“. Conference: 2021 18th International Conference on Informatics in Control, Automation and Robotics DOI: 10.5220/0010556300002994
15. Constantin Wanninger, Raphael Katschinsky, Martin Schörner, Alwin Hoffmann and Wolfgang Reif. „Towards Fully Automated Inspection of Large Components with UAVs: Offline Path Planning“. Conference: 2020 17th International Conference on Informatics in Control, Automation and Robotics DOI: 10.5220/0009887900710080
16. Martin Schörner, Raphael Katschinsky, Constantin Wanninger, Alwin Hoffmann and Wolfgang Reif. „Towards Fully Automated Inspection of Large Components with UAVs: Offline Path Planning and View Angle Dependent Optimization Strategies“. In book: 2020 Informatics in Control, Automation and Robotics DOI: 10.1007/978-3-030-92442-3_7