

## Bachelor-Thesis

Lukas Lux

Matrikelnummer: 1080 18 240012

Prüfungsordnung: BPO Sales Engineering and Product Management 2013

### **Thema: Entwicklung eines Frameworks zur simulationsbasierten Validierung LLM-generierten RoboterCodes in Unity**

Die Programmierung von Robotern ist derzeit ein komplexer und zeitintensiver Prozess, der in der Regel spezielles Fachwissen erfordert. Um diese Hürde zu verringern, erforscht die Wissenschaft neue Ansätze zur Beschleunigung der Arbeitsprozesse. Generative KI-Systeme, insbesondere Large-Language-Modelle (LLMs), bieten dabei ein vielversprechendes Potenzial. Eine zentrale Herausforderung besteht jedoch im sicheren Einsatz von KI-generiertem Quellcode. Am LPS wird deshalb untersucht, wie Simulationstools genutzt werden können, um generierte Lösungen zunächst virtuell zu erproben. Die dabei gewonnenen Erkenntnisse sollen wiederum in die KI-Systeme zurückgeführt werden. Damit dies gelingt, ist eine Überführung der Simulationsergebnisse in eine für LLMs verständliche, textuelle bzw. natürlichsprachliche Form erforderlich.

Ziel dieser Arbeit ist es, eine solche Überführung anhand eines Beispielprozesses und ausgewählter Simulationsparameter zu entwickeln, zu implementieren und methodisch zu evaluieren.

Im Einzelnen sollen folgende Punkte bearbeitet werden:

- Analyse des aktuellen Stands der Technik in den Bereichen Roboterprogrammierung sowie Bewertung von Robotersimulationen.
- Konzeption einer Basis-Architektur zur Übertragung von Erkenntnissen aus Simulationen in eine geeignet weiterzuverarbeitende Form (z. B. JSON).
- Programmiertechnische Umsetzung der Architektur in Unity für ein definiertes Beispielszenario.
- Methodische Erprobung der entwickelten Lösung und Bewertung der erzielten Ergebnisse.

Die Arbeit leistet einen wichtigen Beitrag zum Gelingen des Projektes XYZ am Lehrstuhl für Produktionssysteme.

Ausgabedatum: 16.06.2025

Betreuer: M. Sc. Daniel Syniawa

## **Zusammenfassung**

## **Abstract**

## A Inhaltsverzeichnis

<b>Zusammenfassung</b>	<b>II</b>
<b>Abstract</b>	<b>III</b>
<b>A Inhaltsverzeichnis</b>	<b>IV</b>
<b>B Abbildungsverzeichnis</b>	<b>VII</b>
<b>C Tabellenverzeichnis</b>	<b>IX</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation und Relevanz . . . . .	1
1.2 Zielsetzung und Aufbau der Arbeit . . . . .	1
<b>2 Stand der Technik</b>	<b>3</b>
2.1 Roboterprogrammierung . . . . .	3
2.1.1 Verfahren der Roboterprogrammierung . . . . .	3
2.1.2 Landschaft gängiger Programmierumgebungen . . . . .	4
2.1.3 Physik-Engines zur Simulation von Robotern . . . . .	5
2.2 Large Language Models . . . . .	6
2.2.1 Funktionsweise und Architektur . . . . .	6
2.2.2 Programmcodegenerierung durch Large Language Models . . . . .	7
2.3 LLMs in der Robotik . . . . .	8
2.3.1 Aktuelle Forschungsansätze . . . . .	8
2.3.2 Herausforderungen in der Integration . . . . .	9
2.3.3 Bestehende Frameworks und Tools . . . . .	9
2.4 Zwischenfazit und Forschungsfrage . . . . .	10

<b>3</b>	<b>Implementierung des Frameworks zur simulationsbasierten Validierung</b>	<b>11</b>
3.1	Architektur des Frameworks . . . . .	11
3.1.1	Funktionale Anforderungen . . . . .	11
3.1.2	Zielsetzung und architektonische Anforderungen . . . . .	12
3.1.3	Unity3D als Simulationsplattform . . . . .	13
3.1.4	Systemarchitektur . . . . .	13
3.1.5	Robot Communication Layer . . . . .	16
3.1.6	Event-Driven Architecture . . . . .	18
3.1.7	SafetyEvent und RobotStateSnapshot . . . . .	18
3.2	Implementierung der Monitore zur Fehlererkennung . . . . .	18
3.2.1	Process Flow Monitor . . . . .	19
3.2.2	Collision Detection Monitor . . . . .	21
3.2.3	Singularity Detection Monitor . . . . .	23
3.2.4	Joint Dynamics Monitor . . . . .	28
3.3	Testumgebung und -setup . . . . .	30
3.3.1	Aufbau der Roboterzelle . . . . .	30
3.3.2	Flange als Visualisierungstool . . . . .	33
3.3.3	Modellierung in Unity . . . . .	34
3.4	Datenaufzeichnung und Logging . . . . .	35
<b>4</b>	<b>Validierungsergebnisse der Testszenarien</b>	<b>37</b>
4.1	Überblick und Zielsetzung . . . . .	37
4.2	Prozessflussüberwachung . . . . .	38
4.2.1	Abwandlung der Prozessfolge . . . . .	38
4.2.2	Simulationsergebnis . . . . .	39
4.3	Auswertung Collision Detection Monitor . . . . .	41
4.3.1	Abwandlung des Pfads . . . . .	41
4.3.2	Simulationsergebnis . . . . .	41
4.4	Auswertung Singularity Detection Monitor . . . . .	44
4.4.1	Künstliche Singularitätserzeugung . . . . .	44

4.4.2	Simulationsergebnis . . . . .	45
4.5	Auswertung Joint Dynamics Monitor . . . . .	46
4.5.1	Überhöhung der Gelenkgeschwindigkeiten . . . . .	46
4.5.2	Simulationsergebnis . . . . .	46
4.6	Validierung durch Experteninterview . . . . .	49
4.6.1	Vorgehen und Gegenstand . . . . .	49
4.6.2	Beobachtungen . . . . .	49
4.6.3	Hinweis zur Evaluationsmethodik . . . . .	49
4.6.4	Einordnung des Robotersimulations-Ökosystems . . . . .	49
4.6.5	Zusammenfassung . . . . .	50
4.7	Zusammenfassung der Ergebnisse . . . . .	50
<b>5</b>	<b>Diskussion</b>	<b>52</b>
5.1	Einleitung . . . . .	52
5.2	Diskussion des Frameworks . . . . .	52
5.3	Diskussion der Safetymodule . . . . .	53
5.3.1	Prozessfolgenüberwachung . . . . .	53
5.3.2	Kollisionserkennung . . . . .	53
5.3.3	Achsgeschwindigkeiten und -beschleunigungen . . . . .	53
5.3.4	Singularitätserkennung . . . . .	54
5.4	LLM-gestützte Rückkopplung auf Basis der Safety-Events . . . . .	54
5.5	Reflexion des eigenen Vorgehens . . . . .	55
5.6	Grenzen und Generalisierbarkeit . . . . .	55
<b>6</b>	<b>Fazit und Ausblick</b>	<b>56</b>
6.1	Zusammenfassung . . . . .	56
6.2	Ausblick . . . . .	56
	<b>Literatur</b>	<b>57</b>
	<b>Anhang</b>	<b>61</b>

## B Abbildungsverzeichnis

Abbildung 1:	Schichtenarchitektur des Frameworks. Orange umrandete Module implementieren formalisierte Interfaces (IRobotSafetyMonitor, IRobotConnector). Das RobotState-Objekt dient als gemeinsame Datenstruktur zwischen den Schichten. . . . .	14
Abbildung 2:	Implementierung der IRobotConnector-Interface als Verbindung zwischen RobotState und Roboter-Controller . . . . .	16
Abbildung 3:	Algorithmus zum Ermitteln kinematisch benachbarter Mesh-Collider	22
Abbildung 4:	Screenshot einer Handgelenksingularität in Unity mit farbig dargestellten Koordinatensystemen der DH-Transformationen . . . . .	25
Abbildung 5:	Vorwärtskinematik zur Positionsrechnung . . . . .	27
Abbildung 6:	Mehrstufige Smoothing-Pipeline zur Geschwindigkeitsrechnung	29
Abbildung 7:	Arbeitsraum-Layout des Roboters in der Draufsicht mit Roboter in Home-Position . . . . .	30
Abbildung 8:	Technische Zeichnung des verwendeten Zylinderskopfes, dargestellt mithilfe von Autodesk Fusion 360 . . . . .	32
Abbildung 9:	Technische Zeichnung des ausgelegten Parallelgreifers, dargestellt mithilfe von Autodesk Fusion 360 . . . . .	33
Abbildung 10:	SafetyEvent-Klasse mit Zuordnung zum Monitor und Kritikalität .	35
Abbildung 11:	RobotStateSnapshot-Klasse mit formalisieren Attributen . . . . .	35
Abbildung 12:	Visuelle Darstellung des Prozessflusses durch Konfiguration der Parts und Stations . . . . .	39
Abbildung 13:	JSON-Log zum Prozessfolgenfehler, Achswinkel wurden im Nachhinein entfernt . . . . .	40
Abbildung 14:	Kollision in Unity (links) und zugehörige Position auf Pfad in RobotStudio (rechts). Gelenk 4, 5 und 6 sowie Greifer befinden sich innerhalb der Säulengeometrie. . . . .	41
Abbildung 15:	JSON-Log zur Kollisionserkennung. Sich wiederholende Key-Value Paare wurden verkürzt . . . . .	42
Abbildung 16:	Pose in Unity, bei der eine Wrist-Singularität mit ( $\theta_5 \approx 0^\circ$ ) entsteht	44
Abbildung 17:	Gekürzter Auszug der in Unity aufgezeichneten Safety Events zur Wrist-Singularität . . . . .	45
Abbildung 18:	Gekürzter Auszug der in Unity aufgezeichneten Safety Events des Joint Dynamics Monitor, Ereignis wiederholt sich . . . . .	47

Abbildung 19: Achswinkel (oben) und Achsgeschwindigkeiten (unten) mit markierten Bereichen (rot: Schwellenübertritte in den Rohdaten, blau: Safety Events aus Unity). . . . .	48
---	----



C Tabellenverzeichnis

Tabelle 1: Vergleich verschiedener Robotik-Simulationsplattformen, nach <sup>1</sup> . . . . . 6

Tabelle 2: Überblick über die implementierten Monitore im Framework . . . . . 19

Tabelle 3: Singularitätsschwellwerte und Detektionsbedingungen . . . . . 26

Tabelle 4: Zeitintervalle und Zustände der Joint-Dynamics-Auswertung . . . . . 48

Tabelle 5: Übersicht der getesteten Monitore, Szenarien und erkannten Ereignisse  
im Ergebnisteil . . . . . 50

<sup>1</sup>Vgl. Bilancia et al. 2023

# 1 Einleitung

## 1.1 Motivation und Relevanz

Die rasante technologische Entwicklung und die zunehmende Vielfalt kundenindividuell konfigurierbarer Produkte führen dazu, dass industrielle Produktionsanlagen immer häufiger neu eingerichtet oder umgerüstet werden müssen. In der Robotik erfordert dies die Erstellung und Anpassung von Programmen, die Bewegungsabläufe, Bearbeitungsschritte und Sicherheitsfunktionen eines Roboters definieren. Traditionelle Programmiermethoden sind komplex, controllerspezifisch und setzen ein profundes Verständnis der jeweiligen Kinematik und proprietären Sprachen voraus. Für kleine und mittlere Unternehmen sowie für Fertigungsbereiche mit hoher Variantenvielfalt stellt dies eine Hemmschwelle dar, zumal sich in vielen Branchen ein Mangel an qualifizierten Fachkräften bemerkbar macht.

Gleichzeitig halten generative Sprachmodelle Einzug in die Softwareentwicklung. Sie können aus natürlichsprachlichen Beschreibungen ausführbaren Code erzeugen und versprechen so, die Hürden der Roboterprogrammierung zu senken. In der Praxis ist der direkte Einsatz solcher Large Language Models (LLMs) im industriellen Kontext jedoch riskant: Fehlerhafte Bewegungssequenzen oder fehlerhafte Prozesslogiken können zu Kollisionen, Schäden und Produktionsausfällen führen. Im Zusammenspiel mit den vorherrschenden Sicherheitsanforderungen ist daher eine sorgfältige Validierung des generierten Codes unerlässlich. Simulationstools bieten die Möglichkeit, Roboterprogramme risikofrei zu testen und Abweichungen vom Soll-Verhalten systematisch zu untersuchen. Die vorliegende Arbeit verknüpft diese beiden Entwicklungen: Sie verknüpft LLM-basierte Codegenerierung mit einer simulationsbasierten Prüfung, die sicherheitskritische Aspekte des Programmablaufs überwacht und strukturierte Ereignisprotokolle liefert. Damit wird eine Lücke zwischen der schnellen, aber potenziell unsicheren Codeerzeugung und der industriellen Praxis geschlossen.

## 1.2 Zielsetzung und Aufbau der Arbeit

Ziel dieser Arbeit ist die Konzeption, Umsetzung und Evaluation eines Frameworks zur simulationsbasierten Validierung von mit LLMs generiertem Robotercode. Kernidee ist, dass während der Simulation auftretende Ereignisse – insbesondere Abweichungen in der Prozessabfolge, Kollisionen mit der Umgebung oder dem Werkstück, kinematische Singularitäten sowie Grenzwertverletzungen von Achsgeschwindigkeiten – automatisch detektiert und in einer standardisierten Form gespeichert werden. Diese sogenannten Safety Events umfassen neben einer Beschreibung des erkannten Problems stets einen vollständigen Zustandsabzug der Steuerung (RobotStateSnapshot), der Positionsdaten, Programmzeiger und Metadaten umfasst. Durch die Formalisierung in JSON können die Ergebnisse sowohl für eine manuelle Analyse als auch für

eine mögliche Rückkopplung an generative Sprachmodelle genutzt werden. Auf diese Weise soll ein iterativer Verbesserungsprozess entstehen, der die Qualität der erzeugten Programme steigert und die Eintrittsbarrieren für den Einsatz von Robotik reduziert.

Um dieses Ziel zu erreichen, wird zunächst der Stand der Technik zu Robotersimulation, Offline-Programmierung und LLMs zusammengefasst. Darauf aufbauend wird die Architektur des Frameworks beschrieben, die eine herstellerunabhängige Einbindung unterschiedlicher Robotersteuerungen (über ein Adaptermuster) sowie die modulare Erweiterung um weitere Safety-Monitore ermöglicht. Die Implementierung der vier Überwachungsmodule – Process Flow Monitor, Collision Detection Monitor, Joint Dynamics Monitor und Singularity Detection Monitor – wird anschließend detailliert erläutert. In einem realitätsnahen Beispielszenario mit einer Roboterzelle werden die Module getestet; spezifische Szenarien provozieren Abweichungen im Ablauf, Kollisionen und Grenzwertverletzungen, um die Detektion zu verifizieren. Ein Experteninterview ergänzt die Evaluation und reflektiert die Eignung des Ansatzes aus praktischer Sicht. Abschließend werden die gewonnenen Erkenntnisse diskutiert, Grenzen des Prototyps aufgezeigt und zukünftige Erweiterungsschritte skizziert.

Der Aufbau der Arbeit gliedert sich wie folgt: Kapitel 2 stellt den Stand der Technik dar und ordnet die Problemstellung in den Kontext bestehender Forschungsarbeiten ein. Kapitel 3 beschreibt die Konzeption und Implementierung des Frameworks sowie der einzelnen Safetymodule. Kapitel 4 präsentiert die Ergebnisse der Simulationen und des Experteninterviews. In Kapitel 5 werden die Resultate kritisch diskutiert und die Leistungsfähigkeit, Grenzen und Generalisierbarkeit des Ansatzes bewertet. Kapitel 6 schließt die Arbeit mit Fazit und Ausblick ab und zeigt potenzielle Wege zur Weiterentwicklung auf.

## 2 Stand der Technik

### 2.1 Roboterprogrammierung

Die Roboterprogrammierung kann als die Programmierung von industriellen Manipulatoren verstanden werden, die sich durch ihre programmierbaren und anpassungsfähigen Eigenschaften von anderen Maschinen abheben. Roboterprogramme enthalten präzise Anweisungen und Spezifikationen für die Bewegung des Roboters. Der entscheidende Vorteil der Roboterprogrammierung ist somit Flexibilität: Roboter können durch einfache Software-Umprogrammierung für völlig unterschiedliche Aufgaben eingesetzt werden, während herkömmliche Steuerungstechnik meist auf vordefinierte starre Abläufe beschränkt ist.<sup>2</sup>

#### 2.1.1 Verfahren der Roboterprogrammierung

Grundsätzlich wird bei der Roboterprogrammierung zwischen manuellen und automatischen Verfahren unterschieden.<sup>3</sup> Manuelle Systeme erfordern die explizite Erstellung des Programms durch den Anwender, wobei textbasierte Programmiersprachen (z. B. herstellerspezifische Sprachen wie KUKA Robot Language (KRL) oder ABB RAPID) sowie grafische Oberflächen genutzt werden. Textbasierte und im eingeschränkten Umfang auch grafische Verfahren bieten die Möglichkeit, robotspezifische Datentypen zu deklarieren, einfache Bewegungen zu spezifizieren und mit Werkzeugen und Sensoren zu interagieren. Zur Ausführung von erstelltem Roboterprogrammcode wird dieser an die Robotersteuerung übertragen und unter Einhaltung von Echtzeitbeschränkungen ausgeführt.<sup>4</sup>

Automatische Systeme der Programmierung werden hingegen in lernende Systeme, instruktive Systeme sowie demonstrative Systeme unterteilt. Ein typisches Verfahren demonstrativer Programmierung ist das Führen des Roboters über ein Teach Pendant oder durch die direkte Führung der Roboterhand durch den Arbeitsbereich, wobei die dabei entstehenden Gelenkwinkel oder kartesische Tool Center Point Positionen (TCP-Positionen) aufgezeichnet und im Programm hinterlegt werden. Lernende und instruktive Systeme nutzen Methoden des maschinellen Lernens (z. B. Reinforcement Learning) sowie computerbasiertem Sehen (engl. Computer-Vision) zum Verständnis der Intention und der Umwelt, oft in Rahmen der Mensch-Maschine Kollaboration (MRK).<sup>5</sup>

Ergänzend zur methodischen Einordnung wird bei der Roboterprogrammierung zwischen zwei Durchführungsmodi unterschieden: Online-Programmierung am realen System (z. B. Teach per Teach Pendant oder Handführung), bei der Wegpunkte und Ablauflogik unmittelbar auf der

---

<sup>2</sup>Vgl. Nilsson 1996, S. 1 ff.

<sup>3</sup>Vgl. Biggs und MacDonald 2003, S. 1

<sup>4</sup>Vgl. Mühe et al. 2010, S. 1

<sup>5</sup>Vgl. Biggs und MacDonald 2003, S. 4 ff.

Steuerung aufgezeichnet werden, und Offline-Programmierung (OLP), bei der Programme in virtuellen Zellen entworfen, geprüft und erst anschließend auf die reale Steuerung übertragen werden.<sup>6</sup>

Online-Verfahren sind im industriellen Alltag verbreitet, verursachen jedoch Stillstandszeiten und bergen Test-Risiken.<sup>7</sup> Hierbei können Beschädigungen am Roboter, Maschinen, Werkstücken und Umwelt durch Kollisionen oder fehlerhafter Konfiguration des Roboters auftreten. OLP verlagert Entwurf, Kollisionsprüfung und Taktzeitabschätzung in die Simulation und reduziert so Risiken und Anlagenstillstand.<sup>8</sup> Somit wird die Entwicklung und Evaluierung von Roboterprogrammen ohne physischen Roboter ermöglicht: Programme werden in einer virtuellen Umgebung erstellt, getestet und iterativ verfeinert. Hierfür werden originalgetreue dreidimensionale Computer Aided Design Modelle (3D-CAD-Modelle) des Roboters und seiner Umgebung benötigt, welche teilweise vom Hersteller bereitgestellt werden oder nachmodelliert werden müssen, um den Zielkontext möglichst realitätsnah abzubilden. Ziel ist es durch die OLP problematische Stellen des Roboterprogramms, Ablaufprobleme sowie Nebenwirkungen des direkt und indirekt beeinflussten Prozesses früh zu erkennen.<sup>9</sup> Externe Faktoren wie verformbare Objekte, Fluide oder Personen bei der MRK erhöhen im Rahmen solcher Simulationen maßgeblich die Komplexität. In realitätsnahen Szenarien (z. B. automatisierte Steckverbindungen in Schaltschränken oder Gießen von Schmelzen) treten materialbedingte Nichtidealitäten auf, die zu Wechselwirkungen mit dem Roboter führen. Eine hinreichend genaue physikalische Modellierung des Zielsystems ist daher Voraussetzung.

