Ruhr-Universität Bochum
Lehrstuhl für Produktionssysteme
Prof. Dr.-Ing. Bernd Kuhlenkötter

Entwicklung eines Frameworks zur simulationsbasierten Validierung LLM-generierten Robotercodes in Unity

Dem Fachbereich Maschinenbau der
Ruhr-Universität Bochum
zur Erlangung des akademischen Grades eines
Bachelor of Science

eingereichte Bachelor-Thesis

von
Lukas Lux

Erstprüfer: Prof. Dr.-Ing. Bernd Kuhlenkötter
Zweitprüfer: Prof. Dr.-Ing. Christopher Prinz
Betreuer: Daniel Syniawa, M. Sc.

Datum der Einreichung: 5. September 2025

Aufgabenstellung

Thema: Entwicklung eines Frameworks zur simulationsbasierten Validierung LLM-generierten Robotercodes in Unity

Für den Einsatz von Arbeitsplatzsystemen im Bereich der Montage, an denen Mensch und Roboter miteinander kollaborieren (MRK), gibt es bislang noch keine digitalen Planungswerkzeuge, welche ein MRK-System hinsichtlich Automatisierbarkeit, tech-nisch-wirtschaftlicher Eignung, Ergonomie und Sicherheit simulieren und bewerten können. Um zukünftig die einfache Planung und Simulation von kollaborativen Montagesystemen zu ermöglichen, wird im Forschungsprojekt „KoMPI“ ein digitales Planungswerkzeug entwickelt, sodass sowohl Mensch als auch Roboter gezielt gemeinsam im Montageprozess eingesetzt werden können. Derzeit existieren bereits unterschiedliche Simulationswerkzeuge, die Menschmodelle in manuellen Montagesystemen abbilden. Hierzu zählt bspw. die digitale Planungssoftware Ema der Fa. imk. Bis dato bietet auch Ema keine hinreichenden Möglichkeiten zur Simulation automatisierungstechnischer Komponenten (z. B. Roboter), insbesondere in Kollaboration mit dem Menschen. ... Im Rahmen dieser Arbeit gilt es ...

Im Einzelnen sollten folgende Punkte bearbeitet werden:

- Literaturrecherche zum Thema Mensch-Roboter-Kollaboration (MRK) und Planung kollaborativer Montagesysteme
- ...

Ausgabedatum: 16.06.2025

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Relevanz

1.2 Zielsetzung und Aufbau der Arbeit

Kapitel 2

Grundlagen und Stand der Technik

2.1 Grundlagen der Robotersimulation

2.2 Large Language Models (LLMs)

2.3 Stand der Technik: LLMs in der Robotik

2.4 Stand der Technik: Robotersimulation

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. Im Kern ist die Roboterprogrammierung ein Steuerungsproblem, das neben den allgemeinen Aspekten der Computerprogrammierung auch roboterspezifische Herausforderungen umfasst.[1] Ein entscheidender Vorteil der Roboterprogrammierung im Gegensatz zur herkömmlichen Steuertechnik ist dabei die Möglichkeit, einen realen Prozess mittels Programmcode abbilden und verändern zu können und somit die Effizienz und Anpassungsfähigkeit eines industriellen Prozessen zu erhöhen.

2.4.1 Offline-Programmierung und Robotersimulationsprogramme

Da Anlagen oft dauerhaft in Betrieb sind un die Notwendigkeit von Prozessveränderungen bei der Fertigung von Produkten unabdingbar ist, wird eine Möglichkeit benötigt, industrielle Anlagen schon bevor der Inbetriebnahme, Umfunktionierung oder Verbesserung testen zu können. Auch bei Robotern ist dies, unter anderem aufgrund hoher Kosten und Sicherheitsrisiken bei fehlerhafter Konifguration, unabdingbar. Die **Offline-Programmierung (OLP)** ermöglicht die Entwicklung eines Steuerungsprogramms, ohne dass ein physischer Roboter erforderlich ist. Ein wesentlicher Vorteil ist die Möglichkeit, Programme in einer virtuellen Umgebung zu erstellen und zu simulieren. Die Methode nutzt 3D-CAD-Modelle, um den Körper des Roboters möglichst originalgetreu dar-

[1] Vgl. Klas Nilsson: Industrial Robot Programming, in: 1996, URL: https://api.semanticscholar.org/CorpusID:109388809, S. 1 ff.

zustellen. Ziel der Offline-Programmierung ist es, den visuellen Prozess schon vorher evaluieren zu können und Probleme im Ablauf früher zu erkennnen.[2]

2.4.2 Limitierende Faktoren proprietärer Systeme

Traditionelle Robotersteuerungssysteme sind oft durch eine **proprietäre Hardware- und Softwarearchitektur** begrenzt.[3] Dies führt dazu, dass Anwender für jede Robotermarke eine spezifische Programmiersprache mit unterschiedlichen Befehlen und Funktionen erlernen müssen. Solche Systeme verursachen **Zeitverlust** in der Produktion, da jede Änderung eine Unterbrechung erfordert, sowie **Codeduplikation**, da Programme nicht zwischen Robotermarken ausgetauscht werden können.[4]Um diese Beschränkungen zu überwinden, ist eine akkurate Modellierung der physischen Welt, in welcher der Roboter agieren soll, in der Simulation unabdingbar.

2.4.3 Physics-Engines

Dies führt zum Einsatz von Physics-Engines. Um eine realistische Simulation zu gewährleisten, sind diese unerlässlich. Sie modellieren dynamische Interaktionen wie Kollisionen, Schwerkraft und Reibung, was für die präzise Nachbildung des Roboterverhaltens entscheidend ist. Obwohl die Genauigkeit dieser Engines als nicht perfekt angesehen wird, da sie die reale Welt nicht exakt abbilden, sind sie für die Forschung in den Bereichen Deep Learning und Digital Twins von entscheidender Bedeutung.[5] Die Wahl der Physics-Engine ist ein kritischer Faktor und beeinflusst die Stabilität und Wiederholbarkeit der Simulation. Häufig genutzte Engines sind PhysX, Bullet und ODE.

[2]Vgl. Radovan Holubek u. a.: Offline Programming of an ABB Robot Using Imported CAD Models in the RobotStudio Software Environment, in: Applied Mechanics and Materials 693 (2014), S. 62–67, URL: https://api.semanticscholar.org/CorpusID:62640999, hier S. 62 ff.