### 2.1.2 Landschaft gängiger Programmierumgebungen

Im Gegensatz zur Werkzeugmaschinenprogrammierung, die auf standardisierten Sprachen wie G-Code basiert, erschwert das Fehlen einer universellen, herstellerunabhängigen Programmiersprache die Integration verschiedener Robotertechnologien in einer Produktionsanlage.<sup>10</sup> Industrielle Roboterhersteller, beispielsweise KUKA, ABB, Fanuc oder Stäubli, bieten und unterstützen lediglich eigene, proprietäre Programmiersprachen und Programmierschnittstellen, wobei diese sich in Komplexität, Syntax und Semantik unterscheiden.<sup>11</sup> Zudem müssen Roboterprogrammierer mit eingeschränkten Basisbefehlen und Bibliotheken arbeiten. Diese decken zwar die meisten Standardanforderungen ab, ermöglichen jedoch keine fortgeschrittenen Berechnungen oder komplexen Steuerungsstrategien. Offline-Programmierungswerkzeuge wie RoboDK oder Siemens Process Simulate übersetzen 3D-Modellierungsbefehle mithilfe spezifischer Postprozessoren in herstellereigenspezifische Robotercode. Allerdings unterstützen diese Werkzeuge nicht die vollständigen Funktionsbibliotheken der kommerziellen Robotersprachen und können erfahrene Programmierer

<sup>6</sup>Vgl. Biggs und MacDonald 2003, S. 1; Holubek et al. 2014, S. 62 ff.

<sup>7</sup>Vgl. Bilancia et al. 2023, S. 4

<sup>8</sup>Vgl. Holubek et al. 2014, S. 62 ff.

<sup>9</sup>Vgl. Holubek et al. 2014, S. 62 ff.

<sup>10</sup>Vgl. Bilancia et al. 2023, S. 4

<sup>11</sup>Vgl. Lambrecht et al. 2011, S. 116; Bilancia et al. 2023, S. 4

bei komplexen Programmerroutinen nicht ersetzen.<sup>12</sup> Der Versuch, eine einheitliche, standardisierte Programmiersprache für alle Industrieroboter zu definieren und zu verbreiten, scheitert dabei an der mangelnden Kooperation der Hersteller industrieller Roboter.<sup>13</sup>

Einen gegensätzlichen Ansatz verfolgt das open-source Projekt Robot Operating System (ROS). ROS ist ein quelloffenes Middleware-Framework, das Bibliotheken und Werkzeuge für Nachrichtenübertragung, Paketverwaltung und Hardwareabstraktion bereitstellt und damit eine hersteller- und plattformunabhängige Integrationsschicht anstrebt.<sup>14</sup> In der industriellen Praxis wird ROS/ROS 2 primär als Integrations-/Orchestrierungsebene eingesetzt (z. B. Wahrnehmung, Planung, Zellenkoordination). Echtzeitkritische Bewegungs- und Sicherheitsfunktionen verbleiben üblicherweise auf herstellerseitig eingesetzten Robotersteuerungen, und der dokumentierte industrielle Einsatz bleibt anwendungs- und treiberabhängig.<sup>15</sup> Da ROS 1 keine harten Echtzeitanforderungen der Robotersteuerung erfüllen kann, soll ROS 2 dieses Defizit erfüllen. Das erfordert in der Praxis jedoch ein echtzeitfähiges Betriebssystem sowie sorgfältiges Systemtuning, sodass ein reproduzierbares, deterministisches Ausführungsverhalten weiterhin von Konfiguration und Implementierung abhängt.<sup>16</sup>

### 2.1.3 Physik-Engines zur Simulation von Robotern

Aufgrund der herstellerabhängig begrenzten Möglichkeiten der realistischen Abbildung eines Szenarios bei der OLP, bietet es sich an, auf Programme zur physikalischen Simulation von Körpern und Prozessen zurückzugreifen. Mit dem Einsatz von Physik-Engines soll eine virtuelle Umgebung geschaffen werden, in dem die Realität nahezu realgetreu digital abgebildet werden kann und so eine Simulation des Roboters und seiner Umgebung geschaffen werden. Sie modellieren dynamische Interaktionen wie Kollisionen, Schwerkraft und Reibung, was für die präzise Nachbildung des Roboterhaltens entscheidend ist. Obwohl die Genauigkeit dieser Engines als nicht perfekt angesehen wird, da sie die reale Welt nicht exakt abbilden<sup>17</sup>, sind sie für Forschung und Entwicklung essenziell, um physikalisch anspruchsvolle Prozesse zuverlässig simulieren zu können. Die Wahl der Physik-Engine beeinflusst die Stabilität und Wiederholbarkeit der Simulation. Häufig genutzte Physik-Engines sind PhysX, Bullet und ODE. In Tabelle 1 werden gängige Simulationsprogramme im Hinblick auf die verwendeten Physik-Engines, ROS-Kompatibilität und deren Möglichkeit der Integration verglichen. In der Praxis besteht ein Zielkonflikt zwischen numerischer Stabilität, Reproduzierbarkeit (Determinismus) und physikalischer Genauigkeit. Die Wahl der Engine beeinflusst daher nicht nur die Plausibilität der Dynamik, sondern auch die Vergleichbarkeit von Simulationsergebnissen.<sup>18</sup>

---

<sup>12</sup>Vgl. Bilancia et al. 2023, S. 4

<sup>13</sup>Vgl. Lambrecht et al. 2011, S. 116

<sup>14</sup>Vgl. Quigley et al. 2009, S. 1

<sup>15</sup>Vgl. Bonci et al. 2023, S. 3-6

<sup>16</sup>Vgl. Maruyama et al. 2016, S. 1; Bonci et al. 2023, S. 3-5

<sup>17</sup>Vgl. Audonnet et al. 2022, S. 1 ff.

<sup>18</sup>Vgl. Audonnet et al. 2022, S. 1 ff.

Name	Physik-Engine	Open Source	ROS-Integration	ML-Support
Gazebo	Bullet, DART, ODE, Simbody	Ja	Ja	Extern
Ignition	DART	Ja	Ja	Extern
Webots	ODE	Ja	Ja	Extern
Isaac Sim	PhysX	Nein	Ja	Integriert
Unity	Havok, PhysX, RaiSim	Nein	Nein	Extern
PyBullet	Bullet	Ja	Nein	Extern
CoppeliaSim (V-rep)	Newton, Bullet, ODE, Vortex Dynamics	Nein	Ja	Extern
Mujoco	Mujoco	Ja	Nein	Extern

Tabelle 1: Vergleich verschiedener Robotik-Simulationsplattformen, nach <sup>19</sup>

## 2.2 Large Language Models

Aufgrund kürzlicher Entwicklungen im Bereich der LLMs bieten diese durch ihre Fähigkeit der Generierung textueller Inhalte eine vielversprechende Möglichkeit, bei der Programmierung von Robotern zu unterstützen.

### 2.2.1 Funktionsweise und Architektur

Bei großen Sprachmodellen (LLMs) handelt es sich um statistische Modelle, welche in der Lage sind textuelle Inhalte zu übersetzen, zusammenzufassen, Informationen abzurufen und Konversation zu betreiben. Historisch sind LLMs aus der Möglichkeit, neuronale Netze im Modus des self-supervised learning (deutsch: selbst überwachtes Lernen) und anhand großer Mengen textueller Trainingsdaten zu trainieren, entstanden. Dabei haben sich LLMs innerhalb der letzten 10 Jahre aufgrund der hohen Verfügbarkeit digitaler textueller Daten sowie Innovationen im Bereich der Hardware-Technologie zur wichtigsten Technologie im Bereich Künstliche Intelligenz (KI) entwickelt. <sup>20</sup>

Den entscheidenden Durchbruch für moderne LLMs lieferte die Transformer-Architektur, die sequenzielle Abhängigkeiten durch parallele Attention-Mechanismen ersetzt.<sup>21</sup> Mithilfe des Self-Attention-Mechanismus wird für jedes Token die Relevanz zu allen anderen Tokens einer Sequenz berechnet, wodurch ein Modell kontextuelle Beziehungen direkt erfasst, ohne Informationen sequenziell durch versteckte Schichten propagieren zu müssen. Diese Parallelisierung ermöglicht nicht nur schnelleres Training auf den erwähnten großen Datensätzen, sondern schafft auch die

<sup>20</sup>Vgl. Naveed et al. 2024, S. 1 f.

<sup>21</sup>Vgl. Vaswani et al. 2023, S. 1 ff.

Grundlage für das Skalierungsverhalten moderner Sprachmodelle. Folglich basieren aktuelle LLMs wie GPT-4 oder Claude auf dieser Architektur.

### 2.2.2 Programmcodegenerierung durch Large Language Models

LLMs haben bemerkenswerte Fortschritte in der automatischen Code-Generierung erzielt und revolutionieren damit die Softwareentwicklung. Code-LLMs sind in der Lage erfolgreich Quellcode aus natürlichsprachlichen Beschreibungen zu generieren. Aktuelle LLMs demonstrieren anhand von empirischen Benchmarks zur Evaluation der Qualität von Programmiercode (z. B. HumanEval, MBPP und BigCodeBench), dass sie in der Lage sind progressiv bessere Leistungen bei verschiedenen Schwierigkeitsgraden und Programmieraufgaben zu erzielen.<sup>22</sup> Voraussetzungen für die erfolgreiche Programmierung mithilfe von LLMs sind die Miteinbeziehung von Programmiersprachen und dessen technische und syntaktische Dokumentation in die Trainingsdaten des jeweiligen Modells.

Domänenspezifische Code-Generierung stellt LLMs vor besondere Herausforderungen, die deren praktische Anwendbarkeit erheblich einschränken. Neue und weniger populäre Programmiersprachen (engl.: low-ressource languages) sowie domänenspezifische Programmiersprachen sind in offen abrufbaren Datensätzen oft unterrepräsentiert, was zu einem Mangel verfügbarer Trainingsdaten und erhöhten Hürden durch spezialisierte Syntax führt.<sup>23</sup> LLMs zeigen zudem schwächere Leistungen beim Umgang mit domänenspezifischen Bibliotheken<sup>24</sup> – zusätzlicher Kontext (z. B. Repository-Code, Schnittstellenbeschreibungen) ist hier erforderlich. Diese Problematik betrifft Millionen von Entwicklern: Beispielsweise 3,5 Millionen Rust-Nutzer, einer performanten und vergleichsweise neuartigen Programmiersprache, können LLM-Funktionen nicht vollständig ausschöpfen.

Agentische KI-Systeme entwickeln sich zu einer neuen Generation autonomer Softwareagenten, die komplexe Aufgaben selbstständig ohne kontinuierliche menschliche Anleitung ausführen können. Diese Systeme adressieren teilweise die zuvor beschriebenen Herausforderungen bei domänenspezifischer Code-Generierung, indem sie über Command-Line-Interfaces und externe Tools Zugang zu spezialisierten Bibliotheken und Application Programming Interfaces (APIs) erhalten. Das Model Context Protocol (MCP) etabliert sich als offener Standard zur Verbindung von AI-Assistenten mit Datensystemen, Business-Tools und Entwicklungsumgebungen.<sup>25</sup> Viele Softwareentwicklungstools bieten bereits MCP-Unterstützung, um KI-Agenten besseren Zugang zu domänenspezifischen Kontextinformationen und Code-Repositories zu ermöglichen. Die Standardisierung solcher Protokolle kann eine Lösung für die Datenmangel-Problematik bei low-ressource languages und domänenspezifischen Programmiersprachen darstellen, wobei die praktische Wirksamkeit noch evaluiert werden muss. Für Robotik-Anwendungen ist dabei

---

<sup>22</sup>Vgl. Jiang et al. 2024, S. 1

<sup>23</sup>Vgl. Joel et al. 2024, S. 1

<sup>24</sup>Vgl. Gu et al. 2025, S. 1

<sup>25</sup>Vgl. Anthropic 2024



entscheidend, dass agentische Systeme nicht nur natürlichsprachliche Spezifikationen interpretieren, sondern deterministische, zeitkritische Ausführungspfade erzeugen und an bestehende Steuerungsstacks andocken können.

## 2.3 LLMs in der Robotik

Die Integration von LLMs in die Robotik verfolgt drei komplementäre Paradigmen: Vision-Language-Action Modelle, Code-Generation und Embodied Reasoning.<sup>26</sup>

Die zentralen Linien lassen sich knapp gliedern:

- **Vision-Language-Action (VLA):** Gemeinsame Repräsentation von Wahrnehmung, Sprache und Aktion.
- **Code-Generierung:** Synthese ausführbarer Kontrollprogramme aus natürlichsprachlichen Spezifikationen.
- **Embodied Reasoning:** Multimodale Modelle mit kontinuierlichen Sensordaten und Weltmodellen.

### 2.3.1 Aktuelle Forschungsansätze

Google DeepMinds RT-2 repräsentiert den Vision-Language-Action Ansatz, bei dem Roboteraktionen als Text-Token behandelt und gemeinsam mit visuellen und sprachlichen Daten trainiert werden. Dieser Ansatz ermöglicht emergente Fähigkeiten wie das Verstehen von Zahlen oder Icons ohne explizites Training.<sup>27</sup> Code as Policies verfolgt hingegen die direkte Generierung von ausführbarem Python-Code aus natürlichsprachlichen Befehlen, wobei durch hierarchische Code-Generation komplexe Kontrollstrukturen wie Schleifen und Bedingungen erzeugt werden.<sup>28</sup> Das Modell PaLM-E demonstriert einen dritten Weg durch multimodale Embodied Language Models, die Sprache, Vision und kontinuierliche Sensordaten in einem gemeinsamen Embedding-Raum verarbeiten<sup>29</sup>. Parallel entwickeln Forscher Brain-Body-Problem Ansätze, die kognitive Architekturen mit physischen Roboterkörpern verbinden sowie Prompt-basierte Methoden, bei denen LLMs direkt Low-Level-Kontrollaktionen vorhersagen.<sup>30</sup> Diese Vielfalt der Ansätze zeigt, dass die Forschung noch keine dominante Architektur etabliert hat und verschiedene Wege zur Integration von Sprache und Robotik exploriert.<sup>31</sup>

---

<sup>26</sup>Vgl. Salimpour et al. 2025, S. 2 ff.

<sup>27</sup>Vgl. Brohan et al. 2023, S. 1 ff.

<sup>28</sup>Vgl. Liang et al. 2023, S. 1 ff.

<sup>29</sup>Vgl. Driess et al. 2023, S. 2 f.

<sup>30</sup>Vgl. Wang et al. 2024, S. 1; Bhat et al. 2024, S. 2

<sup>31</sup>Vgl. Salimpour et al. 2025, S. 3 ff.

### 2.3.2 Herausforderungen in der Integration

Die Verankerung abstrakter Sprachkonzepte in physischen Robotersystemen scheitert an drei fundamentalen Problemen. Erstens müssen Roboter symbolische Repräsentationen mit sensorischen Erfahrungen verknüpfen - das von Harnad definierte Symbol-Grounding-Problem bleibt trotz jahrzehntelanger Forschung ungelöst<sup>32</sup>. Zweitens fehlen standardisierte Schnittstellen zwischen hochabstrakten Sprachbefehlen und niedere, hardwarenahe Motorkommandos, wodurch jede Roboterplattform individuelle Übersetzungsmechanismen benötigt. Drittens limitieren Echtzeitanforderungen die Komplexität der Verarbeitung, da Roboter innerhalb von Millisekunden auf Umweltveränderungen reagieren müssen. Moderne Systeme versuchen diese Herausforderungen durch die Integration dreier Wahrnehmungsebenen zu lösen: Interozeption erfasst interne Zustände, Propriozeption überwacht Gelenkstellungen und Bewegungen, während Exterozeption die Umgebung durch Kameras und Sensoren interpretiert<sup>33</sup>. Jedoch führt diese Komplexität zu erhöhtem Rechenaufwand und erschwert die Fehlerdiagnose bei unerwarteten Verhaltensweisen. Folglich benötigen robotische Systeme neue Architekturen, die effizient zwischen abstrakten Sprachmodellen und konkreten Aktionsräumen vermitteln. In der Literatur werden deshalb hybride Architekturen diskutiert, die symbolische Planung, differenzierbare Wahrnehmung und reaktive, zeitkritische Kontrolle kombinieren. Für industrielle Szenarien bleibt die Frage zentral, wie viel Autonomie ein LLM-gestütztes System erhalten kann, ohne Validierbarkeit und Betriebssicherheit zu kompromittieren.

### 2.3.3 Bestehende Frameworks und Tools

Aktuelle Frameworks standardisieren die Integration von LLMs in robotische Systeme durch modulare Architekturen. ROS-LLM verbindet das Robot Operating System mit verschiedenen Sprachmodellen und transformiert natürlichsprachliche Befehle automatisch in ausführbare Aktionssequenzen<sup>34</sup>. Das Framework implementiert drei Ausführungsmodi: sequenzielle Abarbeitung für einfache Aufgaben, Verhaltensbäume für reaktive Systeme und Zustandsautomaten für komplexe Ablaufsteuerungen. Entwickler konfigurieren atomare Aktionen wie das Greifen eines Objektes oder bestimmte Pfadnavigationen, die das LLM dann zu komplexen Verhaltensketten kombiniert. Simulationsumgebungen beschleunigen parallel die Entwicklung durch massiv-parallele GPU-Berechnungen – mit Isaac Sim von NVIDIA lassen sich beispielsweise Tausende Roboterinstanzen gleichzeitig simulieren.<sup>35</sup> Solche Plattformen ermöglichen einen theoretischen Zero-Shot-Transfer von der Simulation zur Realität durch systematische simulierte Domänenrandomisierung von Physikparametern, Sensorrauschen und Umgebungsvariationen.<sup>36</sup> Die Standardisierung solcher Werkzeugketten soll Entwicklungszeiten in Zukunft reduzieren und die Programmierung von Robotern niederschwelliger machen.

---

<sup>32</sup>Vgl. Cohen et al. 2024, S. 1 ff.

<sup>33</sup>Vgl. Valenzo et al. 2022, S. 3,13

<sup>34</sup>Vgl. Mower et al. 2024, S. 1 f.

<sup>35</sup>Vgl. NVIDIA Corporation 2018

<sup>36</sup>Vgl. Fickinger et al. 2025, S. 1

## 2.4 Zwischenfazit und Forschungsfrage

Die bisherigen Grundlagen zeigen: Simulation und Offline-Programmierung sind zentrale Bausteine, um Robotikprozesse vorab sicher und nachvollziehbar zu prüfen. LLMs eröffnen dabei die Möglichkeit, Roboterprogrammcode aus natürlicher Sprache zu generieren, stoßen in industriellen Kontexten jedoch auf die Anforderung, Ergebnisse systematisch und reproduzierbar zu validieren.

Daraus leitet sich die zentrale Frage, wie gut sich von LLMs erzeugter Robotercode in einer simulationsbasierten Umgebung hinsichtlich prozessrelevanter Aspekte prüfen und bewerten lässt.

Zur Beantwortung wird in den folgenden Kapiteln eine geeignete Umgebung zur Simulation eines Roboters aufgebaut und an einem kompakten, realitätsnahen Szenario erprobt. Ziel ist das Ausführen von Robotercode innerhalb dieser Umgebung und belastbare Prüfung nach definierten möglichen Fehler im Programmablauf mit nachvollziehbarer Dokumentation der Ergebnisse. Dazu beschreibt Kapitel 3 dazu die Architektur der Frameworks, die verschiedenen zu prüfenden Fehlerarten sowie das Testszenario. Weiterführend wird durch Anwendung des Frameworks in Kapitel 4 ausgewertet, inwiefern Fehler im Roboterprogrammcode erkannt werden sowie welche Struktur und Inhalt dessen Dokumentation folgen.

## **3 Implementierung des Frameworks zur simulationsbasierten Validierung**

Kern der Arbeit ist die Entwicklung eines Frameworks zur Detektion von Fehlern bei der Ausführung von Roboterprogrammcode. In diesem Kapitel werden die technischen Grundlagen sowie das Vorgehen zur Implementierung des Frameworks so beschrieben und anschließend ein Testszenario entwickelt, welches zur Validierung des entwickelten Frameworks dienen soll.

### **3.1 Architektur des Frameworks**

Das entwickelte Framework wird innerhalb der Laufzeit- und Entwicklungsumgebung der Software Unity3D des Unternehmens Unity Technologies implementiert. Es basiert konzeptuell auf der simulierten Ausführung von Roboterprogrammcode mittels eines virtuellen Controllers innerhalb der Software RobotStudio des Unternehmens ABB und der Übertragung relevanter Bewegungs- und Zustandsdaten in Unity3D. Dort wird ein baugleicher Roboter mittels eines 3D-CAD-Modells abgebildet, welcher so den in RobotStudio simulierten Roboter in annähernder Echtzeit abbildet. Darauf aufbauend werden die Überwachungsmodule (Monitore) integriert, welche unterschiedliche Aspekte des Roboterhaltens beobachten und Abweichungen vom gewünschten Verhalten erkennen und dokumentieren.

Im Folgenden wird erläutert, wie das Framework funktional, architektonisch und algorithmisch aufgebaut und implementiert ist. Grundlegend werden dabei für das Framework folgende Festlegungen vorgenommen, welche sich in Kapitel 3.3.1 wiederfinden: Als Testobjekt wird ein ABB IRB 6700-235/2.65 Roboter verwendet. Dabei handelt es sich um einen Knickarmroboter mit 6 Freiheitsgraden und 6 Gelenken.<sup>37</sup> Dieser wird in RobotStudio Version 2025.3 simuliert und mittels eines OmniCore-Controllers gesteuert.

#### **3.1.1 Funktionale Anforderungen**

Die funktionalen Anforderungen an das Framework lassen sich in zwei Hauptkategorien unterteilen: die Echtzeitüberwachung kritischer Roboterzustände und Programmablaufereignisse sowie die systematische Protokollierung der auftretenden Ereignisse.