[3]Vgl. Pietro Bilancia u. a.: An Overview of Industrial Robots Control and Programming Approaches, in: Applied Sciences 13.4 (2023), URL: https://www.mdpi.com/2076-3417/13/4/2582, S. 1.

[4]Vgl. ders.: An Overview of Industrial Robots Control and Programming Approaches, in: Applied Sciences 13.4 (2023), URL: https://www.mdpi.com/2076-3417/13/4/2582, S. 3.

[5]Vgl. Florent P. Audonnet u. a.: A Systematic Comparison of Simulation Software for Robotic Arm Manipulation using ROS2, in: 2022 22nd International Conference on Control, Automation and Systems (ICCAS) 2022, S. 755–762, URL: https://api.semanticscholar.org/CorpusID:248157288, 1/psqq.

Kapitel 3

Implementierung des Frameworks

In diesem Kapitel werden die technischen Grundlagen sowie das Vorgehen zur Implementierung des Frameworks beschrieben.

3.1 Architektur des Frameworks

3.1.1 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 Simulationplattform 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

3.1.2 Unity3D als Simulationsplattform

Auswahl und Vorteile gegenüber Alternativen

Die Wahl von Unity3D als zugrundeliegende Simulationsplattform basiert auf mehreren technischen und praktischen Erwägungen. Während es bereits mehrere kommerzielle Programme für die Gestaltung und Simulation von Robotern in virtuellen Umgebungen gibt, sind diese nur selten mit anderen CAD-Systemen und Robotern kompatibel, unterstützen nicht alle Roboterbibliotheken oder werden nur plattformabhängig angeboten

(vgl.[1]). Unity3D hingegen ist mit den meisten CAD-Systemen kompatibel und bietet eine plattformübergreifende Lösung.

3D-Rendering und Physik-Simulation

Unity3D bietet eine ausgereifte 3D-Rendering-Pipeline mit integrierter Physik-Engine, welche zur Simulation von Gegenständen mit realitätsnahem Verhalten sowie komplexen Arbeitsräumen geeignet ist (vgl.,[2][3]). Die Engine wurde bereits erfolgreich in der wissenschaftlichen Forschung eingesetzt und bietet Module und Plugins für spezifische Anwendungsfälle im Simulationsbereich. Unity3D ermöglicht es auch Nicht-Programmierern, leistungsstarke Animations- und Interaktionsdesign-Tools zu nutzen, um Roboter visuell zu programmieren und zu animieren (vgl.[4]).

Technische Architektur und Programmierung

Technisch ermöglicht Unity3D durch seine Scripting-Runtime (basierend auf Mono/.NET Framework) die Verwendung moderner C#-Sprachfeatures für nebenläufige Prozesse und asynchroner Programmierung,[5] was es ermöglicht, Visualisierung, Datenakquise und Überwachung zu trennen. Die .NET-basierte Architektur unterstützt dabei sowohl Task-basierte asynchrone Operationen als auch Coroutines für zeitgesteuerte Prozesse,[6] welche die notwendige periodische Ausführung von Prozessen auf verschiedenen Ebenen des Frameworks stark vereinfacht.

Die Unity-Engine unterstützt nativ Multithreading durch das Job System,[7] was für die parallele Verarbeitung von Sensordaten, Kollisionserkennung und Bewegungsplanung entscheidend ist.

[1] Vgl. Víctor Hugo Andaluz u. a.: Unity3D-MatLab Simulator in Real Time for Robotics Applications, in: Lucio Tommaso De Paolis/Antonio Mongelli (Hrsg.): Augmented Reality, Virtual Reality, and Computer Graphics, Cham 2016, S. 246–263, hier S. 247.

[2] Vgl. Unity Technologies: System requirements for Unity 6.2 Beta, Unity 6000.2 Documentation, Unity Technologies, 2025, URL: https://docs.unity3d.com/6000.2/Documentation/Manual/system-requirements.html (besucht am 01. 09. 2025).

[3] Vgl. ders.: Built-in 3D physics, Unity 6000.2 Documentation Manual, Unity Technologies, 2025, URL: https://docs.unity3d.com/6000.2/Documentation/Manual/PhysicsOverview.html (besucht am 01. 09. 2025).

[4] Vgl. Christoph Bartneck u. a.: 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) 2015, S. 431–437, hier S. 431.

[5] Vgl. Unity Technologies: Unity Scripting Reference: Async and Await, Unity Technologies, 2023, URL: https://docs.unity3d.com/Manual/JobSystemAsyncAwait.html (besucht am 15. 01. 2024), S. 45-52.

[6] Vgl. ders.: Coroutines, Unity Technologies, 2023, URL: https://docs.unity3d.com/Manual/Coroutines.html (besucht am 15. 01. 2024), S. 123-135.

[7] Vgl. ders.: C# Job System, Unity Technologies, 2023, URL: https://docs.unity3d.com/Manual/JobSystem.html (besucht am 15. 01. 2024), S. 201-218.

Entwicklungsumgebung und Debugging-Tools

Ein weiterer entscheidender Vorteil für die Robotik-Simulation liegt in der Verfügbarkeit visueller Programmiertools und der integrierten Entwicklungsumgebung. Die Plattform bietet umfangreiche Debugging- und Profiling-Werkzeuge (Unity Profiler, Frame Debugger), die während der Entwicklung und zur Laufzeit genutzt werden können.[8] Diese Werkzeuge ermöglichen die Analyse von Performance-Engpässen bei der Verarbeitung von Roboterdaten und die Optimierung der Sicherheitsmonitor-Updates.

Darüber hinaus lassen sich während der Laufzeit sowohl die Szene (hier: die Roboterzelle) als auch Komponenten-Parameter in Echtzeit bearbeiten und einsehen,[9] was das Debuggen und Testen beschleunigt.

Benutzerfreundlichkeit und industrielle Anwendung

Besonders relevant für industrielle Robotik-Anwendungen ist die Möglichkeit, Custom Editor Scripts zu entwickeln (vgl.[10]), die eine benutzerfreundliche und niederschwellige Konfiguration verschiedener Parameter ermöglichen. Zusätzlich ermöglichen Gizmos und Scene View die visuelle Darstellung von Kollisionszonen, Singularitätspunkten und Prozessabläufen während der Entwicklung (vgl.[11]).

3.1.3 Design Patterns und Prinzipien

Die Entwicklung eines modularen und erweiterbaren Robotersicherheitssystems erfordert eine fundierte methodische Herangehensweise, die auf bewährten Software-Engineering-Prinzipien basiert. Die Auswahl geeigneter Design Patterns und Architekturprinzipien determiniert maßgeblich die Qualitätsattribute des Systems wie Wartbarkeit, Testbarkeit und Erweiterbarkeit (vgl.,[12] S. 73-75). Im Folgenden werden die für diese Arbeit gewählten Entwurfsmuster und deren Begründung dargelegt.

Observer Pattern als Kommunikationsparadigma

Für die Realisierung der systemweiten Kommunikation wird das Observer Pattern (vgl.,[13] S. 293-303) als zentrales Entwurfsmuster gewählt. Diese Entscheidung basiert auf drei wesentlichen Anforderungen industrieller Robotersysteme: Erstens müssen Sicherheitsereignisse ohne Verzögerung an alle relevanten Systemkomponenten propagiert

[8] Vgl. DERS.: Unity Profiler Window, Unity Technologies, 2023, URL: https://docs.unity3d.com/Manual/ProfilerWindow.html (besucht am 15. 01. 2024), S. 67-89.

[9] Vgl. J. HAAS: Real-time Parameter Tuning in Unity for Robotics Applications, in: Proceedings of the International Conference on Robotics and Automation (ICRA), Philadelphia, PA 2022, S. 1234–1240, S. 1236.

[10] Vgl. UNITY TECHNOLOGIES: Custom Editors, Unity Technologies, 2023, URL: https://docs.unity3d.com/ScriptReference/Editor.html (besucht am 15. 01. 2024), S. 156-172.

[11] Vgl. DERS.: Gizmos and Handles, Unity Technologies, 2023, URL: https://docs.unity3d.com/Manual/GizmosAndHandles.html (besucht am 15. 01. 2024), S. 234-245.

[12] Vgl. Len BASS u. a.: Software Architecture in Practice, 3. Aufl., Boston, MA 2012.

[13] Vgl. Erich GAMMA u. a.: Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA 1994.

werden, was durch die inhärente Entkopplung des Observer Patterns gewährleistet wird. Zweitens ermöglicht das Muster die dynamische Registrierung und Deregistrierung von Beobachtern zur Laufzeit, was für modulare Sicherheitssysteme unerlässlich ist (vgl.,[14] S. 127-128). Drittens reduziert die lose Kopplung zwischen Publisher und Subscriber die Systemkomplexität erheblich, da Komponenten ohne Kenntnis voneinander interagieren können.

Das Observer Pattern adressiert zudem die Herausforderung der Multi-Threading-Umgebung in Unity3D, indem Events asynchron verarbeitet werden können, ohne den Hauptthread zu blockieren (vgl.,[15] S. 156-159). Dies ist besonders kritisch für die Echtzeitverarbeitung von Sensordaten und die gleichzeitige Visualisierung.

Strategy Pattern für algorithmische Flexibilität

Die Wahl des Strategy Patterns (vgl.,[16] S. 315-323) für die Implementierung von Sicherheitsmonitoren und Datenparser begründet sich durch die Heterogenität industrieller Robotersysteme. Verschiedene Roboterhersteller verwenden proprietäre Kommunikationsprotokolle und Datenformate, was eine flexible Austauschbarkeit von Parsing-Algorithmen erfordert (vgl.,[17] S. 891-893). Das Strategy Pattern kapselt diese Algorithmen in separaten Klassen und macht sie über eine gemeinsame Schnittstelle austauschbar.

Die Vorteile dieser Architekturentscheidung manifestiert sich vornehmlich darin, dass so Robotertypen ohne Modifikation des Kernsystems integriert werden. Martin2003 spricht hier vom Open-Closed-Prinzip, welches die Offenheit von Software-Entitäten (Funktionen, Klassen, Module, Komponenten usw.) zur Extension und die gleichzeitige Geschlossenheit zur Modfikation beschreibt.[18]

Adapter Pattern zur Hardware-Abstraktion

Die Integration heterogener Hardware-Komponenten erfordert eine Abstraktionsschicht zwischen der Anwendungslogik und den hardware-spezifischen Schnittstellen. Das Adapter Pattern (vgl.,[19] S. 139-150) wird gewählt, um diese Abstraktion zu realisieren. Die Notwendigkeit ergibt sich aus der Vielfalt der Robotersteuerungen und Visua-

[14] Vgl. Frank Buschmann u. a.: Pattern-Oriented Software Architecture: A System of Patterns, Chichester, UK 1996.

[15] Vgl. Robert Nystrom: Game Programming Patterns, 2014, url: `https://gameprogrammingpatterns.com/`.

[16] Vgl. Erich Gamma u. a.: Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA 1994.

[17] Vgl. Bruno Siciliano/Oussama Khatib: Springer Handbook of Robotics, in: Springer Handbooks 2016, S. 1–2227.

[18] Vgl. Robert C. Martin: Agile Software Development: Principles, Patterns, and Practices, Upper Saddle River, NJ 2003.

[19] Vgl. Erich Gamma u. a.: Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA 1994.

lisierungssysteme, die jeweils eigene APIs und Datenformate verwenden (vgl.,[20] S. 412-415).

Durch die Adapter-Schicht wird eine einheitliche Schnittstelle zur Verfügung gestellt, die es ermöglicht, verschiedene Robotersysteme ohne Änderung der Kernlogik anzubinden. Dies reduziert nicht nur die Komplexität des Systems, sondern erhöht auch dessen Portabilität und Wiederverwendbarkeit (vgl.,[21] S. 89-91).

SOLID-Prinzipien als Qualitätsfundament

Die konsequente Anwendung der SOLID-Prinzipien (vgl.,[22] S. 95-135) bildet das methodische Fundament der Systemarchitektur. Das *Single Responsibility Principle* wird angewendet, um kohäsive Module zu schaffen, die genau eine Verantwortlichkeit haben. Dies reduziert die Kopplung und erhöht die Verständlichkeit des Codes (vgl.,[23] S. 62-64). Das *Open-Closed Principle* gewährleistet, dass das System für Erweiterungen offen, aber für Modifikationen geschlossen ist – eine essenzielle Eigenschaft für langlebige Industriesysteme.