##### **Überwachung kritischer Roboterzustände**

Das Framework soll vier zentrale Überwachungsfunktionen implementieren:

---

<sup>37</sup>Vgl. ABB Robotics 2025

1. **Kollisionserkennung:** Identifikation von Kontakten zwischen Roboterkomponenten und Objekten in der Arbeitsumgebung sowie Selbstkollisionen zwischen benachbarten Gliedern der kinematischen Kette.
2. **Singularitätserkennung:** Frühzeitige Erkennung kinematischer Singularitäten (Handgelenk-, Schulter- und Ellbogensingularitäten), die zu Kontrollverlust oder undefinierten Bewegungen führen können.
3. **Prozessflussüberwachung:** Validierung der korrekten Abfolge von Bearbeitungsstationen für Werkstücke und Erkennung von Abweichungen von der spezifizierten Sequenz.
4. **Gelenkdynamiküberwachung:** Kontinuierliche Überprüfung von Gelenkwinkeln, -geschwindigkeiten und -beschleunigungen auf Grenzwertüberschreitungen, insbesondere während der Handhabung von Werkstücken.

### Systematische Ereignisprotokollierung

Mithilfe des Frameworks soll es möglich sein, die genannten Fehlerarten systematisch debuggen zu können und die Fehlersuche in Roboterprogrammcode zu vereinfachen. Neben der reinen Erkennung ist zur späteren Analyse und Fehlersuche die formalisierte Dokumentation der Fehlerereignisse notwendig. Bei der Dokumentation soll jedes detektierte Ereignis um entsprechenden Metadaten zum aktuellen Status sowie hilfreichen Daten zur weiteren Fehleranalyse des Roboters ergänzt werden. Folgende Metadaten lassen sich nach initialem Testen mithilfe der zur Verfügung stehenden Tools extrahieren:

- **Zeitlicher Kontext:** Präziser Zeitstempel des Ereignisses mit Millisekunden-Auflösung
- **Räumlicher Kontext:** Vollständige Gelenkwinkelkonfiguration und TCP-Position zum Ereigniszeitpunkt
- **Programmkontext:** Aktuelles Modul, Routine und Programmzeile der Robotersteuerung
- **Ereignisspezifische Daten:** Je nach Ereignistyp relevante Zusatzinformationen (Kollisionspunkt, Singularitätstyp, Sequenzabweichung, Grenzwertüberschreitung)

Diese strukturierte Erfassung ermöglicht die nachgelagerte genaue Auswertung der vorhandenen Daten und soll das Fehlersuchen (Debuggen) im Roboterprogrammcode erleichtern.

### 3.1.2 Zielsetzung und architektonische Anforderungen

Der Stand der Technik aktueller Simulationsplattformen für Roboter zeigt, dass virtuelle Steuerungen und damit zusammenhängende Simulationsplattformen herstellerspezifisch entwickelt und mit Ausnahme von ROS keine einheitliche Schnittstelle zur Kommunikation mit externen Plattformen oder Physik-Engines bieten. Somit muss für jeden Robotertyp ein designierter Konnektor genutzt werden, um diesen mit einer externen, herstellerunabhängigen Simulationsplattform zu

verbinden. Um eine herstellerunabhängige und erweiterbare Plattform zu entwickeln, sollte also eine gemeinsame Schnittstelle implementiert werden.

Zusammenfassend verfolgt die Architektur des Frameworks drei zentrale Ziele:

1. **Vendor-Agnostik:** Abstraktion verschiedener Roboterhersteller durch einheitliche Interface-basierte Architektur ohne herstellerspezifische Abhängigkeiten im Kern-Framework
2. **Modulare Erweiterbarkeit:** Plugin-System für Safety Monitoring Module und Kommunikationsprotokolle ohne Änderungen der bestehenden Architektur
3. **Echtzeitfähige Kommunikation:** Latenzarme Datenübertragung für Motion Control und ereignisbasierte Sicherheitsüberwachung

### 3.1.3 Unity3D als Simulationsplattform

Unity3D wird in dieser Arbeit ausschließlich als *Simulationslaufzeit* verwendet. Die Engine stellt Szene, Physik und Rendering bereit und bietet dank ihrer ausgereiften 3D/Physik-Infrastruktur eine plattformübergreifende Grundlage für den digitalen Zwilling des Roboters.<sup>38</sup> Die eigentliche Anwendungslogik wird durch klar definierte Adapter von der Engine entkoppelt, wodurch eine strikte Trennung zwischen Simulation und Domäne gewahrt bleibt. Asynchrone Abläufe im Update-Loop dienen lediglich der Taktung und greifen nicht direkt auf Unity-APIs zu. Die Wahl von Unity begründet sich zudem durch die breite Tooling-Unterstützung und die Möglichkeit, sowohl visuelle als auch programmatische Arbeitsabläufe zu kombinieren.<sup>39</sup>

### 3.1.4 Systemarchitektur

Das entwickelte Framework implementiert eine vierschichtige Architektur, die eine klare Trennung der Verantwortlichkeiten gewährleistet (siehe Abbildung 1). Diese Strukturierung folgt etablierten Software-Engineering-Prinzipien, um Wartbarkeit und Erweiterbarkeit zu gewährleisten.

---

<sup>38</sup>Vgl. Andaluz et al. 2016, S. 247

<sup>39</sup>Vgl. Bartneck et al. 2015, S. 431

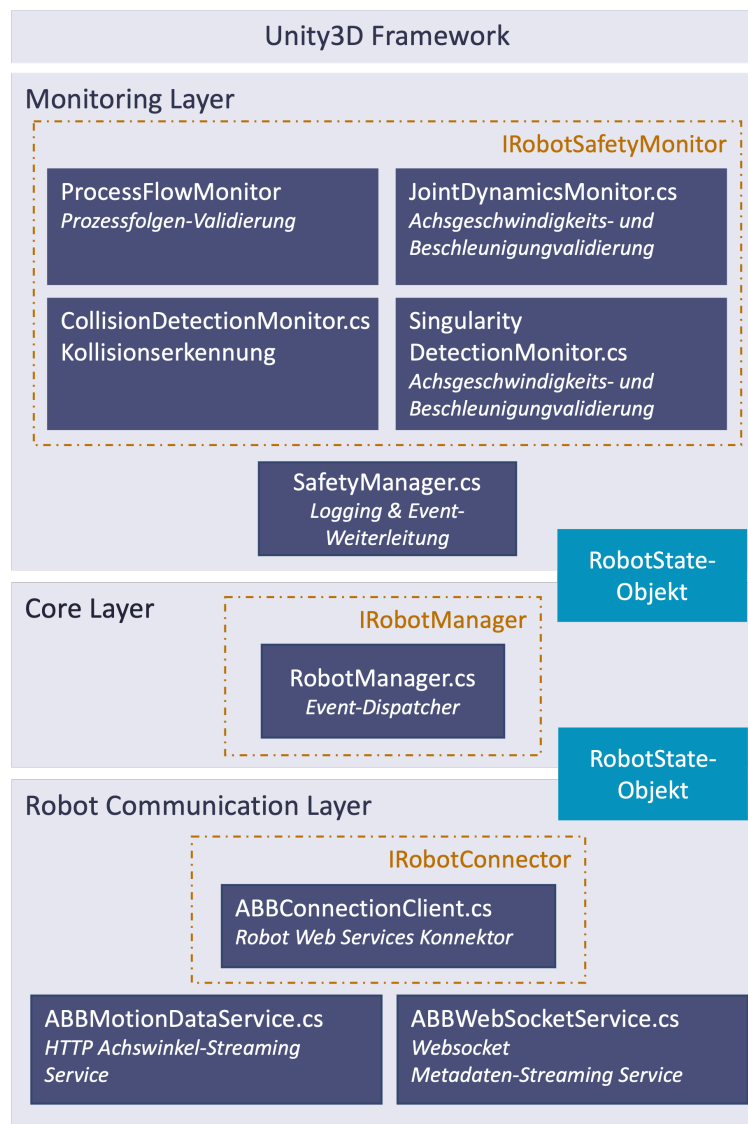


Abbildung 1: Schichtenarchitektur des Frameworks. Orange umrandete Module implementieren formalisierte Interfaces (IRobotSafetyMonitor, IRobotConnector). Das RobotState-Objekt dient als gemeinsame Datenstruktur zwischen den Schichten.

Die Architektur gliedert sich in vier logisch getrennte Schichten:

1. **Unity3D Framework:** Bereitstellung der Simulationsumgebung und Physik-Engine
2. **Monitoring Layer:** Implementierung der Sicherheitsmonitore
3. **Core Layer:** Zentrales State-Management und Event-Koordination
4. **Robot Communication Layer:** Adapter für herstellerspezifische Robotersteuerungen

### Unity3D Framework (Simulationsschicht)

- **Zweck:** Bereitstellung der Simulationslaufzeit.

- **Funktion:** Taktung des Systems (Update-Loop), Szenenhosting und Bootstrap der Adapter. Abseits des Szenenlayouts und der Darstellung der Objekte soll hier keine tiefergehende Prozesslogik implementiert werden.
- **Schnittstellen:** Registrierung der Adapter über ein Dependency-Injection-Konzept; Monitore greifen nicht direkt auf Engine-APIs, lediglich Collider-Objekte werden weitergegeben.

### Core Layer

- **Zweck:** Domänenkern für den Roboterzustand und die Ereignisverteilung.
- **Funktion:** Aktualisierung des RobotState, Verteilung von Zustandsänderungen an registrierte Monitore (Observer) und Aggregation von SafetyEvents.
- **Schnittstellen:** RobotManager, RobotSafetyManager und RobotState bilden die Kernklassen für Dispatch und Aggregation <sup>40</sup>.

### Monitoring Layer

- **Zweck:** Kapselung der Prüfregele als Policies für einzelne Fehlertypen (Kollision, Prozessfolge, Singularität, Dynamik).
- **Funktion:** Erzeugung normalisierter SafetyEvents und Meldung von Grenzverletzungen.
- **Schnittstellen:** Einheitliches Kerninterface IRobotSafetyMonitor definiert die Bindung für alle Monitore.

Der RobotManager im Core Layer fungiert als zentraler Event-Dispatcher, der Zustandsänderungen an alle registrierten Monitoring-Komponenten propagiert. Diese Implementierung des Observer Patterns ermöglicht eine lose Kopplung zwischen den Komponenten<sup>41</sup>: Die Sicherheitsmonitore müssen lediglich das IRobotSafetyMonitor-Interface implementieren und können zur Laufzeit dynamisch registriert oder entfernt werden, ohne andere Systemteile zu beeinflussen.

Die in Abbildung 1 orange markierten Module definieren standardisierte Interfaces, die als Verträge zwischen den Schichten dienen. Diese Interface-basierte Abstraktion realisiert das Open-Closed-Prinzip, welches festlegt, dass Komponenten einerseits offen für Erweiterung sein sollen, andererseits geschlossen für Veränderung<sup>42</sup>: Neue Robotertypen können durch Implementierung des IRobotConnector-Interfaces integriert werden, ohne den Core Layer zu modifizieren. Analog können zusätzliche Sicherheitsmonitore über das IRobotSafetyMonitor-Interface hinzugefügt werden. Diese Architekturentscheidung gewährleistet, dass das System für Erweiterungen offen bleibt, während die Kernfunktionalität stabil und unverändert bleibt.

---

<sup>40</sup>Vgl. Gamma et al. 1994, S. 293 f.

<sup>41</sup>Vgl. Gamma et al. 1994, S. 293-303

<sup>42</sup>Vgl. Martin 2003



Die Kommunikation zwischen den Schichten erfolgt ereignisgesteuert über das `RobotState`-Objekt als gemeinsame Datenstruktur. Die `Communication Layer` aktualisiert kontinuierlich den Roboterzustand durch `WebSocket`-Events für Zustandsdaten der Robotersteuerung und `HTTP`-Polling für Bewegungsdaten. Der `RobotManager` verteilt diese Zustandsänderungen asynchron an alle registrierten `Observer`. Diese `Event-Driven Architecture` adressiert die Anforderungen an ein erweiterbares und gleichzeitiges `Messaging-Modell`. So lassen sich `Events` an mehrere Teilnehmer gleichzeitig weitergeben und verarbeiten, ohne dabei auf den `Processing-Zyklus` zu warten. Weiterführend werden `Events` getrennt voneinander verarbeitet, das heißt der `Main-Thread` muss nicht auf die Fertigstellung der Abarbeitung von `Events` durch Empfänger warten.<sup>43</sup>

Der `SafetyManager` aggregiert die von den Monitoren generierten Sicherheitsereignisse und implementiert eine kontextabhängige `Logging-Strategie`. Bei laufendem Roboterprogramm werden detaillierte `JSON-Logs` für spätere Analyse erstellt, während im `Leerlauf-Zustand` nur kritische Ereignisse in die Konsole geloggt werden. Diese Differenzierung optimiert sowohl die `Performance` als auch die `Nachvollziehbarkeit` von Sicherheitsvorfällen.

Die gewählte Schichtenarchitektur ermöglicht die unabhängige Entwicklung und Testung einzelner Komponenten. Horizontale Erweiterungen (neue Monitore) und vertikale Erweiterungen (neue Robotertypen) können ohne ungewollte Seiteneffekte auf bestehende Komponenten implementiert werden.

### 3.1.5 Robot Communication Layer

Das `Robot Communication Layer` organisiert die Kommunikation mit der zugrundeliegenden Robotersteuerung und implementiert das Interface des Typs `IRobotConnector`. Als Verbindungsclient zwischen `Unity3D` und externer Schnittstelle des Robotersystems implementiert dieser das `IRobotConnector` Interface, welches als Vorlage für eine Anbindung an eine Robotersteuerung jedes Typs fungiert und standardisierte Methoden implementiert (vgl. Abbildung 2).

```
namespace RobotSystem.Interfaces
{
    public interface IRobotConnector
    {
        event System.Action<RobotSystem.Core.RobotState> OnRobotStateUpdated;
        event System.Action<bool> OnConnectionStateChanged;

        bool IsConnected { get; }
        RobotSystem.Core.RobotState CurrentState { get; }

        void Connect();
        void Disconnect();
    }
}
```

Abbildung 2: Implementierung der `IRobotConnector`-Interface als Verbindung zwischen `RobotState` und Roboter-Controller

---

<sup>43</sup>Vgl. Hohpe 2006, S. 1 ff.

Ein Interface lässt sich anhand dieses Beispiels in 3 Funktionsbereiche aufteilen: Events, Attribute und Methoden.

Das Interface definiert Events (Aktionen) die bei definierten Zustandsänderungen in der Laufzeit ausgeführt werden. Andere Bestandteile des Frameworks sind in der Lage, ein Event zu abonnieren und eine Methode zu registrieren, welche ausgeführt werden soll, wenn dieses Event auftritt. Ist dies der Fall, wird das entsprechende Modul über die Änderung benachrichtigt und bekommt gegebenenfalls neue Daten zur Verfügung gestellt. Diese event-getriebene Kommunikation sorgt dafür, dass einerseits alle Komponenten proaktiv auf den neusten Stand der Daten gebracht und gehalten werden, andererseits aber keine ressourcenblockierenden Prozesse ausgeführt werden müssen, um gegebenenfalls Statusänderungen abzufragen.

Weiterführend definiert das Interface IRobotConnector Attribute, welche den aktuellen Verbindungsstatus (verbunden = `true`, getrennt = `false`) speichern sowie das State-Objekt des Roboters. Definiert durch die schreibgeschützten, automatisch implementierte Eigenschaft mit einem Getter können Attribute auch von außerhalb des RobotConnectors abgefragt werden, jedoch nicht überschrieben.

Zuletzt gibt das Interface die Methoden `Connect` und `Disconnect` vor, welche hier die wichtigsten Methoden zum Verbinden und Trennen von der jeweiligen Robotersteuerung darstellen.

Als initiale Komponente wird eine Verbindung zum Controller eines Roboters benötigt, um den Roboter in Unity emulieren zu können und Daten zu verarbeiten. Dazu wird hier mittels eines HTTP-Clients zur Schnittstelle des RobotStudioSDKs über die API RobotWebServices (RWS) eine Verbindung aufgebaut. Die Verbindung zu RobotWebServices entsteht durch einen HTTP-Client und der Authentifizierung mit in RobotStudio festgelegten Zugangsdaten. Anschließend kann über einen erhaltenen Cookie die Verbindung aufrechterhalten werden und aktuelle Daten über den Roboter sowohl abgefragt, also auch der Roboter selbst gesteuert werden.<sup>44</sup>

Die RWS-Schnittstelle bietet einerseits die Möglichkeiten, aktuelle Achswinkel und TCP-Werte abzurufen, als auch weitere Metadaten, wie das aktuell laufende Programm, den aktuellen Motorstatus oder auch die Codezeile, welche aktuell vom Programmzeiger ausgeführt wird. Weiterführend bietet RWS die Möglichkeit, aktuellen analogen und digitalen Signale abzurufen, was nötig ist, um den Greifer steuern zu können.

Bei den oben genannten Daten mit Ausnahme der Achswinkel handelt es sich um Parameter, welche sich im Laufe eines Programmablaufs vergleichsweise selten verändern. Daher wird hier auf den Websockets-Endpunkt von RWS zugegriffen, um innerhalb einer Duplex-Kommunikation sich verändernde Signale oder auch Programmstatus zu empfangen. Dazu wird mithilfe des HTTP-Clients eine Anfrage zur Subscription auf verschiedene Parameter (bspw. den ProgrammPointer) gestellt, und anschließend eine Websockets-Session aufgebaut. Die Achswinkel werden zeitgleich über eine asynchron endlosen Task in einer in Unity definierbaren Frequenz abgefragt.

---

<sup>44</sup>Vgl. ABB

Der Client implementiert dabei das Interface `IRobotConnector` und gibt ein `RobotState`-Objekt mit den empfangenen Daten an den `RobotManager` weiter, welcher hier als zentraler Koordinator des aktuellen `RobotState` fungiert.

### 3.1.6 Event-Driven Architecture

Das System nutzt ein durchgängiges Event-System für lose Kopplung:

- `OnRobotStateUpdated`: Zustandsänderungen
- `OnConnectionStateChanged`: Verbindungsstatus
- `OnSafetyEventDetected`: Sicherheitsereignisse
- `OnMotorStateChanged`: Motorstatusänderungen

### 3.1.7 SafetyEvent und RobotStateSnapshot

Als für die Auswertung der Simulationsergebnisse relevantes Teil des Frameworks wird die `SafetyEvent`-Klasse implementiert. Jedes Mal, wenn ein Ereignis, welches mit der tatsächlichen Simulation eines Roboterprogramms zusammenhängt, auftritt, wird ein `SafetyEvent` instanziiert. Dieses wird initial von der jeweiligen überwachenden Komponente (einem `SafetyMonitor`) instanziiert und mit dem aktuellsten `RobotState` als unveränderliches Objekt (`RobotStateSnapshot`) befüllt. Weiterführend erhält es vom jeweiligen `SafetyMonitor` variierende Kontextinformationen, die später für die Auswertung des Ereignisses verwendet werden. Zusammenfassend besteht ein `SafetyEvent` aus folgenden Komponenten:

- **SafetyEvent**: Unveränderliches Value Object für Sicherheitsereignisse
- **RobotStateSnapshot**: Immutable Zustandserfassung zum Ereigniszeitpunkt
- **Ereignistypen**: Info, Warning, Critical mit konfigurierbaren Schwellwerten
- **Kontextdaten**: Vollständige Roboterzustandserfassung für Forensik

**Einordnung** Die folgenden Kapitel beschreiben die Implementierung der vier Monitore auf Basis der zuvor skizzierten Schichten. In Kapitel 4 werden die Ergebnisse anhand gezielter Testszenarien dargestellt und in gekürzten Log-Auszügen (minted) veranschaulicht.

## 3.2 Implementierung der Monitore zur Fehlererkennung

### Überblick

Zur Orientierung sind die im Framework implementierten Monitore in Tabelle 2 zusammengefasst. Die Tabelle zeigt Ziel, Eingabedaten und Art der generierten Events. Dadurch wird die Einordnung der folgenden Abschnitte erleichtert.

<b>Monitor</b>	<b>Ziel</b>	<b>Eingabedaten / Events</b>
Prozessfolgen	Einhaltung der definierten Abfolge von Arbeitsschritten	Aktuelle Sequenz, Soll-Sequenz / Event bei Abweichung
Kollisionen	Erkennung von Zusammenstößen zwischen Roboter und Umgebung	Geometrieinformationen / Event mit Schweregrad und beteiligten Objekten
Singularitäten	Detektion von Wrist-, Elbow- und Shoulder-Singularitäten	Gelenkwinkel, Jacobian / ENTERING- und EXITING-Events mit Manipulierbarkeit
Gelenkgeschwindigkeiten	Überwachung von Winkel-, Geschwindigkeits- und Beschleunigungsgrenzen	Gelenkwinkel / exceeded- und resolved-Events bei Grenzwertverletzung

Tabelle 2: Überblick über die implementierten Monitore im Framework

### 3.2.1 Process Flow Monitor

#### Theoretische Grundlagen der Prozessflussüberwachung

Die Überwachung von Produktionsabläufen in der Robotik basiert auf der Validierung deterministischer Werkstückpfade durch diskrete Fertigungsstationen. In der Literatur werden solche Systeme als Discrete Event Systems (DES) modelliert, bei denen Zustandsübergänge durch definierte Ereignisse ausgelöst werden.<sup>45</sup>

#### Praktisches Vorgehen

Das System modelliert den Fertigungsprozess durch zwei zentrale Domänenobjekte. Die `Part`-Klasse repräsentiert Werkstücke mit eindeutiger ID, Name und Typ. Jedes Werkstück speichert seine erforderliche Bearbeitungssequenz als Array von Station-Referenzen und verfolgt seinen aktuellen Fortschritt durch einen Sequenzindex. Die `Station`-Klasse definiert Bearbeitungsbereiche durch Trigger-Collider an den physischen Positionen der Arbeitsstationen. Jede Station besitzt einen eindeutigen Namen und numerischen Index zur Sortierung.