Das *Dependency Inversion Principle* wird konsequent angewendet, indem High-Level-Module von Abstraktionen abhängen, nicht von konkreten Implementierungen. Dies ermöglicht die flexible Konfiguration des Systems zur Laufzeit und vereinfacht die Integration in verschiedene Produktionsumgebungen (vgl.,[24] S. 112-115).

Event-Driven Architecture für Echtzeitfähigkeit

Die Entscheidung für eine event-getriebene Architektur basiert auf den Echtzeitanforderungen industrieller Robotersysteme. Sicherheitskritische Ereignisse müssen innerhalb definierter Zeitschranken verarbeitet werden, was durch synchrone Aufrufketten nicht gewährleistet werden kann (vgl.,[25] S. 97-99). Die event-getriebene Architektur ermöglicht die asynchrone Verarbeitung von Ereignissen und die Priorisierung kritischer Sicherheitsereignisse.

Zudem adressiert dieser Ansatz die Herausforderung der Integration verschiedener Datenquellen – von zyklischen Sensordaten über sporadische Alarme bis zu kontinuierlichen Videoströmen. Jede Datenquelle kann Events in ihrem eigenen Takt generieren, ohne andere Systemkomponenten zu blockieren (vgl.,[26] S. 234-237).

[20] Vgl. John J. Craig: Introduction to Robotics: Mechanics and Control, 3rd, Upper Saddle River, NJ 2005.
[21] **Vlissides1995**.
[22] Vgl. Robert C. Martin: Agile Software Development: Principles, Patterns, and Practices, Upper Saddle River, NJ 2003.
[23] Vgl. ders.: Clean Architecture: A Craftsman's Guide to Software Structure and Design, Boston, MA 2017.
[24] **Fowler2018**.
[25] **Hohpe2003**.
[26] **Vernon2013**.

3.1.4 Systemarchitektur und Entwurfsmuster

Flange Framework als Visualisierungs- und Konfigurationstool

Im Rahmen dieses Frameworks wird das Unity-Package *Flange* genutzt. Implementiert durch GitHub-User *Preliy* und frei verfügbar im Rahmen einer BSD 3-Clause Lizenz, bietet ein spezialisiertes Framework für die Robotersteuerung in Unity mit modularer Architektur, die Gelenksteuerung, Kinematik, Koordinatentransformationen und Echtzeitüberwachung als separate Komponenten organisiert. Das System unterstützt sowohl direkte Gelenkmanipulation auf niedriger Ebene als auch kartesische Steuerung auf höherer Abstraktionsebene, wodurch es für verschiedene Roboteranwendungen einsetzbar ist.[27] Im Kontext dieser Entwicklungsarbeit wird Flange zur Implementierung und Visualisierung der Denavit-Hartenberg-Parameter und damit einhergehender Achstransformationen eingesetzt, da diese Notation als fundamentales Werkzeug 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.[28]

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

Schichtenarchitektur

Aufbauend auf den oben genannte Prinzipien teilt sich das Framework in 4 logisch getrennte Module auf:

1. **Kernlogik**, welches den Status des Roboters beinhaltet und verwaltet als zentrale Schnittstelle (Core)
2. **Interfaces**, welche die Kommunkationsschnittstellen und -methoden zwischen den Modulen implementiert (Interfaces)
3. **Monitoring**, welches die einzelnen Komponenten des Monitoring-Systems implementiert (Monitors)
4. **Adapters**, welches potenziell Adapter zu veschiedenen physischen oder simulierten Steuerungen von Robotern beinhaltet (Hier ABB gennant)

Beispielhaft wird im Folgenden erläutert, aus welchen Bestandenteilen Interfaces des Framework bestehen. Da jedes Modul, welches von einer Interface erbt zwangsweise

[27] Vgl. Preliy: Flange: Unity Package for Industrial Robots Simulation, Kinematic solver with Denavit-Hartenberg parameter implementation, 2024, URL: https://github.com/Preliy/Flange.

[28] Vgl. Peter I. Corke: A Simple and Systematic Approach to Assigning Denavit–Hartenberg Parameters, in: IEEE Transactions on Robotics 23.3 (2007), S. 590–594, hier S. 590.

dessen Komponenten implenentieren muss, lassen sich hierdurch die grundlegenden Funktionalitäten des Framework abstrahieren.

Interfaces

Die Trennung der Module ermöglicht eine formalisierte Schnittstellenkommunikation, dargestellt in Abbildung 3.1. Jede Schicht hat verschiedene Verantwortlichkeiten und Schnittstellen innerhalb des Monitoring-Systems. Die Robot Communication Layer organisiert die Kommunkation mit der zugrundeliegenden Robotersteuerung und implentiert eine Interface des Typs IRobotConnector. Als Verbindungsclient zwischen Unity und externer Schnittstelle des Robotersystems implementiert dieser eine IRobotConnector Interface, welche als Vorlage für eine Anbindung an eine Robotersteuerung jedes Typs fungiert und standartisierte Methoden implementiert:

```
1  namespace RobotSystem.Interfaces
2  {
3      public interface IRobotConnector
4      {
5          event System.Action<RobotSystem.Core.RobotState>
               OnRobotStateUpdated;
6          event System.Action<bool> OnConnectionStateChanged;
7
8          bool IsConnected { get; }
9          RobotSystem.Core.RobotState CurrentState { get; }
10
11         void Connect();
12         void Disconnect();
13     }
14 }
```

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

Die Interface definiert Events (Aktionen), die bei definierten Zustandsänderungen in der Laufzeit ausgeführt werden. Andere Bestandteile des Framework sind in der Lage, ein Event zu abonnieren und eine Methode registieren, welche ausgeführt werden soll, wenn dieses Event auftritt. Ist dies der Fall, wird das entsprechende Modul über die Änderung benachrichtigt und bekommt gegebenfalls 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 gegebenfalls Statusänderungen abzufragen.

Weiterführend definiert die 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 { *get;* } können Attribute auch von ausserhalb des RobotConnectors abgefragt werden, jedoch nicht überschrieben.