Die Prozesspfade werden deklarativ im Unity-Inspector durch Zuweisung der Station-Referenzen definiert. Diese Konfiguration ermöglicht die flexible Modellierung verschiedener Fertigungsabläufe ohne Programmänderungen. Zur Laufzeit validiert das System automatisch die Einhaltung der definierten Sequenzen und protokolliert jeden Stationsbesuch mit Zeitstempel für spätere Analysen.

<sup>45</sup>Vgl. Cassandras und Lafortune 2021, S. 39 f.

Die Schwierigkeit liegt hier in der korrekten Interpretation von Trigger-Events unter Berücksichtigung der dynamischen Objekthierarchie. Wenn ein Werkstück gegriffen wird, wechselt es in Unity vom Weltkoordinatensystem in das lokale Koordinatensystem des Greifers. Diese Hierarchieänderung erzeugt scheinbare Ein- und Austritts-Events an Stationen, die nicht als Prozessübergänge interpretiert werden dürfen.

Der Monitor implementiert daher eine Grip-State-Awareness, die vier relevante Zustandsübergänge unterscheidet: Ablegen (Validierung der Zielstation), Aufnehmen (Protokollierung der Quellstation), Transport (wird ignoriert) und ungegriffene Bewegungen (Fehlererkennung). Diese Unterscheidung erfolgt durch hierarchische Analyse der Transform-Komponente - ein Werkstück gilt als gegriffen, wenn es Kind-Objekt des Roboters ist.

Die Implementierung nutzt ein Layer-System zur Ereignisfilterung. Werkstücke operieren auf Layer 30 (Parts), Stationen auf Layer 31 (ProcessFlow). Diese Trennung ermöglicht selektive Trigger-Erkennung ohne Interferenz mit der physikalischen Kollisionserkennung des Roboters.

### **Konkrete Implementierung**

Der ProcessFlowMonitor koordiniert die Überwachung durch Subscription auf Station-Events. Bei jedem Trigger-Event korreliert er den Grip-State mit der Werkstückbewegung und validiert Transitionen gegen die definierte Sequenz. Der Monitor klassifiziert drei Fehlertypen: WrongSequence für nicht-sequenzielle Bewegungen, SkippedStation für übersprungene Stationen und UnknownStation für undefinierte Ziele.

Bei Erkennung eines Prozessfolgenfehlers generiert der Monitor ein SafetyEvent mit vollständigen Metadaten. Die Schweregrade sind konfigurierbar - standardmäßig werden falsche Sequenzen als kritisch und übersprungene Stationen als Warnungen klassifiziert. Ein Cooldown-Mechanismus verhindert mehrfache Meldungen identischer Verletzungen. Das Event wird an den RobotSafetyManager weitergeleitet, der es mit dem aktuellen Roboterzustand anreichert und gemäß der konfigurierten Logging-Strategie verarbeitet.

Die Überwachung der Prozessfolgen ist insbesondere in realen Produktionsszenarien relevant, da eine fehlerhafte Reihenfolge zu Sicherheitsrisiken oder Ausschuss führen kann. In Kapitel 4.2 wird ein Beispiel gezeigt, in dem eine bewusst abweichende Abfolge simuliert wurde.

### 3.2.2 Collision Detection Monitor

#### Theoretische Grundlagen der Kollisionserkennung

Die Kollisionserkennung in Unity basiert auf einem zweistufigen Verfahren, das von der integrierten PhysX 5.1 Engine implementiert wird <sup>46</sup>. In der **Broad Phase** werden zunächst Axis-Aligned Bounding Boxes (AABB) aller Objekte auf mögliche Überlappungen geprüft. Dieser Schritt reduziert die Anzahl der detaillierten Prüfungen von  $O(n^2)$  auf  $O(n \log n)$  durch räumliche Partitionierung <sup>47</sup>. Nur Objektpaare mit überlappenden Bounding Boxes werden an die **Narrow Phase** weitergereicht, wo der präzise GJK-Algorithmus (Gilbert-Johnson-Keerthi) die tatsächliche Kollision zwischen konvexen Geometrien ermittelt. <sup>48</sup>

Unity unterscheidet dabei zwischen physikalischen Kollisionen und Trigger-Events. Während physikalische Kollisionen Reaktionskräfte berechnen, detektieren Trigger lediglich Überlappungen und generieren entsprechende Events <sup>49</sup>. Hier kommen lediglich Trigger-Collider zum Einsatz, da sich sonst die Umgebung durch physikalische Wechselwirkungen während der Simulation verändern würde. Die Layer-Matrix ermöglicht zusätzlich eine selektive Kollisionserkennung, indem bestimmte Layer-Kombinationen von der Prüfung ausgeschlossen werden können. Dies reduziert sowohl die Rechenlast als auch unerwünschte Kollisionsmeldungen zwischen semantisch nicht relevanten Objekten.

#### Praktisches Vorgehen

Für die Kollisionserkennung eines Roboterarms mit sechs Freiheitsgraden müssen spezielle Anpassungen vorgenommen werden: Da benachbarte Glieder der kinematischen Kette konstruktionsbedingt immer in Kontakt stehen, würden diese ohne Filterung kontinuierlich Kollisionen melden. Unity's `Physics.IgnoreCollision()` API ermöglicht das explizite Ausschließen solcher Kollisionspaare. Das Layer-System wird zur semantischen Trennung verschiedener Objektkategorien genutzt. Der Roboter operiert auf dem Standard-Layer, während Werkstücke auf Layer 30 (Parts) und Stationen auf Layer 31 (ProcessFlow) (siehe Kapitel 3.2.1) platziert werden. Durch Konfiguration der Collision Layer Matrix wird sichergestellt, dass nur sicherheitsrelevante Kollisionen zwischen Roboter und Hindernissen detektiert werden, während Process Flow Trigger-Events separat behandelt werden.

---

<sup>46</sup>Vgl. NVIDIA Corporation 2023

<sup>47</sup>Vgl. Ericson 2004, S. 14

<sup>48</sup>Vgl. Ericson 2004, S. 399 ff.

<sup>49</sup>Vgl. Unity Technologies 2025

## Konkrete Implementierung

Die Implementierung nutzt Trigger-Collider für berührungslose Detektion. Jedes Roboterglied erhält einen konvexen Mesh-Collider mit einem dedizierten Detector-Komponenten. Die Filterung benachbarter Glieder erfolgt durch hierarchische Analyse der Roboterstruktur, dargestellt in 3.

```
private void SetupAdjacentFrameIgnoring()
{
    // Dynamically ignore collisions between adjacent frames in the kinematic chain
    Frame[] frames = GetComponentsInChildren<Frame>();

    // Sortieren der Hierarchie der Frame
    System.Array.Sort(frames, (a, b) =>
        ↪ GetHierarchyDepth(a.transform).CompareTo(GetHierarchyDepth(b.transform)));

    for (int i = 0; i < frames.Length - 1; i++)
    {
        Transform currentFrame = frames[i].transform;
        Transform nextFrame = frames[i + 1].transform;
        // Methode sucht nach Collidern im GameObject und setzt
        // Physics.IgnoreCollision
        IgnoreCollisionsBetweenParts(currentFrame, nextFrame);
    }
}
```

Abbildung 3: Algorithmus zum Ermitteln kinematisch benachbarter Mesh-Collider

Tritt nun eine Kollision mit einem der vordefinierten Objekte auf, wird die `OnRobotPartCollision()`-Methode des Monitors aufgerufen. Diese prüft zunächst, ob es sich um ein gegriffenes Werkstück handelt - solche Kollisionen werden ignoriert, da das Werkstück temporär Teil des Roboters ist. Anschließend wird durch einen Cooldown-Mechanismus verhindert, dass dieselbe Kollision innerhalb kurzer Zeit mehrfach gemeldet wird.

Die Kollisionserkennung unterscheidet zwischen kritischen und unkritischen Ereignissen basierend auf konfigurierbaren Tags. Objekte mit Tags wie "Machine" oder "Obstacles" lösen kritische Safety Events aus, während andere Kollisionen als Warnungen klassifiziert werden. Der Monitor generiert dabei ein `SafetyEvent`-Objekt mit vollständigen Metadaten über die Kollision, einschließlich der beteiligten Roboterglieder, des Kollisionspunkts in Weltkoordinaten und der Distanz zwischen den Objekten.

Diese Ereignisse werden an den zentralen `RobotSafetyManager` weitergeleitet, der sie mit dem aktuellen Roboterzustand anreichert und entsprechend der konfigurierten Logging-Strategie verarbeitet. Durch diese Architektur bleibt der Collision Detection Monitor unabhängig vom spezifischen Robotertyp und kann flexibel in verschiedenen Simulationsszenarien eingesetzt werden.

Die protokollierten Events bestehen jeweils aus dem Namen des Monitors, einer textuellen Beschreibung, den beteiligten Objekten sowie einem Schweregrad (critical oder warning). Diese Struktur ermöglicht eine klare Einordnung der Ereignisse in Kapitel 4.3 im Ergebnisteil.

### 3.2.3 Singularity Detection Monitor

#### Theoretische Grundlagen der Singularitätsdetektion

Kinematische Singularitäten stellen ein fundamentales Problem in der Robotersteuerung dar und treten auf, wenn die Jacobi-Matrix des Roboters ihren vollen Rang verliert. In diesen Konfigurationen verliert der Roboter die Fähigkeit, sich in bestimmte Richtungen im kartesischen Raum zu bewegen, was zu Kontrollverlust und potentiell gefährlichen Situationen führen kann.<sup>50</sup>

**Mathematische Definition** Eine kinematische Singularität tritt auf, wenn die Jacobi-Matrix  $\mathbf{J}(\boldsymbol{\theta})$ , welche des Roboters ihren vollen Rang verliert:

$$\text{rank}(\mathbf{J}(\boldsymbol{\theta})) < \min(m, n) \quad (3.1)$$

wobei  $\mathbf{J}(\boldsymbol{\theta}) \in \mathbb{R}^{m \times n}$  die Jacobi-Matrix,  $\boldsymbol{\theta}$  der Gelenkwinkelvektor,  $m$  die Anzahl der Freiheitsgrade im kartesischen Raum und  $n$  die Anzahl der Roboterelenke darstellt. Die Jacobi-Matrix beschreibt die Beziehung zwischen Gelenkgeschwindigkeiten  $\dot{\boldsymbol{\theta}}$  und kartesischen Geschwindigkeiten des TCP  $\mathbf{v}$ :

$$\mathbf{v} = \mathbf{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}} \quad (3.2)$$

Tritt eine Singularität auf, so wird die Jacobi-Matrix singulär.

$$\det(\mathbf{J}) = 0 \quad (3.3)$$

Dadurch wird die inverse Kinematik nicht eindeutig lösbar ist und es können theoretisch unendliche Gelenkgeschwindigkeiten auftreten.<sup>51</sup>

#### Frameworkspezifisches Vorgehen

Im Rahmen der praktischen Implementierung wird hier ein auf Schwellwerten und absoluten sowie winkelbasierten Entfernungen zur Singularität angewendet. Durch die Anwendung der rein mathematischen Definition liesse sich zwar jede beliebige Singularität in einer seriellen Kinematik erkennen, jedoch bleiben die betroffenen Gelenke und Art der Singularität ungewiss und müssten iterativ bestimmt werden. Daher wird hier eine pragmatische Berechnungsmethodik verwendet.

---

<sup>50</sup>Vgl. Siciliano und Khatib 2016

<sup>51</sup>Vgl. Nakamura 1991



Für serielle Roboter manipulatoren mit sechs Freiheitsgraden wie den hier verwendeten ABB IRB 6700 können drei primäre Singularitätstypen unterschieden werden: Schultersingularitäten, Ellbogensingularitäten und Handgelenksingularitäten.<sup>52</sup>

**Schultersingularitäten** treten auf, wenn sich das Handgelenkszentrum (Schnittpunkt der Achsen 4, 5, 6) direkt über oder nahe der Rotationsachse des ersten Gelenks befindet:

$$\sqrt{x_{wc}^2 + y_{wc}^2} < \epsilon \quad (3.4)$$

wobei  $(x_{wc}, y_{wc})$  die Position des Handgelenkszentrums in der XY-Ebene und  $\epsilon$  der Singularitätsschwellenwert ist.

**Ellbogensingularitäten** treten auf, wenn der Roboter die Grenzen seines Arbeitsraums erreicht. Dies geschieht typischerweise bei vollständig ausgestreckter oder eingeklappter Konfiguration. Beim Knickarmrobotern wie dem IRB 6700 verläuft die Rotationsachse des vierten Gelenks nicht direkt durch den Ursprung vom dritten Gelenk, sondern ist durch eine Translation verschoben. Das impliziert, dass die vollständige Streckung des Armes nicht durch einen Winkel von  $0^\circ$  bzw.  $180^\circ$  auftritt. Hier lässt sich allgemeiner der Zusammenhang der Vektoren zwischen Gelenk 2 und 3 sowie 2 und 5 anwenden:

$$\theta = \angle(\vec{v}_{23}, \vec{v}_{25}) \approx 0^\circ \text{ oder } \theta \approx 180^\circ \quad (3.5)$$

wobei:

$$\vec{v}_{23} = \vec{p}_3 - \vec{p}_2 \quad (3.6)$$

$$\vec{v}_{25} = \vec{p}_5 - \vec{p}_2 \quad (3.7)$$

mit  $\vec{p}_i$  als Position des Gelenks  $i$ . Die Singularitätsbedingung lautet:

$$\theta < \epsilon \quad \text{oder} \quad \theta > 180^\circ - \epsilon \quad (3.8)$$

**Handgelenksingularitäten** entstehen, wenn die Rotationsachsen der letzten drei Gelenke (Gelenke 4, 5, 6) kollinear werden. Dies tritt typischerweise auf, wenn  $\theta_5 = 0^\circ$  oder  $\theta_5 = 180^\circ$ . Mathematisch beschrieben durch:

$$\mathbf{z}_4 \parallel \mathbf{z}_6 \text{ oder } |\mathbf{z}_4 \cdot \mathbf{z}_6| \approx 1 \quad (3.9)$$

wobei  $\mathbf{z}_i$  die Rotationsachse (Z-Achse) des  $i$ -ten Gelenks im Weltkoordinatensystem darstellt.

---

<sup>52</sup>Vgl. Spong et al. 2006

**Manipulierbarkeitsindex (Yoshikawa-Maß)** Der von Yoshikawa Yoshikawa 1985 eingeführte Manipulierbarkeitsindex ist eine der am häufigsten verwendeten Metriken:

$$\mu(\boldsymbol{\theta}) = \sqrt{\det(\mathbf{J}(\boldsymbol{\theta})\mathbf{J}^T(\boldsymbol{\theta}))} \quad (3.10)$$

Für quadratische Jacobi-Matrizen vereinfacht sich dies zu:

$$\mu(\boldsymbol{\theta}) = |\det(\mathbf{J}(\boldsymbol{\theta}))| \quad (3.11)$$

Der Index nimmt Werte zwischen 0 (Singularität) und einem maximalen Wert an, wobei höhere Werte bessere Manipulierbarkeit indizieren. Die Robotik-Literatur bietet verschiedene Ansätze zur Behandlung von Singularitäten, die sich in präventive und reaktive Strategien unterteilen lassen. Um diese in der Praxis anzuwenden, müssen jedoch teilweise numerische Auswertungen durchgeführt werden, um Schwellwerte zu erfassen. Daher kommen diese hier nicht zum Einsatz.

Zur zusätzlichen Charakterisierung wird jedoch die Yoshikawa-Manipulierbarkeit berechnet, da der Wert als Kennzahl für die Nähe zu einer Singularität verwendet werden kann. Er wird in den Ergebnissen in Kapitel 4.4 aufgeführt.

### Praktisches Vorgehen

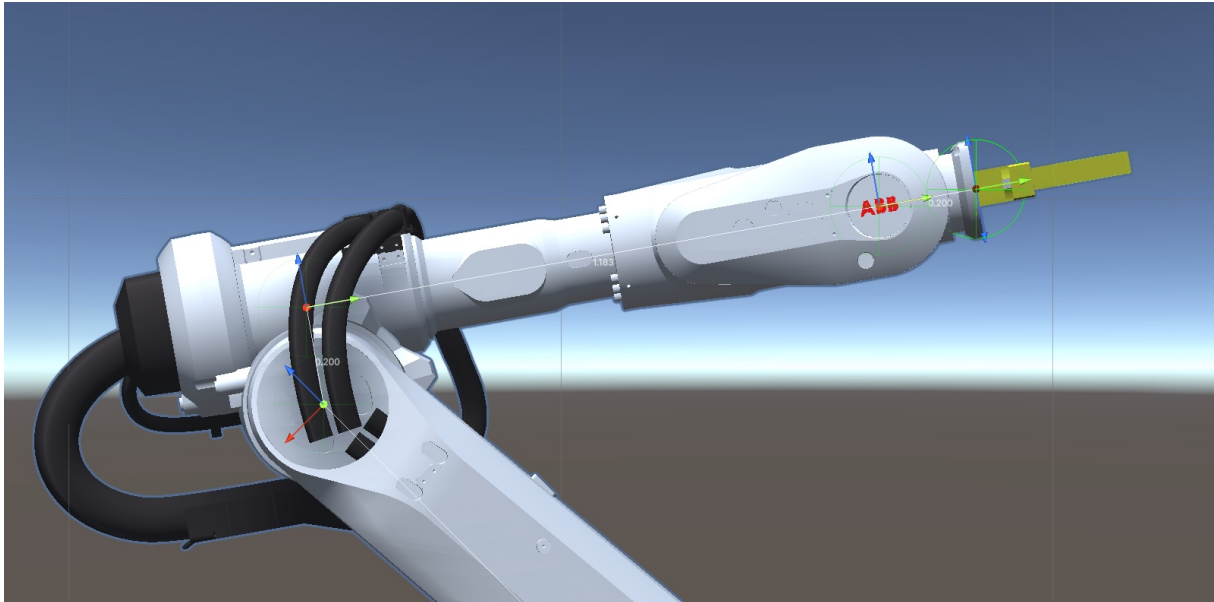


Abbildung 4: Screenshot einer Handgelenksingularität in Unity mit farbig dargestellten Koordinatensystemen der DH-Transformationen

**Schritt 1: Positionsberechnung und Achsentransformation**

Die Positionen der relevanten Gelenke werden durch die Anwendung der Vorwärtstransformation nach Denavit-Hartenberg berechnet:

$$\vec{p}_i = \text{DH}(\theta_1, \dots, \theta_i) \quad \text{für } i = 2, 3, 5 \quad (3.12)$$

Die Rotationsachsen werden aus den Transformationsmatrizen extrahiert:

$$\mathbf{z}_i = \mathbf{T}_i[:, 2] \quad \text{für } i = 1, 4, 6 \quad (3.13)$$

**Schritt 2: Singularitätsanalyse**

Für jeden Singularitätstyp wird die entsprechende Bedingung überprüft:

- **Schultersingularität:**

$$d_{wc} = \sqrt{x_{wc}^2 + y_{wc}^2} \quad (3.14)$$

wobei  $(x_{wc}, y_{wc})$  die Position des Handgelenkszentrums in der XY-Ebene ist.

- **Ellbogensingularität:**

$$\theta_{elbow} = \angle(\vec{p}_3 - \vec{p}_2, \vec{p}_5 - \vec{p}_2) \quad (3.15)$$

- **Handgelenksingularität:**

$$c_{46} = |\mathbf{z}_4 \cdot \mathbf{z}_6| \quad (3.16)$$

**Schritt 3: Schwellwertvergleich**

Singularitätstyp	Beispiel-Schwellwert	Bedingung
Schulter	$\tau_{\text{shoulder}} = 100 \text{ mm}$	$d_{wc} < \tau_{\text{shoulder}}$
Ellbogen	$\tau_{\text{elbow}} = 5^\circ$	$\theta_{\text{elbow}} < \tau_{\text{elbow}}$ oder $\theta_{\text{elbow}} > 180^\circ - \tau_{\text{elbow}}$
Handgelenk	$\tau_{\text{wrist}} = 5^\circ$	$c_{46} > \tau_{\text{wrist}}$

Tabelle 3: Singularitätsschwellwerte und Detektionsbedingungen

**Schritt 4: Manipulierbarkeitsberechnung**

Für detektierte Singularitäten wird ein approximierter Manipulierbarkeitsindex nach Yoshikawa berechnet:

$$w = \sqrt{\det(\mathbf{J}(\theta)\mathbf{J}^T(\theta))} \quad (3.17)$$

Ein kleiner Wert von  $w$  indiziert die Nähe zu einer singulären Konfiguration. Das entwickelte Framework implementiert eine winkelbasierte Singularitätsdetektion, die geometrische Eigenschaften der Roboterkinematik direkt nutzt, anstatt auf rechenintensive Jacobi-Matrix-

Berechnungen angewiesen zu sein. Die achsenbasierte Methode basiert auf der Erkenntnis, dass Handgelenks- und Ellbogensingularitäten geometrisch durch die Kollinearität von Rotationsachsen charakterisiert werden können. Schulter singularitäten werden analog durch die Entfernung des ersten Gelenks zum Handgelenkzentrum in der XY-Ebene festgestellt.

### Konkrete Implementierung der Detektionsmethoden

Die Implementierung des SingularityDetectionMonitor nutzt die Vorwärtskinematik des Preliy Flange Frameworks zur Berechnung der Gelenkpositionen. Die zentrale Methode ComputeJointPosition berechnet die kartesische Position eines beliebigen Gelenks durch sukzessive Anwendung der Denavit-Hartenberg-Transformationsmatrizen:

```
private Vector3 ComputeJointPosition(float[] jointAngles, int jointIndex)
{
    Matrix4x4 baseTransform = Matrix4x4.identity;

    // Apply joint transformations to get to desired joint position
    // To get joint N position, apply transformations 0 to N-1
    for (int i = 0; i < jointIndex - 1 && i < robotFrames.Length - 1; i++)
    {
        var frame = robotFrames[i + 1]; // Frame i+1 corresponds to joint i
        var config = frame.Config;

        // Create transformation matrix using DH parameters
        float theta = jointAngles[i] * Mathf.Deg2Rad + config.Theta;
        Matrix4x4 dhTransform = HomogeneousMatrix.CreateRaw(new FrameConfig(
            config.Alpha, config.A, config.D, theta
        ));

        baseTransform = baseTransform * dhTransform;
    }

    return baseTransform.GetPosition();
}
```

Abbildung 5: Vorwärtskinematik zur Positionsberechnung

Die Methode ComputeJointPosition in Abbildung 5 implementiert die klassische Vorwärtskinematik durch Multiplikation homogener Transformationsmatrizen. Jede Matrix  $\mathbf{T}_i$  wird aus den DH-Parametern  $(\alpha_i, a_i, d_i, \theta_i)$  konstruiert, wobei  $\theta_i$  der aktuelle Gelenkwinkel plus einem konstanten Offset ist. Die resultierende Transformationsmatrix beschreibt die Position und Orientierung des Gelenks im Basiskoordinatensystem.

Das Framework nutzt Unitys Matrix4x4-Klasse für die Transformationsberechnungen und die GetPosition()-Methode zur Extraktion der Translationskomponente. Die Koordinatentransformation zwischen dem DH-Parametersystem und Unitys linkshändigem Y-up Koordinatensystem wird dabei durch die Methode HomogeneousMatrix.CreateRaw() des Flange-Frameworks

transparent gehandhabt. Das ermöglicht eine nahtlose Integration der mathematischen Robotik-Konzepte in die Unity-Umgebung, während die Echtzeitfähigkeit durch eventgetriebene Berechnung gewährleistet ist. Bei der Detektion einer Singularität bzw. dem Unterschreiten des im Unity-Editor definierten Grenzwertes wird ein `SafetyEvent`-Objekt instanziiert und an die `SafetyMonitor` Klasse weitergegeben. Darin werden zusätzlich Event-Metadaten zu den Daten der Gelenkwinkelabstände ausgegeben. Sobald der kritische Bereich verlassen wurde, wird ein weiteres `SafetyEvent` ausgegeben, um den Bereich, in welcher die Singularität auftritt, abstecken zu können.

### 3.2.4 Joint Dynamics Monitor

**Theoretische Grundlagen** Die Überwachung kinematischer Grenzwerte während der Werkstückhandhabung stellt eines der zentralen Ziele des Frameworks dar. Die besondere Herausforderung in HTTP-basierten Systemen liegt in der diskreten Datenakquisition mit Polling-Intervallen von 50-1000ms, wodurch die numerische Differentiation zu erheblichen Instabilitäten führt.

Geschwindigkeit  $\dot{\theta}_i$  und Beschleunigung  $\ddot{\theta}_i$  können folgendermassen aus der diskreten Zeitreihe approximiert werden:

$$\dot{\theta}_i[k] = \frac{\theta_i[k] - \theta_i[k-1]}{\Delta t_k}, \quad \ddot{\theta}_i[k] = \frac{\dot{\theta}_i[k] - \dot{\theta}_i[k-1]}{\Delta t_k} \quad (3.18)$$

In der Praxis und nach initialem Testen hat sich herausgestellt, dass die Berechnung dessen durch die unstetige Bewegung unrealistische Werte liefert. Das hängt damit zusammen, dass der HTTP-Endpunkt des Robotercontrollers in festgelegten Zeitabständen abgefragt wird und so die Gelenkpositionen des Roboters erneuert werden. Bewegt sich der Roboter über eine bestimmte Strecke, wird die von Unity lediglich als unstetige Bewegung mit theoretisch unendlich grosser Geschwindigkeit wahrgenommen. Daher ist hier eine Glättung der Werte nötig, um eine stetige Bewegung zu simulieren.

**Mehrstufiger Filteransatz** Zur Stabilisierung implementiert das Framework einen dreistufigen Filteransatz:

1. **Exponential Moving Average (EMA):**  $\dot{\theta}_{i,EMA}[k] = \alpha \cdot \dot{\theta}_{i,raw}[k] + (1 - \alpha) \cdot \dot{\theta}_{i,EMA}[k-1]$  mit  $\alpha = 0.2$
2. **Moving Window Average:**  $\dot{\theta}_{i,MW}[k] = \frac{1}{N} \sum_{j=0}^{N-1} \dot{\theta}_{i,EMA}[k-j]$  mit Fenstergröße  $N = 8$
3. **Outlier Rejection:**  $|\dot{\theta}_{i,final}[k] - \dot{\theta}_{i,final}[k-1]| \leq \tau_v \cdot \dot{\theta}_{i,max}$  mit  $\tau_v = 0.2$

Durch die verwendete Glättung werden schnelle Änderungen in den Gelenkwerten erst zeitverzögert sichtbar. Dies wirkt sich unmittelbar auf die Erkennung von Geschwindigkeitsverletzungen aus, da der Monitor die Ereignisse nicht exakt im Moment des Überschreitens ausgibt, sondern nach einigen Abtastungen.

**Grenzwertdefinition** Die Grenzwerte werden aus den Flange-Framework-Konfigurationen extrahiert oder manuell spezifiziert. Ein Sicherheitsfaktor  $\lambda = 0.8$  reduziert die nominellen Maximalwerte präventiv. Für den ABB IRB 6700 ergeben sich beispielsweise Geschwindigkeitsgrenzen von 88°/s für die Hauptachsen und bis zu 168°/s für das Handgelenk.

**Implementierung der Smoothing-Pipeline** Die zentrale Smoothing-Methode kombiniert die Filteransätze:

```
private float[] SmoothVelocities(float[] rawVelocities)
{
    // Hinzufügen der neuen Geschwindigkeiten zum Buffer und Buffer anpassen
    velocityBuffer.Add((float[])rawVelocities.Clone());
    if (velocityBuffer.Count > smoothingWindowSize) {
        velocityBuffer.RemoveAt(0);
    }
    float[] result = new float[6];
    for (int i = 0; i < 6; i++) {
        // Glätten mit dem EMP
        smoothedVelocities[i] = (smoothingAlpha * rawVelocities[i]) + ((1f -
        ↪ smoothingAlpha) * smoothedVelocities[i]);
        // Anwenden des Moving Window Average
        if (velocityBuffer.Count >= smoothingWindowSize) {
            float windowSum = 0f;
            for (int j = 0; j < velocityBuffer.Count; j++) {
                windowSum += velocityBuffer[j][i];
            }
            result[i] = windowSum / velocityBuffer.Count;
        }
        else
        { result[i] = smoothedVelocities[i]; }
        // OutlierRejection
        float maxChange = maxJointVelocities[i] * velocityOutlierThreshold;
        if (velocityBuffer.Count > 1) {
            // GetMovingAverage berechnet den gleitenden Durschnitt zwischen der
            // letzten und vorletzten Bufferposition
            float previousSmoothed = velocityBuffer.Count >= 2 ?
                GetMovingAverage(velocityBuffer, i, velocityBuffer.Count - 1) :
                ↪ smoothedVelocities[i];
            float change = Mathf.Abs(result[i] - previousSmoothed);
            // Überschreitet die Änderung der Geschwindigkeit den Schwellwert,
            // wird er auf den maximalen positiven oder negativen Wert begrenzt
            if (change > maxChange) {
                result[i] = previousSmoothed + Mathf.Sign(result[i] - previousSmoothed) *
                ↪ maxChange; }
        }
    }
    return result;
}
```

Abbildung 6: Mehrstufige Smoothing-Pipeline zur Geschwindigkeitsberechnung

Als weiteres Feature implementiert der Monitor eine Methode, welche sich an die Update-Rate der State-Updates (also die Rate neuer Gelenkdaten) anpasst. So kann garantiert werden, dass

ein State nicht doppelt verarbeitet und so beispielsweise die Geschwindigkeit falsch interpretiert wird, da es sich um den immer gleichen State des Roboters handelt.

Bei der Überschreitung der Schwellwerte wird der SafetyMonitor mittels eines SafetyEvents benachrichtigt. Als zusätzliche Metadaten enthält diese Meldung die Eventart (Geschwindigkeits- oder -Beschleunigungsüberschreitung) und die damit zusammenhängenden Schwellwerte und aktuellen Werte. Da es in dem Rahmen vorkommen kann, dass hier die Geschwindigkeit über einen längeren Zeitraum überschritten wird, gibt der Monitor ebenfalls ein SafetyEvent aus, wenn die Geschwindigkeit unter den gegebenen Schwellwert sinkt.

### 3.3 Testumgebung und -setup

#### 3.3.1 Aufbau der Roboterzelle

##### Arbeitsraum-Layout

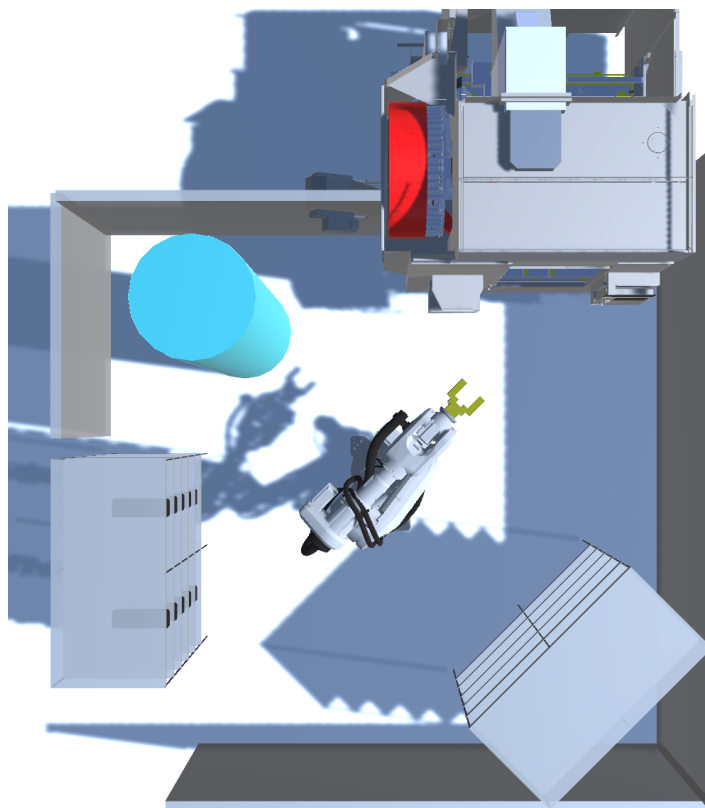


Abbildung 7: Arbeitsraum-Layout des Roboters in der Draufsicht mit Roboter in Home-Position

Die Roboterzelle ist im Rahmen dieses Testszenarios als isolierter Arbeitsraum mit mehreren Hindernisse, Begrenzungen und 3 semantischen Stationen aufgebaut. Die vom Roboter zu bewältigenden Aufgaben beschränken sich dabei auf ein Pick-and-Place-Szenario von Zylinderköpfen, dargestellt in Abbildung 7. Das hierfür erstellte Layout sieht den Roboter in Zentrum dem

Arbeitsraumes auf dem Boden stehend vor. Hindernisse, Begrenzungen und Stationen sind kreisförmig um den Roboter angeordnet. In Abbildung 7 blau markiert, befindet sich eine Säule als künstliches Hindernis. Links und rechts unten vom Mittelpunkt der Zelle aus befinden sich zwei Regale als Teilelager. Oben befindet sich eine 5-Achs-Fräsmaschine mit einer offenen Teileablage durch Entfernung der Schutztür zu Vereinfachungszwecken dieses Szenarios. Der Arbeitsraum ist begrenzt durch halbdurchsichtige Wände. Der Prozessablauf sieht vor, dass der Roboter Zylinderköpfe aus dem linken Regal aufnimmt, diese in der Maschine ablegt, in einer Warteposition auf die Beendigung der Bearbeitung wartet und anschliessend das bearbeitete Teil aufnimmt, um es im rechten Regal abzulegen. Ein beispielhafter Bearbeitungsprozess mit der 5-Achs-Fräsmaschine ist hier das Bohren und Schneiden von Gewinden in die Zylinderköpfe.

Der Aufbau und die Anordnung des Arbeitsraums ergibt sich aus den funktionalen Anforderungen, die das Framework abdecken soll. Durch den relativ begrenzten Raum müssen Bewegungspfade des Roboters an Hindernissen wie der Säule vorbei geplant werden und etwaige durch den Roboter gegriffene Werkstücke mitberechnet werden. Weiterführend verringern die Regale den Bewegungsraum für den Roboter zur Umorientierung vor und nach dem Aufnehmen und Ablegen der Teile, was zu einer erhöhten Wahrscheinlichkeit von Singularitäten führt. Die 3 semantischen Stationen sorgen dafür, dass sich hier Fehler im Prozessablauf abbilden lassen. Die Achsgeschwindigkeiten lassen sich in diesem Szenario ebenfalls beliebig variieren. Somit lassen sich alle funktionalen Anforderungen des Frameworks innerhalb dieses vereinfachten Szenarios abbilden.



## Zylinderköpfe als Werkstücke

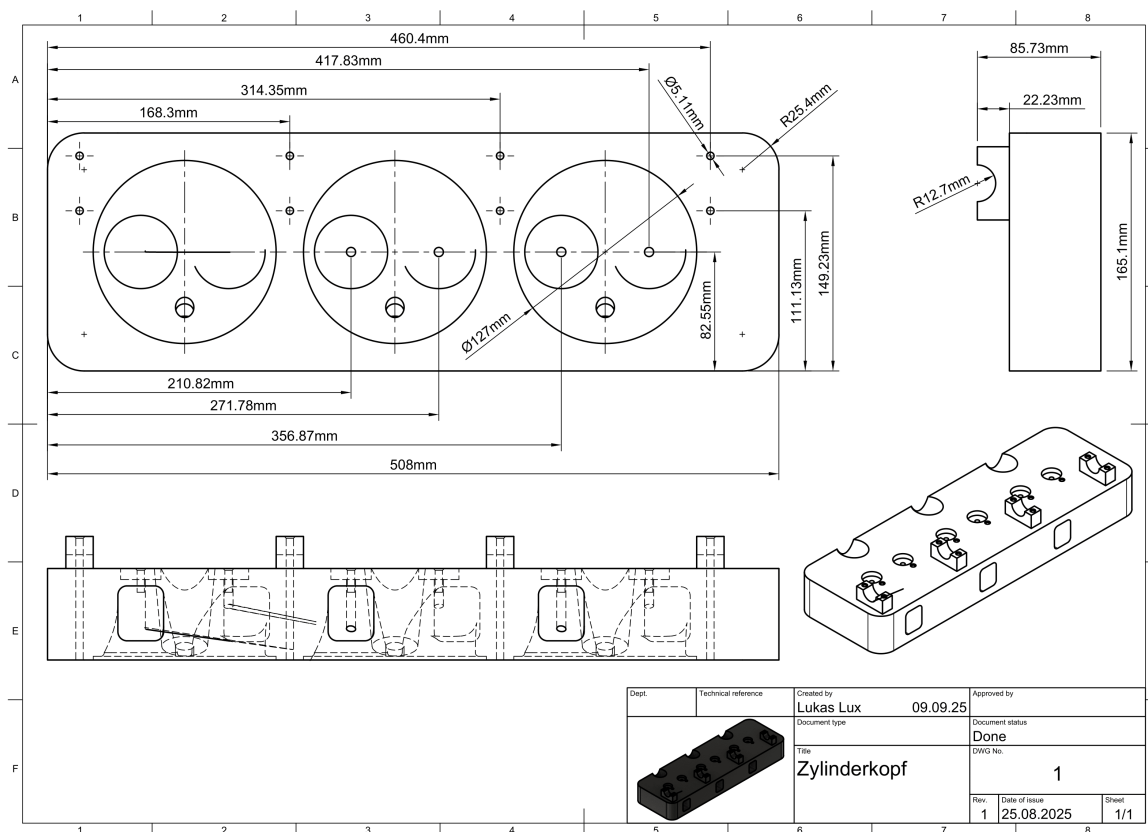


Abbildung 8: Technische Zeichnung des verwendeten Zylinderkopfes, dargestellt mithilfe von Autodesk Fusion 360

Als Werkstücke werden hier Zylinderköpfe für V6-Motoren gewählt. Das Werkstück bringt ein mit den Spezifikationen des ABB IRB 6700 übereinstimmendes Gewicht mit und ist an den Seitenflächen durch einen Parallelgreifer gut greifbar. Die Abmasse, des Zylinderkopfes sind Abbildung 8 zu entnehmen.

## Greiferauslegung

Um das Werkstück sicher handhaben zu können, wird zusätzlich zum Roboter auch End-Effektor benötigt, welcher das Werkstück sicher durch den Arbeitsraum bewegen kann. Für die Handhabung des Zylinderkopfs wurde speziell ein Parallelgreifer ausgelegt, da die Entscheidung für einen geeigneten Greifer komplex ist und das Anforderungsprofil kaum normierbar ist<sup>53</sup>.

<sup>53</sup>Vgl. Hesse 2011, S. 91



der Robotik eine systematische Beschreibung der Geometrie serieller Robotermechanismen ermöglicht und somit die Anwendung etablierter algorithmischer Verfahren für kinematische Berechnungen, Jacobi-Matrizen sowie Bewegungsplanung unterstützt.<sup>56</sup>

Flange ermöglicht eine direkte Konfiguration des Roboters über *Frame* und *JointTransformation* Scripts. Ein *Frame* definiert dabei die Denavit-Hartenberg-Parameter für einen Teil der kinematischen Kette, eine *JointTransformation* die Parameter des Gelenks, also möglicher maximaler Ausschlag in positive und negative Achsrotationsrichtung sowie maximale Geschwindigkeit und Beschleunigung. Diese Funktionen werden im Rahmen dieser Implementierung genutzt, um den zu simulierenden Roboter theoretisch nachvollziehbar im Raum bewegen zu können.

### 3.3.3 Modellierung in Unity

Der Arbeitsraum wurde der vorangegangenen Darstellung entsprechend in Unity modelliert. Dabei wurden die von Unity standardmäßig gewählten Einstellungen verwendet, sowohl bei der Modellierung als auch im Kontext der Physik-Engine. Jegliche Hindernisse wurden dabei mit *Collidern* versehen und dem Tag *textttObstacle* und respektive *Machine* für die Maschine, um diese bei der späteren Auswertung als Hindernisse kenntlich zu machen. Weiterführend wurden den einzelnen Stationen (linkes und rechtes Regal, Maschine) das Script *Station.cs* angehängt und dort im Sinne des bereits beschriebenen Arbeitsablaufs ein steigender Index zugewiesen. Analog dazu wird allen Zylinderköpf-Objekte das Script *Part.cs* zugewiesen, welche diese als Werkstücke identifiziert und mit dem durch Drag-and-Drop im Unity Inspector die Reihenfolge der einzelnen Stationen definiert wird, welche das Teil durchlaufen muss.

Alle Monitore wurden in einer einheitlichen Unity-Testumgebung ausgeführt. Die Simulationen basieren auf einem digitalen Zwilling des Roboters, der kontinuierlich seine Gelenkwinkel und Zustände an die Monitore übermittelt. Die erkannten Ereignisse werden unmittelbar protokolliert und in eine JSON-Logdatei geschrieben. Auf diese Weise sind die Ergebnisse der verschiedenen Monitore konsistent dokumentiert und im Ergebniskapitel direkt vergleichbar.

---

<sup>56</sup>Vgl. Corke 2007, S. 590

### 3.4 Datenaufzeichnung und Logging

```
public SafetyEvent(string monitorName, SafetyEventType eventType, string description,
    ↳ RobotState currentState)
{
    this.monitorName = monitorName;
    this.eventType = eventType;
    this.description = description;
    this.timestamp = DateTime.Now;
    this.robotStateSnapshot = currentState != null ? new
    ↳ RobotStateSnapshot(currentState) : null;
}
```

Abbildung 10: SafetyEvent-Klasse mit Zuordnung zum Monitor und Kritikalität

Als zentrales Aufzeichnungsformat implementiert der SafetyMonitor eine JSON-Struktur für die durch die einzelnen Module erzeugten Ereignisse. Die Key-Value-Paare dieser Objekte sind durch die Klassen RobotStateSnapshot und SafetyEvent vordefiniert. Wird ein Ereignis ausgelöst, so wird ein neues SafetyEvent instanziiert (vgl. Abbildung 10).