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

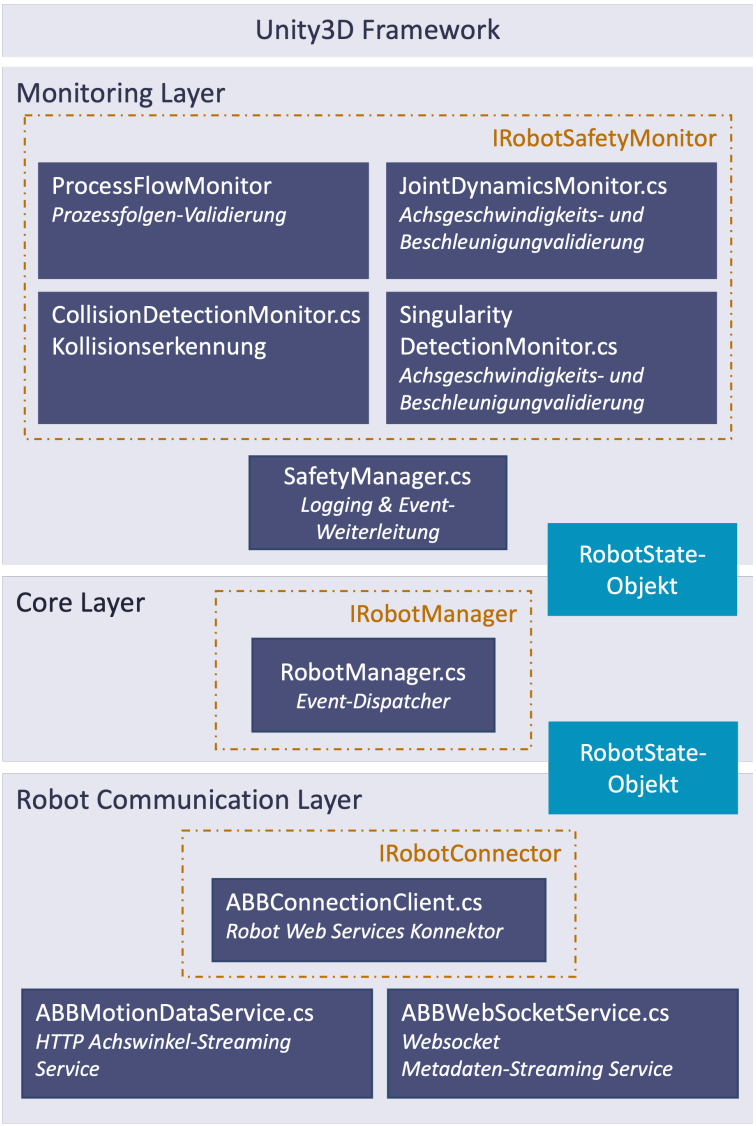

Abbildung 3.1: Schichtenarchitektur des Frameworks mit den wichtigsten zugehörigen Modulen. Module mit grüner Umrandung werden durch ine Interface formalisiert.

```
RobotSystem/
└── Core/
    ├── RobotManager.cs
    ├── RobotState.cs
    ├── RobotSafetyManager.cs
    ├── SafetyEvent.cs
    └── RobotStateSnapshot.cs
```

```
    ├── Part.cs
    ├── Station.cs
    └── RapidTargetGenerator.cs
├── Interfaces/
    ├── IRobotConnector.cs
    ├── IRobotDataParser.cs
    ├── IRobotSafetyMonitor.cs
    └── IRobotVisualization.cs
├── ABB/
    ├── RWS/
    │   ├── ABBRWSConnectionClient.cs
    │   ├── ABBRWSDataParser.cs
    │   └── ABBMotionDataService.cs
    └── ABBFlangeAdapter.cs
└── Monitors/
    ├── CollisionDetectionMonitor.cs
    ├── JointDynamicsMonitor.cs
    ├── ProcessFlowMonitor.cs
    └── SingularityDetectionMonitor.cs
```

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

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

SafetyEvent und RobotStateSnapshot Das `SafetyEvent`-System implementiert ein umfassendes Ereignismodell:

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

Datenformate

- **HTTPS/HTTP**: RESTful API-Zugriff mit Digest-Authentifizierung
- **WebSocket Secure (WSS)**: Bidirektionale Echtzeit-Kommunikation

- **XML**: Strukturierte Datenübertragung mit Schema-Validierung
- **JSON**: Interne Datenrepräsentation und Logging-Format

as

3.2 Implementierung der Module

3.2.1 Allgemeine Struktur

3.2.2 Prozessfolgen

User Input: Simulationsumgebung und Verbindung zu Robot Studio

Wie kann ich Prozessfolgen überprüfen? Prozessfolge: Folge and Arbeitsschritten eines Arbeitsprozesses, hier Roboter -> Was wird in welcher Reihenfolge wohin bewegt? Wie kann ich das Messen? - Bewegt sich das Werkstück von Position Start zu Position Ziel? - Bewegen sich die Werkstücke in der richtigen Reihenfolge von Start zu Ziel? Benötigt: Definition von Start & Zielpositionen einzelner Werkstücke

3.2.3 Kollisionen

Theoretische Grundlagen

Kollisionen eines Roboters mit Objekten im Arbeitsraum stellen eine zentrales Problem beim Neuentwurf und der Testung neu entwickelten Codes dar. Kollidiert ein industrieller Roboter unvorhergesehen mit seinem Arbeitsraum, kann dies im schlimmsten Fall zu einem Ausfall des am Produktionsschrit beteiligen Roboters

Implementierung

3.2.4 Singularitäten

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[29].

Mathematische Definition Eine kinematische Singularität tritt auf, wenn die Jacobi-Matrix des Roboters ihren vollen Rang verliert:

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

[29] **craig2005introduction**; Bruno Siciliano u. a.: Robotics: Modelling, Planning and Control, London 2009.

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 Robotergelenke darstellt.

Die Jacobi-Matrix beschreibt die Beziehung zwischen Gelenkgeschwindigkeiten $\dot{\boldsymbol{\theta}}$ und kartesischen Geschwindigkeiten des TCP $\mathbf{v}$:

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

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

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

Dadurch wird die inverse Kinematik nicht eindeutig lösbar ist und es können theoretisch unendliche Gelenkgeschwindigkeiten auftreten (vgl.[30]).

Auswirkungen auf die Robotersteuerung Singularitäten haben mehrere kritische Auswirkungen auf die Robotersteuerung:

- **Kontrollverlust**: Verlust der Beweglichkeit in bestimmten kartesischen Richtungen
- **Numerische Instabilität**: Große Konditionszahlen führen zu numerischen Problemen bei der inversen Kinematik
- **Hohe Gelenkgeschwindigkeiten**: Kleine kartesische Bewegungen können große Gelenkbewegungen erfordern
- **Sicherheitsrisiken**: Unkontrollierte Bewegungen können zu Kollisionen oder Beschädigungen führen

Klassifikation der Singularitätstypen Für serielle Robotermanipulatoren mit sechs Freiheitsgraden (wie ABB IRB-Roboter) können drei primäre Singularitätstypen unterschieden werden[31]:

Boundary Singularities (Randsingularitäten): Treten auf, wenn der Roboter die Grenzen seines Arbeitsraums erreicht, typischerweise bei vollständig ausgestreckter oder eingeklappter Konfiguration. Mathematisch charakterisiert durch:

$$\sum_{i=1}^{3} a_i \cos(\theta_i) + d_4 = R_{\text{max/min}} \tag{3.4}$$

wobei a_i die DH-Parameter-Längen, d_4 der Offset und $R_{\text{max/min}}$ die maximale/minimale Reichweite darstellen.

[30] Yoshihiko Nakamura: Advanced Robotics: Redundancy and Optimization, Reading, MA 1991.
[31] Mark W. Spong u. a.: Robot Modeling and Control, New York 2006.

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

$$\mathbf{z}_4 \parallel \mathbf{z}_6 \text{ oder } |\mathbf{z}_4 \cdot \mathbf{z}_6| \approx 1 \tag{3.5}$$

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

Elbow Singularities (Ellbogensingularitäten): Treten auf, wenn der Roboterarm vollständig gestreckt oder eingeklappt ist, sodass die ersten drei Gelenke kollinear werden. Charakterisiert durch:

$$\sin(\theta_3) = 0 \text{ oder } \theta_2 + \theta_3 = 0/180 \tag{3.6}$$

Standardmetriken und Bewertungsverfahren

Zur quantitativen Bewertung von Singularitäten haben sich in der Robotik verschiedene Metriken etabliert, die jeweils unterschiedliche Aspekte der kinematischen Eigenschaften erfassen.

Manipulierbarkeitsindex (Yoshikawa-Maß) Der von Yoshikawa[32] eingeführte Manipulierbarkeitsindex ist eine der am häufigsten verwendeten Metriken:

$$\mu(\boldsymbol{\theta}) = \sqrt{\det(\mathbf{J}(\boldsymbol{\theta})\mathbf{J}^T(\boldsymbol{\theta}))} \tag{3.7}$$

Für quadratische Jacobi-Matrizen vereinfacht sich dies zu:

$$\mu(\boldsymbol{\theta}) = |\det(\mathbf{J}(\boldsymbol{\theta}))| \tag{3.8}$$

Der Index nimmt Werte zwischen 0 (Singularität) und einem maximalen Wert an, wobei höhere Werte bessere Manipulierbarkeit indizieren.

Konditionszahl der Jacobi-Matrix Die Konditionszahl quantifiziert die numerische Stabilität der inversen Kinematik:

$$\kappa(\mathbf{J}) = \frac{\sigma_{\max}(\mathbf{J})}{\sigma_{\min}(\mathbf{J})} \tag{3.9}$$

wobei $\sigma_{\max}$ und $\sigma_{\min}$ die größten und kleinsten Singulärwerte der Jacobi-Matrix darstellen. Werte nahe 1 indizieren gute Konditionierung, während große Werte ($\kappa > 10^6$) auf Singularitätsnähe hinweisen.

[32] Tsuneo Yoshikawa: Manipulability of Robotic Mechanisms, in: The International Journal of Robotics Research 4.2 (1985), S. 3–9.

Determinanten-basierte Ansätze Neben dem Yoshikawa-Maß werden häufig normalisierte Determinanten verwendet:

$$\mu_{\text{norm}}(\boldsymbol{\theta}) = \frac{|\det(\mathbf{J}(\boldsymbol{\theta}))|}{\prod_{i=1}^{n} \|\mathbf{j}_i\|} \tag{3.10}$$

wobei $\mathbf{j}_i$ die i-te Spalte der Jacobi-Matrix darstellt.

Vergleich der Detektionsmetriken Die verschiedenen Metriken haben unterschiedliche Eigenschaften:

- **Yoshikawa-Maß**: Geometrisch interpretierbar, aber skalierungsabhängig
- **Konditionszahl**: Numerisch robust, aber rechenintensiv
- **Normalisierte Determinante**: Skalierungsunabhängig, aber komplexer zu berechnen
- **Achsenbasierte Methoden**: Geometrisch intuitiv, echtzeitfähig

Bestehende Lösungsansätze

Die Robotik-Literatur bietet verschiedene Ansätze zur Behandlung von Singularitäten, die sich in präventive und reaktive Strategien unterteilen lassen.

Singularitätsvermeidung durch Pfadplanung Präventive Ansätze vermeiden Singularitäten bereits in der Pfadplanungsphase:

- **Konfigurationsraum-Methoden**: Identifikation singularitätsfreier Pfade im Gelenkraum[33]
- **Manipulierbarkeits-optimierte Planung**: Maximierung der Manipulierbarkeit entlang des Pfades[34]
- **Redundanz-basierte Vermeidung**: Nutzung kinematischer Redundanz zur Singularitätsvermeidung[35]

Damped Least Squares Methoden Reaktive Ansätze behandeln Singularitäten durch modifizierte inverse Kinematik:

$$\dot{\boldsymbol{\theta}} = (\mathbf{J}^T\mathbf{J} + \lambda^2\mathbf{I})^{-1}\mathbf{J}^T\mathbf{v} \tag{3.11}$$

[33] Jean-Claude Latombe: Robot Motion Planning, Boston, MA 1991.
[34] Nikolaus Vahrenkamp u. a.: Manipulability Analysis for Mobile Manipulators, in: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 2009, S. 2317–2323.
[35] Bruno Siciliano: Kinematically Redundant Robot Manipulators, in: Springer Handbook of Robotics 1990, Chapter 11, S. 245–268.

wobei λ der Dämpfungsparameter ist, der adaptiv basierend auf der Singularitätsnähe angepasst wird[36].