Der RobotStateSnapshot kann initial leer sein, da durch Multithreading nicht in jedem Fall eine direkte Referenz zum aktuellen Roboterzustand vorliegt. Über die Methode SetEventData() der SafetyEvent-Klasse werden anschließend die spezifischen Daten ergänzt, beispielsweise die Kollisionsposition bei einer Kollisionserkennung.

```
public RobotStateSnapshot(RobotState state)
{
    if (state != null)
    {
        captureTime = DateTime.Now;
        // Program info
        isProgramRunning = state.isRunning;
        currentModule = state.currentModule ?? "";
        currentRoutine = state.currentRoutine ?? "";
        currentLine = state.currentLine;
        currentColumn = state.currentColumn;
        executionCycle = state.executionCycle ?? "";
        motorState = state.motorState ?? "";
        controllerState = state.controllerState ?? "";
        // Abrufen der aktuellen Achswinkel
        jointAngles = state.GetJointAngles();
        hasValidJointData = state.hasValidJointData;
        motionUpdateFrequencyHz = state.motionUpdateFrequencyHz;
        // IO Signale
        gripperOpen = state.GripperOpen;
        // Verbindungsinformationen
        robotType = state.robotType ?? "";
        robotIP = state.robotIP ?? "";
    }
}
```

Abbildung 11: RobotStateSnapshot-Klasse mit formalisieren Attributen

Der `SafetyMonitor` ergänzt das Ereignis um den aktuellen `RobotStateSnapshot`, der zusätzliche Informationen wie aktuelle Achswinkel, Motordaten, Programmzeiger und Programmname enthält (vgl. Abbildung 11). Die Events werden zwischengespeichert und am Ende der Ausführung gesammelt als `.json`-Datei in einem vordefinierten Ordner abgelegt.

## **Einordnung im Gesamtkontext**

Die Architektur des Frameworks besteht somit aus den genannten Modulen, die in Kapitel 3 im Detail beschrieben werden. Ihre Funktionsweise wird in Kapitel 4 anhand von Testszenarien überprüft und ausgewertet.

## 4 Validierungsergebnisse der Testszenarien

### 4.1 Überblick und Zielsetzung

In diesem Kapitel werden die Ergebnisse der im vorherigen Kapitel beschriebenen Implementierung vorgestellt. Im Fokus steht die Überprüfung der vier entwickelten Safetymodule – Prozessfolgenüberwachung, Kollisionserkennung, Achsgeschwindigkeits- und Beschleunigungsüberwachung sowie Singularitätserkennung – innerhalb der aufgebauten Simulationsumgebung. Ziel ist es zu überprüfen, inwiefern im gewählten Testsetup eine Erkennung von Fehlern in der Roboterbewegung und Interaktion, welche mit den vier untersuchten Parametern zusammenhängen, auftreten. Ziel ist es zu überprüfen, inwiefern im gewählten Testsetup eine Erkennung von Fehlern in der Roboterbewegung und Interaktion, welche mit den vier untersuchten Parametern zusammenhängen, auftreten.

Für jedes Modul wurden gezielte Testfälle definiert, die sowohl korrekte als auch fehlerhafte Szenarien abbilden, um die Funktionsweise und Zuverlässigkeit der Module zu überprüfen. Die Ergebnisse werden anhand von Beobachtungen aus der Simulation, gespeicherten Zustands- und Ereignisdaten sowie grafischen Darstellungen aufgezeigt. Die Testfälle wurden dabei als Pfade in RobotStudio definiert. Diese Pfade werden von RobotStudio in RAPID-Code umgewandelt und gespeichert. Durch die Synchronisation mit dem Controller und dem Setzen des aktuellen Programms als Standardprogramm lässt sich das Programm in RobotStudio simulieren. So wurde für jedes Szenario vorgegangen.

Zur zusätzlichen Validierung wurde ein Experteninterview durchgeführt, in dem die Testcases vorgestellt und die Funktionsweise des Frameworks diskutiert wurden. Die Aussagen des Experten werden an geeigneter Stelle in diesem Kapitel dargestellt und in Kapitel 5 kritisch eingeordnet.

## 4.2 Prozessflussüberwachung

### 4.2.1 Abwandlung der Prozessfolge

Die Validierung der Prozessflussüberwachung erfolgt über die Ausführung eines korrekten Szenarios als auch eines fehlerhaften Szenarios. Der Fehler soll provoziert werden, in dem das Werkstück abweichend zum in Abbildung 12 gezeigten gewünschten Ablauf direkt in das zweite Regal bewegt wird. Semantisch bedeutet dies ein Überspringen der Station *Machine*. Das soll ein SafetyEvent triggern, sobald das Werkstück im falschen Regal abgelegt wird.

#### Korrekter Prozessfluss

1. Bewegung von Home-Position zu linkem Regal
2. Greifen des Werkstücks
3. Bewegung zu Maschine
4. Platzieren des Werkstücks in Maschine
5. Warten auf Beendigung des Bearbeitungsprozesses in Warteposition (Bearbeitung hier nur simuliert)
6. Bewegen zum Werkstück und Greifen aus Maschine
7. Bewegung zu rechtem Regal
8. Platzieren des Objekts in rechtem Regal
9. Rückkehr zur Home-Position

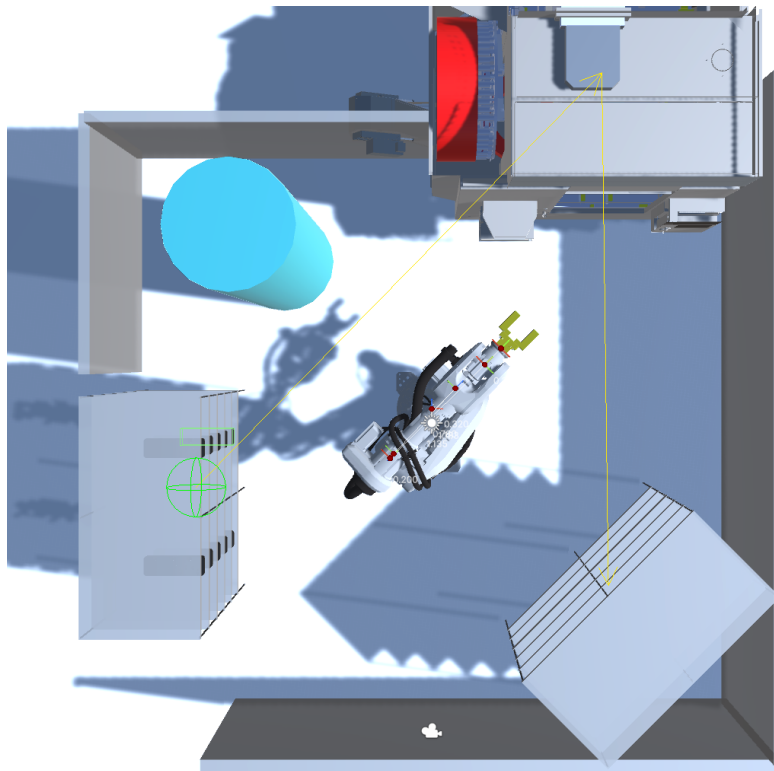


Abbildung 12: Visuelle Darstellung des Prozessflusses durch Konfiguration der Parts und Stations

Im veränderten Prozessfluss wird nun der Schritt der Bearbeitung übersprungen. Dies kann in der Praxis durch eine inkorrekte Abfolge im Roboterprogrammcode passieren. Somit ist der veränderte Prozessfluss wie folgt definiert.

#### Abgewandelter Prozessfluss

1. Bewegung von Home-Position zu linkem Regal
2. Greifen des Werkstücks
3. Bewegung zu Maschine
4. Bewegung zu rechtem Regal
5. Platzieren des Objekts in rechtem Regal
6. Rückkehr zur Home-Position

#### 4.2.2 Simulationsergebnis

Wird der korrekte Prozessfluss abgespielt, werden im Bezug auf den ProcessFlowMonitor keine Ereignisse protokolliert. Bei der Ausführung des Roboterprogramms mit verändertem Prozessfluss findet sich in der nach Beendigung des Programms mit Zeitstempel und Modulname versehenem JSON-Log ein Eintrag mit einem SafetyEvent getriggert durch den Process Flow Monitor:



```

{
  "programName": "Module1",
  "totalSafetyEvents": 1,
  "safetyEvents": [
    {
      "monitorName": "Process Flow Monitor",
      "eventType": 1,
      "description": "Process flow violation: Part 'CylinderHead1' attempted invalid
↪ transition from StorageIn to StorageOut. Expected: Machine. Violation type:
↪ SkippedStation",
      "robotStateSnapshot": {
        "isProgramRunning": true,
        "currentModule": "Module1",
        "currentRoutine": "PlaceinStorageOut",
        "currentLine": 99,
        "currentColumn": 9,
        "executionCycle": "once",
        "motorState": "running",
        "controllerState": "motoron",
        "jointAngles": [
          "...",
        ],
        "hasValidJointData": true,
        "motionUpdateFrequencyHz": 18.038461607840238,
        "gripperOpen": true,
        "robotType": "ABB",
        "robotIP": "127.0.0.1"
      }
    }
  ]
}

```

Abbildung 13: JSON-Log zum Prozessfolgenfehler, Achswinkel wurden im Nachhinein entfernt

In Abbildung 13 ist abzulesen, dass der Fehler hier korrekt erkannt und klassifiziert wurde. Dabei zeigt das Feld `description` den genauen Hergang des Events an. Hier wurde eine invalide Transition von `StorageIn` zu `StorageOut` versucht. Das Framework erkennt ebenfalls, welcher Prozessschritt der Richtige gewesen wäre. Ausserdem wird hier als Violation Type der Type `SkippedStation` angegeben. Dieser ist als übersprungene Station zu klassifizieren, was in diesem Fall korrekt ist. Zusätzlich werden weitere Parameter der Simulation und des Controllers weitergegeben, unter anderem welches Modul und welche Routine des Moduls zum Zeitpunkt des Auftretens ausgeführt wurde sowie welcher Programmzeile der ProgramPointer sich zum Zeitpunkt der Event-Auslösung befand. Durch den Wert des Keys `totalSafetyEvents` ist zu erkennen, wie viele Ereignisse bei der Ausführung vom Programm getriggert wurden. Hier ist es lediglich der oben Genannte.

## 4.3 Auswertung Collision Detection Monitor

### 4.3.1 Abwandlung des Pfads

Zur Auswertung des Collision Detection Monitors wurde der oben bereits genannte Prozess genutzt. Anschliessend wurde der Pfad, auf dem sich der Roboter zwischen seiner Home-Position und dem linken Regal, in dem er ein Werkstück greifen soll, verändert. Rechts dargestellt in Abbildung 14, führt der Pfad im Gegensatz zum kollisionsfreien, näher am Roboter verlaufenden Pfad aufgrund des ausladenden Umschwungs durch die Säulengeometrie. Durch das Abfahren des Pfades gegen den Uhrzeigersinn wird proviziert, dass der Roboter sich durch die in Abbildung 14 links blau eingefärbte Säule hindurch bewegen muss. Die Säule selbst bekommt den Tag `Obstacle` und wird durch einen vorhandenen Collider, welcher den Dimensionen der Säule entspricht, zum Kollisionshindernis.

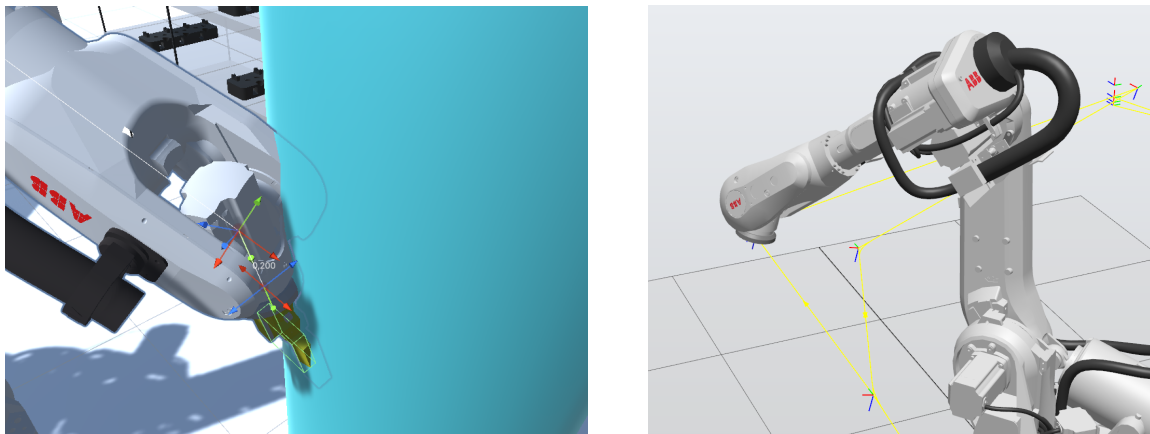


Abbildung 14: Kollision in Unity (links) und zugehörige Position auf Pfad in RobotStudio (rechts). Gelenk 4, 5 und 6 sowie Greifer befinden sich innerhalb der Säulengeometrie.

### 4.3.2 Simulationsergebnis

In Abbildung 15 ist der Output nach Kollision mit der Säule dargestellt. Dem Event wurden zusätzlich Eventdaten angefügt, welche hier den Punkt der Kollision im Unity Koordinatensystem als auch die Entfernung zum Zentrum des kollidierenden Objekts darstellen. Die Kollision wird somit zuverlässig erkannt. Wichtig ist dabei zu erwähnen, dass die Kollisionserkennung stark von den verwendeten Collidern abhängt. Hier verwendet das Framework Mesh-Collider zur genauen Abbildung der Robotergeometrie, das Kollisionobjekt wird durch einen primitiven zylindrischen Collider definiert. Beim Outputformat der `eventDataJson` handelt es sich um string-escaped JSON. Die Daten sind also als Event-Daten in einem String komprimiert.

```
[
  {
    "monitorName": "Collision Detector",
    "eventType": 2,
    "description": "CRITICAL COLLISION: Gripper -> Saele",
    "robotStateSnapshot": "[...]",
    "eventDataJson": "{\"robotLink\":\"Gripper\",\"collisionObject\":\"Saele\",\"colli-
    → sionPoint\":{\"x\":-1.146544337272644,\"y\":0.038220398128032687,\"z\":1.14
    → 65449333190919},\"distance\":1.6219075918197632}"
  },
  {
    "monitorName": "Collision Detector",
    "eventType": 2,
    "description": "CRITICAL COLLISION: Joint_4 -> Saele",
    "robotStateSnapshot": "[...]",
    "eventDataJson": "{\"robotLink\":\"Joint_4\",\"collisionObject\":\"Saele\",\"colli-
    → sionPoint\":{\"x\":-1.146544337272644,\"y\":0.038220398128032687,\"z\":1.14
    → 65449333190919},\"distance\":1.6219075918197632}"
  },
  {
    "monitorName": "Collision Detector",
    "eventType": 2,
    "description": "CRITICAL COLLISION: Joint_6 -> Saele",
    "robotStateSnapshot": "[...]",
    "eventDataJson": "{\"robotLink\":\"Joint_6\",\"collisionObject\":\"Saele\",\"colli-
    → sionPoint\":{\"x\":-1.146544337272644,\"y\":0.038220398128032687,\"z\":1.14
    → 65449333190919},\"distance\":1.6219075918197632}"
  },
  {
    "monitorName": "Collision Detector",
    "eventType": 2,
    "description": "CRITICAL COLLISION: Joint_5 -> Saele",
    "robotStateSnapshot": "[...]",
    "eventDataJson": "{\"robotLink\":\"Joint_5\",\"collisionObject\":\"Saele\",\"colli-
    → sionPoint\":{\"x\":-1.146544337272644,\"y\":0.038220398128032687,\"z\":1.14
    → 65449333190919},\"distance\":1.6219075918197632}"
  },
  {
    "monitorName": "Collision Detector",
    "description": "Collision detected between Gripper and CylinderHead (8)",
    "eventType": 1,
    "robotStateSnapshot": "[...]",
    "eventDataJson": "{\"robotLink\":\"Gripper\",\"collisionObject\":\"CylinderHead
    → (8)\",\"collisionPoint\":{\"x\":-1.9333001375198365,\"y\":0.5249927043914795,\"
    → z\":-1.1674463748931885},\"distance\":2.3186628818511965}"
  }
]
```

Abbildung 15: JSON-Log zur Kollisionserkennung. Sich wiederholende Key-Value Paare wurden verkürzt

Weiterführend ist zu erkennen, dass die Kollision mit verschiedenen Gliedern des Roboters sequentiell erkannt wird. Sobald der Roboter sich visuell weiter in die Säule bewegt, wird jeweils bei der Kollision mit einem weiteren Glied eine weitere Kollision erkannt und ein eigenes Event getriggert. Die Reihenfolge der Joints in diesem Szenario ist nach eingehender Überprüfung als korrekt zu bewerten, da die Robotergeometrie dafür sorgt, dass Gelenk 4 deutlich breiter

ist als 5 und 6. Gelenk 5 und 6 sind in die Geometrie von Gelenk 4 eingefasst, daher kollidiert der Roboter initial mit Gelenk 4, bevor eine Kollision an den kinematisch dahinterliegenden Gelenken erkannt wird.

Weiterführend lässt sich beim Greifen des Werkstücks feststellen, dass hier ebenfalls eine Kollision erkannt wird: Durch das Greifen des Werkstücks wird eine falsch-positive Kollision getriggert, da bevor das Werkstück gegriffen wird und semantisch in die kinematische Kette des Roboters verschoben wird, für einen kurzen Zeitpunkt eine Kollision stattfindet. Ein Beispiel dazu findet sich im letzten Block von Abbildung 15. Gleiches lässt sich beim Ablegen des Werkstücks beobachten. Wichtig zu erwähnen ist der unterschiedliche EventType, da das Werkstück aufgrund von fehlendem Tag nicht als kritisch eingestuft wird.

## 4.4 Auswertung Singularity Detection Monitor

### 4.4.1 Künstliche Singularitätserzeugung

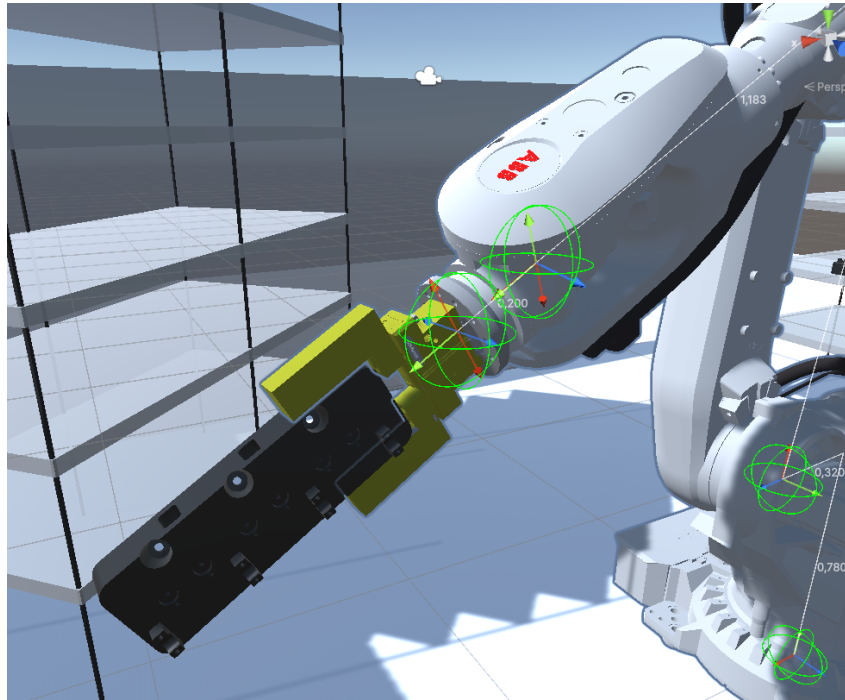


Abbildung 16: Pose in Unity, bei der eine Wrist-Singularität mit ( $\theta_5 \approx 0^\circ$ ) entsteht

Zur Untersuchung der Singularitätserkennung wurde in RobotStudio ein Szenario erstellt, das gezielt eine Wrist-Singularität provoziert. Dabei wurde der Roboter in eine Pose geführt, in der die Achsen 4 und 6 nahezu kollinear verlaufen und somit die Bedingung  $\theta_5 \approx 0^\circ$  erfüllt ist. Die Pose ist in Abbildung fig:wristSingularity zu erkennen, hier befindet sich der Roboter nach dem Greifen des Werkstücks auf dem Weg zum rechten Regal, muss sich für das Platzieren des Werkstücks im Regal jedoch umorientieren. So kommt die Singularität zustande.

### 4.4.2 Simulationsergebnis

```
[
  {
    "monitorName": "Singularity Detector",
    "eventType": 2,
    "description": "Entering Wrist Singularity (\u03B85 \u2248 0\u00B0)",
    "robotStateSnapshot": "[...]",
    "eventDataJson": {
      "singularityType": "Wrist Singularity (theta_5 approx 0 deg)",
      "jointAngles": [
        -82.7,
        -6.9,
        36.6,
        -112.8,
        -4.9,
        -153.5
      ],
      "wristThreshold": 5.0,
      "manipulability": 0.191,
      "isEntering": true,
      "[...]": "[...]"
    }
  },
  {
    "monitorName": "Singularity Detector",
    "eventType": 1,
    "description": "Exiting Wrist Singularity (\u03B85 \u2248 0\u00B0)",
    "robotStateSnapshot": "[...]",
    "eventDataJson": "[...]"
  }
]
```

Abbildung 17: Gekürzter Auszug der in Unity aufgezeichneten Safety Events zur Wrist-Singularität

Die vom Monitor in Unity aufgezeichneten Safety Events sind in Abbildung 17 als gekürzter Auszug dargestellt. Es werden sowohl das „Entering“- als auch das „Exiting“-Ereignis erfasst, jeweils mit den zugehörigen Gelenkwinkeln und einem berechneten Manipulierbarkeitswert. Nicht relevante Felder des Snapshots wurden entfernt, da die Gelenkwinkel bereits im eventDataJson enthalten sind. Zusätzliche Informationen wurden mit "[...]" abgekürzt.

Die Analyse der Ereignisse zeigt, dass der Monitor den Eintritt in die Wrist-Singularität bei einer Gelenkkonfiguration von etwa  $[-82.7, -6.9, 36.6, -112.8, -4.9, -153.5]^\circ$  registrierte. Der berechnete Manipulierbarkeitswert lag hier bei  $w \approx 0.19$ . Beim Verlassen der Pose  $[-91.8, -8.3, 37.8, -91.3, 5.2, -182.6]^\circ$  wurde das „Exiting“-Ereignis ausgegeben, wobei der Manipulierbarkeitswert mit  $w \approx 0.20$  ebenfalls sehr niedrig blieb. Die Ereignisse decken sich mit dem in RobotStudio provozierten Szenario und markieren den Übergang in und aus einer singulären Konfiguration. Zur Detektion von Singularitäten anderen Typs (hier Schulter- und

Ellbogensingularitäten) wird das gleiche Verfahren angewendet. Hier decken sich die Ergebnisse mit den oben Beschriebenen.

## 4.5 Auswertung Joint Dynamics Monitor

### 4.5.1 Überhöhung der Gelenkgeschwindigkeiten

Der *Joint Dynamics Monitor* erfasst kontinuierlich die Dynamik der sechs Roboterachsen. Die Implementierung in Unity basiert auf den vom digitalen Zwilling gestreamten Gelenkwinkeln, aus denen Geschwindigkeiten und Beschleunigungen differenziell berechnet werden. Zur Signalanalyse werden mehrere Mechanismen kombiniert, darunter exponentielle Glättung sowie Fenster-basiertes Mittel zur Dämpfung von Ausreißern. Zusätzlich wird ein Sicherheitsfaktor von 0,8 auf die in RobotStudio spezifizierten Maximalwerte angewendet, sodass die Schwellwerte im Monitor niedriger liegen als die realen physikalischen Limits (vgl. Implementierung in `JointDynamicsMonitor.cs`). Hier wird mittels des Befehl `v7000` im RAPID-Code in RobotStudio die Geschwindigkeit auf den programmierbaren Maximalwert gesetzt. Dies wurde an zwei Stellen der Roboterbewegung umgesetzt: Bei der Bewegung von der Home-Position zum linken Regal und bei der Bewegung nach dem Greifen des Werkstücks zur Maschine. Als primäre Drehachse ist hier eine Überhöhung der Geschwindigkeit von Gelenk 1 (zentrale horizontale Drehachse des Roboters) zu erwarten.

### 4.5.2 Simulationsergebnis

Während der Testsimulation wurden durch den Monitor zwei Überschreitungen der Geschwindigkeitsgrenzen auf **Joint 1** registriert. Diese Ereignisse sind in der aus Unity aufgezeichneten Logdatei dokumentiert. Ein Auszug ist in Abbildung 18 dargestellt. Dort wird jeweils die Überschreitung der Geschwindigkeit und das nachfolgende „resolved“-Ereignis vermerkt, zusammen mit dem aktuellen Gelenkwinkelzustand des Roboters.

```
[
  {
    "monitorName": "Joint Dynamics Monitor",
    "eventType": 1,
    "description": "Joint 1 velocity limit exceeded: 53.95°/s",
    "robotStateSnapshot": {
      "jointAngles": [
        -2.12,
        -4.52,
        31.67,
        196.26,
        -57.86,
        -74.34
      ],
      "[...]": "[...]"
    }
  },
  {
    "monitorName": "Joint Dynamics Monitor",
    "eventType": 1,
    "description": "Joint 1 velocity limit resolved: 48.63°/s",
    "robotStateSnapshot": {
      "jointAngles": "[...]",
      "[...]": "[...]"
    }
  }
]
```

Abbildung 18: Gekürzter Auszug der in Unity aufgezeichneten Safety Events des Joint Dynamics Monitor, Ereignis wiederholt sich

Zum Vergleich wurden die Gelenkwinkel aus RobotStudio exportiert und anhand der berechneten Achsgeschwindigkeiten ausgewertet. Abbildung 19 zeigt die Ergebnisse: Im oberen Diagramm sind die Achswinkel aller sechs Gelenke dargestellt (J1 hervorgehoben in rot), darunter die berechneten Geschwindigkeiten mit markierten Schwellwertbereichen. Die rot eingefärbten Abschnitte kennzeichnen Intervalle, in denen die aus RobotStudio exportierten Daten die definierte Grenze von  $\pm 50^\circ/\text{s}$  überschreiten. Die blau markierten Bereiche repräsentieren die durch den Monitor in Unity ausgegebenen Safety Events. Zu erkennen ist eine starke Überschneidung der Bereiche. Weiterführend zeichnet sich vor allem in der zweiten Geschwindigkeitsüberschreitung eine Verzögerung beim Ein- und Austritt in den kritischen Zustand ab. Die Events in Unity werden hier zeitverzögert weitergegeben.



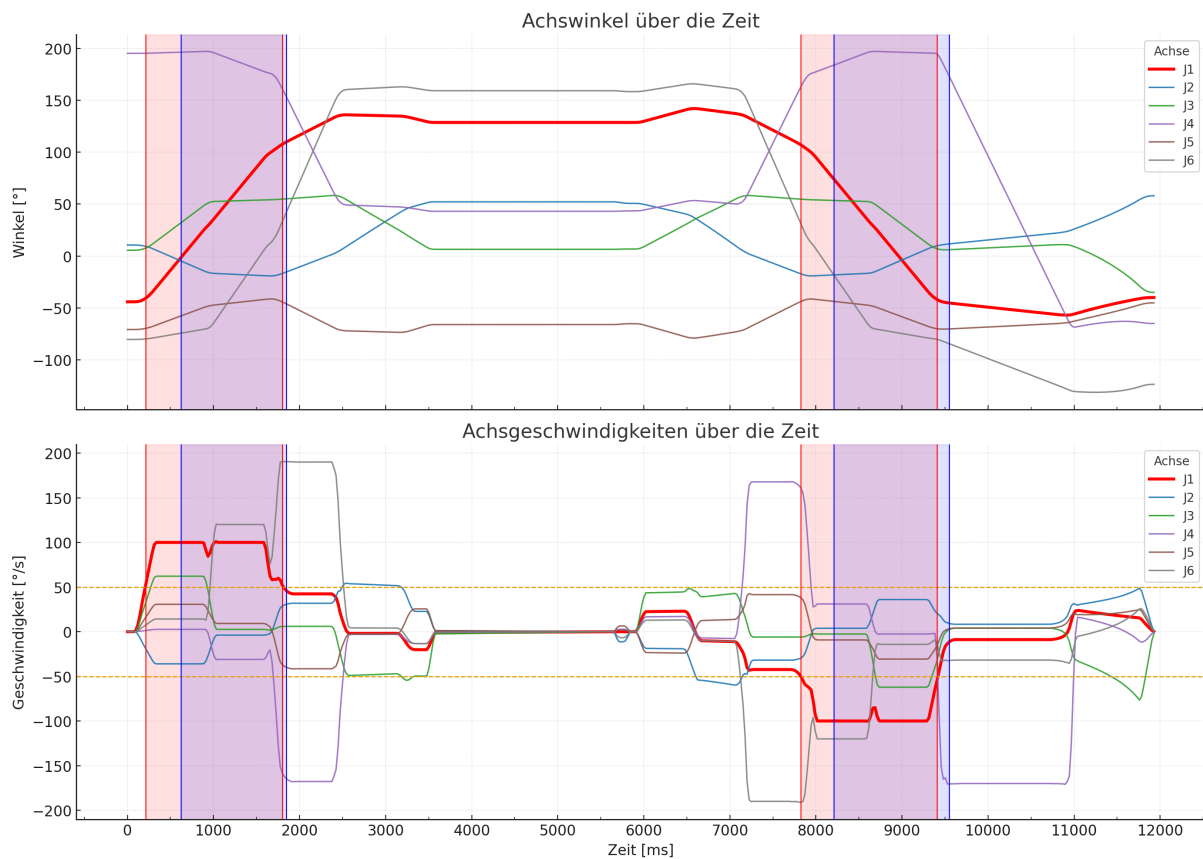


Abbildung 19: Achswinkel (oben) und Achsgeschwindigkeiten (unten) mit markierten Bereichen (rot: Schwellenübertritte in den Rohdaten, blau: Safety Events aus Unity).

Eine Gegenüberstellung der Zeitintervalle ist in Tabelle 4 enthalten. Dort werden die Start- und Endzeitpunkte der Abschnitte, die Dauer sowie die Gelenkwinkel und -geschwindigkeiten an diesen Punkten angegeben. Anhand dieser Darstellung wird sichtbar, dass die blau markierten Safety Events zeitlich nach den rot markierten Schwellenübertritten liegen. Die in Tabelle 4 und Abbildung 19 dargestellten Intervalle wurden ermittelt, indem die aus RobotStudio exportierten Gelenkwinkel mit den in den Safety Events gespeicherten Zuständen vergleichbar sind. Dazu wurde der euklidische Abstand zwischen den Vektoren der Achswinkel berechnet, um den jeweils nächstliegenden Zeitpunkt in den Referenzdaten zu bestimmen. Auf diese Weise lassen sich die vom Monitor in Unity gemeldeten Ereignisse mit den in den Rohdaten beobachteten Schwellenübertritten korrelieren.

Typ	Start [ms]	Ende [ms]	Dauer [ms]	Start J1 [°]	Ende J1 [°]
Dynamik (rot)	216	1800	1584	-40.35	107.62
Zielwinkel (blau)	624	1848	1224	-1.25	109.89
Dynamik (rot)	7824	9408	1584	107.08	-41.88
Zielwinkel (blau)	8208	9552	1344	74.15	-45.24

Tabelle 4: Zeitintervalle und Zustände der Joint-Dynamics-Auswertung

## 4.6 Validierung durch Experteninterview

Im Rahmen der Ergebnisdarstellung wurde ein Experteninterview mit Daniel Syniawa (M.Sc.) durchgeführt. Ziel war es, die Funktionsfähigkeit des Frameworks praxisnah zu validieren und Einschätzungen zur industriellen Einordnung zu gewinnen. Das Interview diente ausschließlich der *deskriptiven* Ergänzung der Ergebnisse; eine vertiefte Interpretation erfolgt in Kapitel 5.

### 4.6.1 Vorgehen und Gegenstand

Im Interview wurden die Software und ihre Architektur vorgestellt. Anschließend wurden die in Kapitel 4 beschriebenen Testfälle gemeinsam in *RobotStudio* nachvollzogen: Der Roboter führte die Szenarien aus, während parallel beobachtet wurde, ob und wie die Module Ereignisse auslösen und ob diese im Logging erfasst werden. Damit wurde die grundsätzliche Funktionsweise des Frameworks demonstriert (Auslösen von Safety-Events, Erzeugung von JSON-Logs mit Programmkontext).

### 4.6.2 Beobachtungen

Prinzipiell konnte ein schnelles Verständnis für die Anwendung des Frameworks und dessen Funktionsweise gewonnen werden. Bei der Testung der Fälle wurde der praktische Nutzen der abgespeicherten JSON-Logs hervorgehoben: Für jedes Ereignis liegen der aktuelle Programmzeiger, das aktuell ausgeführte Programm sowie die relevante Roboterpose bzw. Kontextinformationen vor. Dies erleichtert die Fehlersuche und macht die Analyse auch für weniger erfahrene Anwender nachvollziehbar.

### 4.6.3 Hinweis zur Evaluationsmethodik

Für eine belastbare Evaluation schlug Syniawa vor, die Leistung des Frameworks *quantitativ* gegen manuelle Verfahren zu vergleichen. Konkret: Wie schnell findet eine Person den Fehler (z. B. Auftreten einer Singularität) im RAPID-Code in *RobotStudio* im Vergleich zur Erkennung durch Ausführung und Logging im Framework? Ein solcher Vergleich wäre mit erheblichem Aufwand verbunden, würde aber die Einordnung der Wirksamkeit deutlich schärfen.

### 4.6.4 Einordnung des Robotersimulations-Ökosystems

Syniawa wies darauf hin, dass die aktuelle Werkzeuglandschaft stark proprietär geprägt ist und es nur wenige plattformübergreifende Lösungen mit integrierter Physik gibt. Häufig ist bereits die stabile Anbindung eines Roboters herausfordernd; Simulation in *RobotStudio* erfordert viel Expertenwissen und Zeit. Die *Robot Web Services* (RWS) seien komplex und eher knapp dokumentiert; insgesamt sei die Nutzerbasis in diesem Bereich klein. Diese Beobachtungen unterstreichen die Relevanz eines modularen, erweiterbaren Ansatzes wie in dieser Arbeit umgesetzt.

### 4.6.5 Zusammenfassung

Das Interview bestätigte die grundsätzliche Funktionalität des Frameworks in den demonstrierten Testfällen und identifizierte einen klaren Pfad für eine zukünftige, quantitative Evaluation. Zudem wurde der praktische Mehrwert der strukturierten Logs (Programmzeiger, laufendes Programm, Pose/Kontext) betont. Die Einordnung der industriellen Robotersimulationslandschaft liefert den Rahmen, in dem die vorgestellten Ergebnisse zu sehen sind.

### 4.7 Zusammenfassung der Ergebnisse

Die Ergebnisse zeigen, dass die in Unity implementierten Monitore in allen Testfällen die in RobotStudio provozierten Szenarien widerspiegeln konnten. Dabei wurde deutlich, dass sich für jedes Modul charakteristische Muster im Logging abzeichnen: Prozessabweichungen wurden sequenziell dokumentiert, Kollisionen mit Schweregraden versehen, Singularitäten mit Gelenkwinkeln und Manipulierbarkeitswerten erfasst, und Geschwindigkeitsverletzungen durch Event-Paare (exceeded/resolved) gekennzeichnet.

Monitor	Getestetes Szenario	Erkannte Ereignisse / Beobachtungen
Process Flow	Ablauf mit bewusst fehlerhafter Reihenfolge der Operationen	Abweichung von der erwarteten Sequenz korrekt erkannt, Events dokumentieren Verletzung der Prozessfolge.
Collision Detection	Simulation mit Kollision zwischen Greifer und Werkstück bzw. Störkörper	Mehrere Kollisionen aufgezeichnet, inklusive beteiligter Objekte; Events mit Schweregrad (critical/warning) unterschieden.
Singularity Detection	Pose in RobotStudio, die eine Wrist-Singularität ( $\theta_5 \approx 0^\circ$ ) provoziert	ENTERING- und EXITING-Events erfasst; Gelenkwinkel und Manipulierbarkeitswerte im JSON protokolliert; Szenario deckt sich mit RobotStudio.
Joint Dynamics	Bewegung mit Überschreitung der Geschwindigkeitsgrenzen auf J1 ( $\pm 50^\circ/\text{s}$ )	Zwei Event-Paare (exceeded/resolved) aufgezeichnet; Verzögerung zwischen Schwellenübertritt (Rohdaten) und Event (Monitor) sichtbar.

Tabelle 5: Übersicht der getesteten Monitore, Szenarien und erkannten Ereignisse im Ergebnisteil

Die Übersicht in Tabelle 5 verdeutlicht die Unterschiede zwischen den Monitoren hinsichtlich Art der Szenarien und Form der erfassten Events. Auffällig ist, dass sich in einigen Fällen eine zeitliche Verzögerung zwischen den in den Rohdaten beobachteten Zuständen und den vom Monitor generierten Ereignissen zeigt. Diese Beobachtung ergibt sich aus den in Kapitel 3

beschriebenen Mechanismen (z. B. Glättung, Abtastrate).

Im nächsten Abschnitt wird ein Experteninterview herangezogen, um die hier dargestellten Ergebnisse einzuordnen und im Hinblick auf ihre Relevanz für praktische Anwendungen zu reflektieren.

## 5 Diskussion

### 5.1 Einleitung

In diesem Kapitel werden die in Kapitel 4 präsentierten Ergebnisse kritisch reflektiert und in den fachlichen Kontext eingeordnet. Im Zentrum stehen die Gesamtarchitektur des Frameworks und die vier implementierten Safetymodule. Ergänzend wird ein Experteninterview mit Daniel Syniawa herangezogen, in dem die Software, die Architektur sowie konkrete Testcases in RobotStudio gemeinsam betrachtet wurden. Ziel ist es, die Stärken und Grenzen der Arbeit transparent zu machen und konkrete Ansatzpunkte für zukünftige Arbeiten zu identifizieren.

### 5.2 Diskussion des Frameworks

Die Architektur hat sich als tragfähige Grundlage erwiesen: Durch den adapterbasierten Zugriff auf die Roboterschnittstelle (z. B. ABB Robot Web Services) und eine konsequent *event-getriebene* Struktur konnten Safetymodule unabhängig voneinander entwickelt und über ein gemeinsames Interface in das *RobotSystem* integriert werden. Das Event-System (Observer-Pattern) entkoppelt Erkennung und Verarbeitung; die JSON-basierte Persistenz vereinfacht Nachvollziehbarkeit und Weiterverarbeitung. Positiv wirkte sich zudem die Nutzung der Denavit–Hartenberg-Parameter aus, die eine konsistente kinematische Modellierung verschiedener Robotermodelle erlaubt.

Gleichzeitig bleiben offen, inwiefern einzelne Simulationsparameter oder Modellierungsentscheidungen in Unity bei hochkomplexen Geometrien eine Limitation darstellen *könnten*. Die Unity-Engine bietet hier zwar eine Grundlage physischer Modellierung, abschliessende Aussagen zur Genauigkeit in sehr dynamischen oder kleinschrittigen Prozessen können aber nicht getroffen werden. Diese Aspekte wurden nicht systematisch untersucht und markieren Raum für künftige Studien.

Das Experteninterview mit Daniel Syniawa ordnet die Arbeit in die Praxis ein: Er bestätigte, dass es insgesamt nur wenige plattformübergreifende Werkzeuge mit integrierter Physik gibt und dass die Tool-Landschaft stark proprietär geprägt ist. Nach seiner Erfahrung ist bereits die stabile Inbetriebnahme und Anbindung von Robotern für viele Firmen aufwendig; physikalische Simulation mit einer Modellierung des Arbeitsraumes in RobotStudio sowie die Auswertung der hier untersuchten Parameter ist nur in begrenztem Umfang möglich und benötigt beträchtliches Expertenwissen und Zeit. Syniawa wies außerdem darauf hin, dass Services proprietär Hersteller wie die RobotStudio-API (Robot Web Services) nicht trivial in der Anwendung sind, oft schlecht

dokumentiert und auf eine insgesamt kleine Nutzerbasis trifft. Auch das lässt sich im Rahmen der Entwicklung dieses Frameworks bestätigen.

Als hilfreich lässt sich ausserdem der Output des JSON-basierten Loggings werten: Syniawa spricht hier vor allem von den mit einem SafetyEvent zusammen herausgegebenen Metadaten zum aktuellen Stand des Programmzeigers und der aktuellen ausgeführten Routine. Dies stellt einen vielversprechenden Ansatz zum Debugging im Roboterprogrammcode dar, welcher manuell mit deutlich mehr Aufwand verbunden wäre. Hier gilt es weiter zu evaluieren, inwiefern sich dies quantitativ beschreiben lässt, so Syniawa.

## **5.3 Diskussion der Safetymodule**

### **5.3.1 Prozessfolgenüberwachung**

Das Modul erkennt Abweichungen in der vorgesehenen Reihenfolge zuverlässig, sofern Stationen korrekt modelliert und ausgelöst werden. Nicht abgedeckt sind Konstellationen, in denen ein Werkstück *zwischen* zwei Stationen unbeabsichtigt abgelegt oder verloren wird, ohne dass eine Station detektiert wird. Die Praxistauglichkeit hängt damit von der Qualität der Prozessmodellierung und der Art des Prozesses ab. Im aktuellen Fokus stand hier ein sequentieller Prozess, die Literatur beschreibt hier im Kontext industrieller Fertigung jedoch mehrere Prozessarten und theoretische Modellierungsansätze. Im Interview wurde die grundsätzliche Relevanz dieses Moduls bestätigt; zugleich wird deutlich, dass die industrielle Praxis oft komplexere Ablaufmodelle erfordert.

### **5.3.2 Kollisionserkennung**

In der Simulation wurden die vorgesehenen Kollisionsfälle erkannt; zugleich traten Fehlalarme auf, insbesondere beim Greifen und Loslassen von Werkstücken. Wie stark Modellierungsdetails oder Simulationsparameter das Verhalten in Grenzfällen beeinflussen, wurde in dieser Arbeit nicht systematisch untersucht und ist als potenzielle Limitation zu betrachten. Weiterführend ist die Genauigkeit der Modellierung der Meshes des Roboters und der Umgebung hier essentiell: Unity modelliert konvexe Meshes, welche die räumlichen Grenzen darstellen mit maximal 255 Kanten. Daher kann es passieren, dass die tatsächliche Topologie des Robotermodells vom Kollisionskörper abweicht. Insgesamt liefert das Modul einen belastbaren Proof-of-Concept, dessen Generalisierung im Rahmen weiterführender Evaluierungen zu prüfen ist.

### **5.3.3 Achsgeschwindigkeiten und -beschleunigungen**

Grenzwertverletzungen wurden zuverlässig detektiert. In den Ergebnissen zeigte sich allerdings ein zeitlicher Versatz zwischen den in RobotStudio vorliegenden Referenzdaten und der Detektion in Unity: Überschreitungen wurden etwas später erkannt und endeten geringfügig später.