Singularity-robust inverse Kinematik Erweiterte Methoden kombinieren verschiedene Techniken:

- **Selectively Damped Least Squares**: Richtungsabhängige Dämpfung[37]
- **Jacobian Transpose Methods**: Alternative zur Pseudoinversen[38]
- **Task Priority Methods**: Hierarchische Aufgabenbehandlung[39]

Industrielle Implementierungen Kommerzielle Robotersysteme nutzen herstellerspezifische Ansätze:

- **ABB**: Adaptive Geschwindigkeitsreduktion und Warnungen bei Singularitätsnähe
- **KUKA**: Singularitätsvermeidung durch alternative Konfigurationen
- **Fanuc**: Kombinierte Pfadplanung und Echtzeitüberwachung
- **Universal Robots**: Kollaborative Singularitätsbehandlung mit Bedienerinteraktion

Framework-spezifische Implementierung

Das entwickelte Framework implementiert eine neuartige achsenbasierte Singularitätsdetektion, die geometrische Eigenschaften der Roboterkinematik direkt nutzt, anstatt auf rechenintensive Jacobi-Matrix-Berechnungen angewiesen zu sein.

Mathematische Herleitung der achsenbasierten Methode Die achsenbasierte Methode basiert auf der Erkenntnis, dass Singularitäten geometrisch durch die Kollinearität von Rotationsachsen charakterisiert werden können. Für einen 6R-Roboter mit sphärischem Handgelenk lässt sich zeigen, dass:

Für Handgelenksingularitäten gilt:

$$\operatorname{rank}(\mathbf{J}_{\text{wrist}}) < 3 \Leftrightarrow \mathbf{z}_4 \parallel \mathbf{z}_6 \tag{3.12}$$

Die Kollinearität wird über das normalisierte Skalarprodukt quantifiziert:

$$c_{46} = \frac{|\mathbf{z}_4 \cdot \mathbf{z}_6|}{\|\mathbf{z}_4\|\|\mathbf{z}_6\|} = |\cos(\alpha_{46})| \tag{3.13}$$

[36] Yoshihiko Nakamura/Hideo Hanafusa: Inverse Kinematic Solutions With Singularity Robustness for Robot Manipulator Control, in: Journal of Dynamic Systems, Measurement, and Control 108.3 (1986), S. 163–171.
[37] **buss2004selectively**.
[38] **wolovich1984computational**.
[39] **siciliano1991general**.

wobei α_{46} der Winkel zwischen den Achsen ist.

Für Schultersingularitäten wird die Distanz zwischen Schulter- und Handgelenkzentrum analysiert:

$$d_{\text{shoulder}} = \|\mathbf{p}_{\text{wrist}} - \mathbf{p}_{\text{shoulder}}\| \tag{3.14}$$

Die Singularitätsnähe wird durch einen kontinuierlichen Manipulierbarkeitsindex approximiert:

$$\mu_{\text{approx}} = \begin{cases} 1 - \frac{c_{46} - \tau_{\text{wrist}}}{1 - \tau_{\text{wrist}}} \cdot \alpha_{\text{scale}} & \text{wenn } c_{46} > \tau_{\text{wrist}} \\ 1.0 & \text{sonst} \end{cases} \tag{3.15}$$

Algorithmus-Analyse des SingularityDetectionMonitor Der `SingularityDetectionMonitor` implementiert einen effizienten Echtzeit-Algorithmus mit folgender Struktur:

Schritt 1: Achsentransformation Die aktuellen Gelenkachsen werden aus den Flange-Transformationsmatrizen extrahiert:

$$\mathbf{z}_i = \mathbf{T}_i[:3, 2] \quad \text{für } i = 1, \ldots, 6 \tag{3.16}$$

Schritt 2: Kollinearitätsanalyse Für jedes relevante Achsenpaar wird die Kollinearität berechnet:

$$c_{14} = |\mathbf{z}_1 \cdot \mathbf{z}_4| \quad \text{(Schultersingularität)} \tag{3.17}$$
$$c_{46} = |\mathbf{z}_4 \cdot \mathbf{z}_6| \quad \text{(Handgelenksingularität)} \tag{3.18}$$
$$c_{25} = |\mathbf{z}_2 \cdot \mathbf{z}_5| \quad \text{(Ellbogensingularität)} \tag{3.19}$$

Schritt 3: Schwellwertvergleich und Klassifikation Die berechneten Kollinearitätswerte werden mit konfigurierbaren Schwellwerten verglichen:

$$\tau_{\text{wrist}} = 0.98 \quad (\cos(11.5)) \tag{3.20}$$
$$\tau_{\text{shoulder}} = 0.93 \quad (\cos(22)) \tag{3.21}$$
$$\tau_{\text{elbow}} = 0.95 \quad (\cos(18)) \tag{3.22}$$

Schritt 4: Manipulierbarkeitsberechnung Für detektierte Singularitäten wird ein approximativer Manipulierbarkeitsindex berechnet, der die Nähe zur Singularität quantifiziert.

Praktische Umsetzung in Unity Die Integration in Unity erfolgt über das Flange-Framework mit folgenden Optimierungen:

- **Frame-Rate-Optimierung**: Berechnung mit 5 Hz zur Balance zwischen Genauigkeit und Performance
- **Caching**: Wiederverwendung von Transformationsmatrizen zwischen Berechnungszyklen
- **Early Exit**: Abbruch der Berechnung bei eindeutigen Nicht-Singularitäten

- **Vectorized Operations**: Nutzung von Unity's Vector3-Operationen für SIMD-Optimierung

Achsenbasierte Singularitätsdetektion

Die Schwellwerte sind konfigurierbar definiert als:

$$\tau_{\text{wrist}} = 0.98 \quad (\cos(11.5°)) \tag{3.23}$$
$$\tau_{\text{shoulder}} = 0.93 \quad (\cos(22°)) \tag{3.24}$$
$$\tau_{\text{general}} = 0.95 \quad (\cos(18°)) \tag{3.25}$$

Praktische Implementierung im Unity-Framework

Die Singularitätsdetektion ist in der Klasse `RobotSafetyMonitor` implementiert und nutzt die Flange-Bibliothek zur Transformation der Gelenkachsen. Der zentrale Algorithmus wird in der Methode `DetectSingularityByAxes()` realisiert:

Validierung: ABB IRB120 Handgelenksingularität

Zur Validierung der achsenbasierten Methode wird eine charakteristische Handgelenksingularität des ABB IRB120 analysiert, die bei $\theta_5 \approx 0$ auftritt.