Dieser Versatz ist plausibel auf Aktualisierungsrate und eingesetzten Glättungsalgorithmus zurückzuführen, dessen Parameter konfigurierbar sind. Ob dadurch ein Versatz mit den in der Praxis vom Roboter gefahrenen Achsgeschwindigkeiten und -beschleunigungen entsteht, lässt sich hier nicht abschliessend bewerten.

### 5.3.4 Singularitätserkennung

Die gewählte, winkelbasierte Heuristik funktionierte für den betrachteten Roboter, ist jedoch nicht universell. Alternativ bieten sich Kennzahlen an, die näher an der Kinematik operieren, etwa das Manipulability-Maß (Yoshikawa) oder der kleinste Singulärwert der Jacobi-Matrix als Abstandsmaß zur Singularität. Eine generische, Jacobian-basierte Methode zur Erkennung von Freiheitsgradverlusten wurde implementiert, im Rahmen der vorliegenden Tests jedoch nicht eingesetzt; eine Erweiterung auf andere Roboter wäre möglich, wurde aber nicht vorgenommen.

Im Interview formulierte Syniawa einen pragmatischen Maßstab für die Evaluation: Für eine belastbare Beurteilung wäre ein direkter Vergleich mit manuellem Debugging in RobotStudio sinnvoll, also ein Messaufbau, der die Zeit bis zur Fehlerlokalisierung im RAPID-Code der Zeit gegenüberstellt, die das Framework über Ausführung und Logging benötigt. Zugleich hob er hervor, dass die automatische Bereitstellung von Programmzeiger, aktueller Pose und Kontext im Event-Log eine erhebliche Arbeitserleichterung darstellt und die Analyse prinzipiell auch weniger erfahrenen Anwendern ermöglicht.

## 5.4 LLM-gestützte Rückkopplung auf Basis der Safety-Events

Die Ergebnisse zeigen, dass das Framework Fehlerzustände konsistent und kontextreich protokolliert: Für jedes Ereignis liegen Aktuelle Bewegungs- und Programmdateien, beispielsweise das aktuelle Modul, Routine, Programmzeile sowie Motordaten und Achswinkel vor. Hinzu kommen event-spezifische Felder im `eventDataJson`, etwa Kollisionspunkt und Distanz oder Gelenkwinkel und Manipulierbarkeitswert bei Singularitäten<sup>57</sup>. Diese maschinenlesbare Struktur eignet sich unmittelbar als gezielter Eingabekontext für generative Modelle: Der Fehler wird präzise beschrieben, der relevante Zustandsausschnitt ist enthalten, und die Semantik stammt aus den domänenspezifischen Monitoren. Das bietet eine gute Basis, um Code- oder Pfadänderungen vorzuschlagen und anschließend identisch zu verifizieren.

Operativ lässt sich darauf ein kurzer Iterationszyklus aufbauen: Aus einem fehlgeschlagenen Lauf durch Nutzer- oder LLM-generierten Roboterprogrammcode wird ein kompakter Fehlerkontext aus gebildet und an ein LLM übergeben: Das LLM liefert einen minimalen Änderungsvorschlag am Robotercode bzw. an der Pfaddefinition und durch die Re-Simulation kann eine erneute Prüfung des Codes stattfinden. Die Akzeptanzkriterien für eine erfolgreiche Behebung des Fehlers

<sup>57</sup>Vgl. u. a. Process-Flow-Event inkl. Programmpointer und RobotStateSnapshot sowie die Beschreibung des JSON-Aufbaus; Kollisionsevents mit `collisionPoint` und Distanz; Singularitätsevents mit Gelenkwinkeln und Manipulierbarkeitswert; Joint-Dynamics-Events als „exceeded“/„resolved“.

beispielsweise mit dem minimalen Achswinkel bereits im Rahmen der Eventdaten und Monitorlogik im System vor wodurch die Bewertung reproduzierbar bleibt. Damit kann das Framework als Schnittstelle zwischen realitätsnaher Ausführung und LLM-gestützter Verbesserung dienen. Praktisch sind drei Punkte entscheidend und aus den Ergebnissen ableitbar: erstens ein schlanker, stabiler Prompt-Ausschnitt aus genau den Feldern, die im Logging ohnehin verfügbar sind (z. B. Programmpointer, Gelenkwinkel, Kollisionspunkt), zweitens feste Akzeptanzregeln in der Simulation, drittens Versionierung und Wiederholbarkeit der Läufe. Auf dieser Basis lässt sich in zukünftiger Arbeit untersuchen, in welchem Umfang sich Fehlerquote und Nachbearbeitungszeit durch die Rückkopplung tatsächlich reduzieren – die in den Ergebnissen sichtbaren Muster wie liefern bereits geeignete Zielgrößen.

## **5.5 Reflexion des eigenen Vorgehens**

Die Entwicklung folgte bewusst einem iterativen Vorgehen: von einem kleinen Testfall hin zu einer breiteren Abdeckung, mit Zwischenstufen des Refactorings. Dieses Vorgehen erwies sich als geeignet, um Architekturentscheidungen (Adapter/Observer) empirisch zu validieren. Rückblickend entstanden stellenweise Komponenten, die für den unmittelbaren Use-Case komplexer waren als nötig; gleichwohl war die iterativ-explorative Herangehensweise im Kontext einer Framework-Entwicklung zweckmäßig und hat zur jetzigen Struktur geführt.

## **5.6 Grenzen und Generalisierbarkeit**

Die vorliegende Arbeit wurde in der Simulation evaluiert; Echtzeitverhalten, Sensitivität gegenüber Simulationsparametern sowie Übertragbarkeit auf weitere Roboter wurden nicht systematisch untersucht. Darüber hinaus beschränkt sich die Adaptervalidierung auf ABB Robot Web Services. Das Interview verdeutlicht, dass proprietäre Ökosysteme, komplexe Schnittstellen und eine kleine Nutzerbasis zusätzliche Hürden für Verallgemeinerung und Transfer darstellen. Diese Punkte markieren bewusste Grenzen des aktuellen Stands und leiten unmittelbar zu Folgestudien über.



## 6 Fazit und Ausblick

### 6.1 Zusammenfassung

Das entwickelte Framework demonstriert die Machbarkeit eines modularen, eventgetriebenen Ansatzes für sicherheitsrelevante Überwachungsaufgaben in der Roboterprogrammierung. Die Module erfüllten ihre Kernfunktionen, zugleich wurden klare Erweiterungspfade sichtbar. Das Experteninterview mit Daniel Syniawa bestätigte Relevanz und Nutzen des Ansatzes und unterstrich die praktischen Hürden einer stark proprietären Werkzeuglandschaft. Insgesamt stellt die Arbeit einen belastbaren Proof-of-Concept dar, der durch quantitative Evaluationen und architektonische Erweiterungen in Richtung eines praxistauglichen Systems weitergeführt werden kann.

### 6.2 Ausblick

Aus den Diskussionen ergeben sich zwei unmittelbare Schienen: Erstens eine *quantitative* Evaluation, die die Leistungsfähigkeit des Frameworks systematisch gegen manuelle Verfahren stellt. Mögliche Metriken umfassen etwa Erkennungszeit, Zuverlässigkeit (Recall) der Detektion und Fehlalarmrate; für die Dynamiküberwachung sind zudem akzeptable Toleranzbänder zu definieren und transparent zu begründen. Zweitens eine *architektonische* Weiterentwicklung: engere Kopplung an Controllerschnittstellen, Erweiterung auf weitere Herstelleradapter sowie die Prüfung echtzeitnaher Ausführungsumgebungen.

Langfristig eröffnet die strukturierte JSON-Ausgabe eine Brücke zu automatisierten Analyse- und Korrekturprozessen. Eine naheliegende Linie ist die Einbindung von Large Language Models (LLMs), die auf Basis von Logdaten und Layoutinformationen Roboterprogramme analysieren und Verbesserungsvorschläge generieren. In einem nächsten Schritt ließe sich dies als MCP-Server konzipieren, der Rückmeldungen zyklisch in das Framework einspeist und damit den Bedarf an tiefem Expertenwissen reduziert, ohne den Sicherheitsanspruch zu unterlaufen.

## Literatur

- ABB (2025). *Robot Web Services API Documentation*. ABB. URL: <https://developercenter.robotstudio.com/api/RWS> (besucht am 20.08.2025).
- ABB Robotics (2025). *Produktspezifikation IRB 6700 on OmniCore*. Product Specification 3HAC080365-003. Revision K, Workspace Main version a641, checked in 2025-01-29. S-721 68 Västerås, Sweden: ABB AB, Robotics & Discrete Automation. URL: <https://library.e.abb.com/public/b62cda447b7343778b71830c0fa8123b/3HAC080365%20PS%20IRB%206700%20on%20OmniCore-de.pdf?x-sign=rCJlPoYfTIIfTA+9EOP5NBvCj7jHVI9lxvc0VnEIvzhx9TbrSb6BeNgdE8vZufZdM>.
- Andaluz, VH, FA Chicaiza, C Gallardo, WX Quevedo, J Varela, JS Sá nchez und O Arteaga (2016). „Unity3D-MatLab Simulator in Real Time for Robotics Applications“. In: *Augmented Reality, Virtual Reality, and Computer Graphics*. Hrsg. von LT De Paolis und A Mongelli. Cham: Springer International Publishing, S. 246–263. ISBN: 978-3-319-40621-3.
- Anthropic (2024). *Introducing the Model Context Protocol*. <https://www.anthropic.com/news/model-context-protocol>. Accessed: 2025-09-06.
- Audonnet, FP, A Hamilton und G Aragon-Camarasa (2022). „A Systematic Comparison of Simulation Software for Robotic Arm Manipulation using ROS2“. In: *2022 22nd International Conference on Control, Automation and Systems (ICCAS)*, S. 755–762. URL: <https://api.semanticscholar.org/CorpusID:248157288>.
- Bartneck, C, M Soucy, K Fleuret und EB Sandoval (2015). „The robot engine – Making the unity 3D game engine work for HRI“. In: *2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, S. 431–437. DOI: 10.1109/ROMAN.2015.7333561.
- Bhat, V, AU Kaypak, P Krishnamurthy, R Karri und F Khorrami (2024). *Grounding LLMs For Robot Task Planning Using Closed-loop State Feedback*. arXiv: 2402.08546 [cs.R0]. URL: <https://arxiv.org/abs/2402.08546>.
- Biggs, G und B MacDonald (2003). *A Survey of Robot Programming Systems*. Techn. Ber. Survey of manual (text-/grafikbasiert) und automatic (PbD, Learning) robot programming systems. Auckland, New Zealand: Department of Electrical & Electronic Engineering, University of Auckland.
- Bilancia, P, J Schmidt, R Raffaeli, M Peruzzini und M Pellicciari (2023). „An Overview of Industrial Robots Control and Programming Approaches“. In: *Applied Sciences* 13.4. ISSN: 2076-3417. DOI: 10.3390/app13042582. URL: <https://www.mdpi.com/2076-3417/13/4/2582>.
- Bonci, A, A Di Biase, MC Giannini und S Longhi (2023). „Robot Operating System 2 (ROS2)-Based Frameworks for Increasing Robot Autonomy: A Survey“. In: *Applied Sciences* 13.24.

- DOI: 10.3390/app132412796. URL: <https://iris.univpm.it/retrieve/04e7a132-fd87-4980-941d-33e028b34a77/applsci-13-12796%2B.pdf>.
- Brohan, A, N Brown, J Carbajal, Y Chebotar, X Chen, K Choromanski, T Ding, D Driess, A Dubey, C Finn, P Florence, C Fu, MG Arenas, K Gopalakrishnan, K Han, K Hausman, A Herzog, J Hsu, B Ichter, A Irpan, N Joshi, R Julian, D Kalashnikov, Y Kuang, I Leal, L Lee, TWE Lee, S Levine, Y Lu, H Michalewski, I Mordatch, K Pertsch, K Rao, K Reymann, M Ryoo, G Salazar, P Sanketi, P Sermanet, J Singh, A Singh, R Soricut, H Tran, V Vanhoucke, Q Vuong, A Wahid, S Welker, P Wohlhart, J Wu, F Xia, T Xiao, P Xu, S Xu, T Yu und B Zitkovich (2023). *RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control*. arXiv: 2307.15818 [cs.R0]. URL: <https://arxiv.org/abs/2307.15818>.
- Cassandras, CG und S Lafortune (2021). *Introduction to Discrete Event Systems*. 3. Aufl. Cham: Springer, S. XXVI, 804. ISBN: 978-3-030-72272-2. DOI: 10.1007/978-3-030-72274-6. URL: <https://doi.org/10.1007/978-3-030-72274-6>.
- Cohen, V, JX Liu, R Mooney, S Tellex und D Watkins (2024). *A Survey of Robotic Language Grounding: Tradeoffs between Symbols and Embeddings*. arXiv: 2405.13245 [cs.R0]. URL: <https://arxiv.org/abs/2405.13245>.
- Corke, PI (2007). „A Simple and Systematic Approach to Assigning Denavit–Hartenberg Parameters“. In: *IEEE Transactions on Robotics* 23.3, S. 590–594. DOI: 10.1109/TR0.2007.896765.
- Driess, D, F Xia, MSM Sajjadi, C Lynch, A Chowdhery, B Ichter, A Wahid, J Tompson, Q Vuong, T Yu, W Huang, Y Chebotar, P Sermanet, D Duckworth, S Levine, V Vanhoucke, K Hausman, M Toussaint, K Greff, A Zeng, I Mordatch und P Florence (2023). *PaLM-E: An Embodied Multimodal Language Model*. arXiv: 2303.03378 [cs.LG]. URL: <https://arxiv.org/abs/2303.03378>.
- Ericson, C (2004). *Real-Time Collision Detection*. 1. Aufl. CRC Press. ISBN: 978-1558607323.
- Fickinger, A, A Bendahi und S Russell (2025). *Provable Sim-to-Real Transfer via Offline Domain Randomization*. arXiv: 2506.10133 [cs.LG]. URL: <https://arxiv.org/abs/2506.10133>.
- Gamma, E, R Helm, R Johnson und J Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- Gu, X, M Chen, Y Lin, Y Hu, H Zhang, C Wan, Z Wei, Y Xu und J Wang (2025). „On the Effectiveness of Large Language Models in Domain-Specific Code Generation“. In: *ACM Transactions on Software Engineering and Methodology* 34.3, S. 1–22. ISSN: 1557-7392. DOI: 10.1145/3697012. URL: <http://dx.doi.org/10.1145/3697012>.
- Hesse, S (2011). *Greifertechnik*. München: Carl Hanser Verlag GmbH & Co. KG. DOI: 10.3139/9783446427419. eprint: <https://www.hanser-elibrary.com/doi/pdf/10.3139/9783446427419>. URL: <https://www.hanser-elibrary.com/doi/abs/10.3139/9783446427419>.
- Hohpe, G (2006). *Programming Without a Call Stack – Event-driven Architectures*. Available at <http://www.eaipatterns.com>.

- Holubek, R, DRD Sobrino, P Košťál und R Ružarovský (2014). „Offline Programming of an ABB Robot Using Imported CAD Models in the RobotStudio Software Environment“. In: *Applied Mechanics and Materials* 693, S. 62–67. URL: <https://api.semanticscholar.org/CorpusID:62640999>.
- Jiang, J, F Wang, J Shen, S Kim und S Kim (2024). *A Survey on Large Language Models for Code Generation*. arXiv: 2406.00515 [cs.CL]. URL: <https://arxiv.org/abs/2406.00515>.
- Joel, S, JJ Wu und FH Fard (2024). *A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages*. arXiv: 2410.03981 [cs.SE]. URL: <https://arxiv.org/abs/2410.03981>.
- Lambrecht, J, M Chemnitz und J Krüger (2011). „Control layer for multi-vendor industrial robot interaction providing integration of supervisory process control and multifunctional control units“. In: *2011 IEEE Conference on Technologies for Practical Robot Applications*, S. 115–120. DOI: 10.1109/TEPRA.2011.5753492.
- Liang, J, W Huang, F Xia, P Xu, K Hausman, B Ichter, P Florence und A Zeng (2023). *Code as Policies: Language Model Programs for Embodied Control*. arXiv: 2209.07753 [cs.R0]. URL: <https://arxiv.org/abs/2209.07753>.
- Martin, RC (2003). *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall. ISBN: 978-0135974445.
- Maruyama, Y, S Kato und T Azumi (2016). „Exploring the Performance of ROS2“. In: *Proceedings of the 2016 International Conference on Embedded Software (EMSOFT)*. ACM. DOI: 10.1145/2968478.2968502. URL: <https://web.ics.purdue.edu/~rvoyles/Classes/ROS%5C%5FMFET642/Maruyama.ExploringROS2.2016.pdf>.
- Mower, CE, Y Wan, H Yu, A Grosnit, J Gonzalez-Billandon, M Zimmer, J Wang, X Zhang, Y Zhao, A Zhai, P Liu, D Palenicek, D Tateo, C Cadena, M Hutter, J Peters, G Tian, Y Zhuang, K Shao, X Quan, J Hao, J Wang und H Bou-Ammar (2024). *ROS-LLM: A ROS framework for embodied AI with task feedback and structured reasoning*. arXiv: 2406.19741 [cs.R0]. URL: <https://arxiv.org/abs/2406.19741>.
- Mühe, H, A Angerer, A Hoffmann und W Reif (2010). „On reverse-engineering the KUKA Robot Language“. In: *arXiv preprint*. arXiv: 1009.5004 [cs.R0].
- Nakamura, Y (1991). *Advanced Robotics: Redundancy and Optimization*. Reading, MA: Addison-Wesley Publishing Company. ISBN: 978-0201151985.
- Naveed, H, AU Khan, S Qiu, M Saqib, S Anwar, M Usman, N Akhtar, N Barnes und A Mian (2024). *A Comprehensive Overview of Large Language Models*. arXiv: 2307.06435 [cs.CL]. URL: <https://arxiv.org/abs/2307.06435>.
- Nilsson, K (1996). „Industrial Robot Programming“. In: URL: <https://api.semanticscholar.org/CorpusID:109388809>.
- NVIDIA Corporation (2018). *Reinforcement Learning Algorithm Helps Train Thousands of Robots Simultaneously*. NVIDIA Technical Blog. Accessed: September 10, 2025. URL: <https://developer.nvidia.com/blog/nvidia-researchers-develop-reinforcement-learning-algorithm-to-train-thousands-of-robots-simultaneously/>.

- NVIDIA Corporation (2023). *PhysX SDK Documentation*. NVIDIA Corporation. URL: <https://developer.nvidia.com/physx-sdk> (besucht am 15.01.2024).
- Preliy (2024). *Flange: Unity Package for Industrial Robots Simulation*. Kinematic solver with Denavit-Hartenberg parameter implementation. URL: <https://github.com/Preliy/Flange>.
- Quigley, M, B Gerkey, K Conley, J Faust, T Foote, J Leibs, E Berger, R Wheeler und AY Ng (2009). „ROS: an open-source Robot Operating System“. In: *ICRA Workshop on Open Source Software*, S. 1–6. URL: <https://robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf>.
- Salimpour, S, L Fu, F Keramat, L Militano, G Toffetti, H Edelman und JP Queralta (2025). *Towards Embodied Agentic AI: Review and Classification of LLM- and VLM-Driven Robot Autonomy and Interaction*. arXiv: 2508.05294 [cs.R0]. URL: <https://arxiv.org/abs/2508.05294>.
- Schunk (2025). *Schunk FGR Greiferauswahl-Tool*. SCHUNK SE und Co. KG. <https://schunk.com/de/de/konfigurator-fgr>.
- Siciliano, B und O Khatib, Hrsg. (2016). *Springer Handbook of Robotics*. 2. Aufl. Cham, Switzerland: Springer International Publishing. ISBN: 978-3319325507. DOI: 10.1007/978-3-319-32552-1.
- Spong, MW, S Hutchinson und M Vidyasagar (2006). *Robot Modeling and Control*. New York: John Wiley & Sons. ISBN: 978-0471649908.
- Unity Technologies (2025). *Use collisions to trigger other events*. Accessed: 2025-09-02. Unity Technologies. URL: <https://docs.unity3d.com/6000.2/Documentation/Manual/collider-interactions-other-events.html>.
- Valenzo, D, A Ciria, G Schillaci und B Lara (2022). „Grounding Context in Embodied Cognitive Robotics“. In: *Frontiers in Neurorobotics* Volume 16 - 2022. ISSN: 1662-5218. DOI: 10.3389/fnbot.2022.843108. URL: <https://www.frontiersin.org/journals/neurorobotics/articles/10.3389/fnbot.2022.843108>.
- Vaswani, A, N Shazeer, N Parmar, J Uszkoreit, L Jones, AN Gomez, L Kaiser und I Polosukhin (2023). *Attention Is All You Need*. arXiv: 1706.03762 [cs.CL]. URL: <https://arxiv.org/abs/1706.03762>.
- Wang, YJ, B Zhang, J Chen und K Sreenath (2024). *Prompt a Robot to Walk with Large Language Models*. arXiv: 2309.09969 [cs.R0]. URL: <https://arxiv.org/abs/2309.09969>.
- Yoshikawa, T (1985). „Manipulability of Robotic Mechanisms“. In: *The International Journal of Robotics Research* 4.2, S. 3–9. DOI: 10.1177/027836498500400201.

## **Anhang**

### **Anhang A**

...

### **Anhang B**

...

## **Eidesstattliche Erklärung**

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen, als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe.

Ort, Datum

.....  
Vor- und Nachname