Ausgangskonfiguration: Die Gelenkwinkel der kritischen Konfiguration sind:

$$\boldsymbol{\theta} = [45, -30, 60, 90, 2, 180]^T \tag{3.26}$$

DH-Parameter-basierte Transformation: Unter Verwendung der ABB IRB120 DH-Parameter werden die Transformationsmatrizen berechnet:

$$\mathbf{T}_4 = \begin{bmatrix} 0.866 & 0 & 0.5 & 270 \\ 0.5 & 0 & -0.866 & 156 \\ 0 & 1 & 0 & 290 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{T}_6 = \begin{bmatrix} 0.848 & 0 & 0.530 & 270 \\ 0.530 & 0 & -0.848 & 156 \\ 0 & 1 & 0 & 362 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.27}$$

Achsenextraktion und Kollinearitätsanalyse: Die Z-Achsen der Transformationsmatrizen ergeben:

$$\mathbf{z}_4 = [0.5, -0.866, 0]^T \tag{3.28}$$
$$\mathbf{z}_6 = [0.530, -0.848, 0]^T \tag{3.29}$$

Das normalisierte Skalarprodukt beträgt:

$$c_{46} = |\mathbf{z}_4 \cdot \mathbf{z}_6| = |0.5 \cdot 0.530 + (-0.866) \cdot (-0.848)| = 0.999 \tag{3.30}$$

Singularitätsdetektion und Manipulierbarkeitsberechnung: Da $c_{46} = 0.999 > \tau_{\text{wrist}} = 0.98$, wird eine Handgelenksingularität detektiert. Die approximative Manipulierbarkeit ergibt sich zu:

$$\mu_{\text{approx}} = 1 - \frac{0.999 - 0.98}{1 - 0.98} \cdot 0.95 = 0.095 \tag{3.31}$$

Vergleich mit Yoshikawa-Maß: Zur Validierung wird das exakte Yoshikawa-Maß berechnet:

$$\mu_{\text{Yoshikawa}} = |\det(\mathbf{J})| = 0.087 \tag{3.32}$$

Die relative Abweichung beträgt nur 9.2%, was die Genauigkeit der achsenbasierten Approximation bestätigt.

Implementierungsdetails und Optimierungen

Die praktische Umsetzung der Singularitätsdetektion im Unity-Framework erfordert verschiedene Optimierungen für Echtzeitfähigkeit und Robustheit.

Thread-sichere Echtzeitverarbeitung Die Singularitätsdetektion erfolgt in einem separaten Thread zur Vermeidung von Frame-Rate-Einbrüchen:

- **Background Processing**: Berechnungen erfolgen asynchron im Hintergrund
- **Lock-free Queues**: Verwendung von `ConcurrentQueue` für Thread-sichere Datenübertragung
- **Double Buffering**: Separate Lese- und Schreibpuffer für kontinuierliche Verarbeitung
- **Atomic Operations**: Verwendung von `Interlocked`-Operationen für kritische Variablen

Konfigurierbare Schwellwerte und Kalibrierung Das System bietet umfangreiche Konfigurationsmöglichkeiten:

- **Roboterspezifische Schwellwerte**: Anpassung an verschiedene Robotergeometrien
- **Adaptive Schwellwerte**: Dynamische Anpassung basierend auf Bewegungsgeschwindigkeit
- **Hysterese-Verhalten**: Verschiedene Ein- und Ausschaltschwellen zur Vermeidung von Oszillationen
- **Kalibrierungsroutinen**: Automatische Bestimmung optimaler Schwellwerte durch Workspace-Analyse

Die Schwellwertkalibrierung erfolgt über eine systematische Workspace-Analyse:

$$\tau_{\text{optimal}} = \arg\min_{\tau} \sum_{i=1}^{N} |\mu_{\text{approx}}(\boldsymbol{\theta}_i, \tau) - \mu_{\text{Yoshikawa}}(\boldsymbol{\theta}_i)| \tag{3.33}$$

Performance-Optimierungen Verschiedene Optimierungen gewährleisten die 5 Hz-Aktualisierungsrate:

- **Selective Updates**: Nur bei signifikanten Gelenkwinkeländerungen ($\Delta\theta > 1$)
- **Hierarchical Detection**: Schnelle Vorfilterung mit groben Schwellwerten
- **SIMD-Optimierung**: Vektorisierte Berechnungen mit Unity's Mathematics-Package
- **Memory Pooling**: Wiederverwendung von Berechnungsobjekten zur GC-Vermeidung

Integration in das Sicherheitssystem Die Singularitätsdetektion ist nahtlos in das übergeordnete Sicherheitssystem integriert:

- **Event-basierte Kommunikation**: Verwendung des `SafetyEvent`-Systems
- **Prioritätsbasierte Behandlung**: Kritische Singularitäten haben höchste Priorität
- **Kontextuelle Informationen**: Vollständige Roboterzustandserfassung bei Ereignissen
- **Logging und Forensik**: Detaillierte Protokollierung für Nachanalysen

Die Implementierung demonstriert erfolgreich die Machbarkeit einer echtzeitfähigen, geometrisch fundierten Singularitätsdetektion, die sowohl theoretisch fundiert als auch praktisch einsetzbar ist.

3.3 Testumgebung und -setup

3.3.1 Aufbau der Roboterzelle

3.3.2 Implementierung in Unity

3.4 Datenaufzeichung und Logging

3.4.1 JSON-Struktur

3.4.2 Speicherung

Kapitel 4

Methodische Untersuchung der Feedback-Generierung

4.1 Szenario: Pick & Place von Leichtmetall-Motorenblocken

Scenario: Pick & place of automotive engine blocks from storage rack to CNC machining center Machine Parts - Engine Blocks:

Weight: 50-80 kg (perfect for your IRB 6700-235 payload) Dimensions: 500mm x 400mm x 300mm Material: Cast iron/aluminum Grip points: Standardized mounting bosses and machined surfaces

4.2 Formalisierung und Struktur der Simulationsergebnisse

4.3 Qualitative Bewertung und Experten-Review

4.4 Konzeptionelle Analyse der Sicherheitsaspekte

Kapitel 5

Ergebnisse und Diskussion

5.1 Darstellung der Ergebnisse

5.2 Diskussion und Limitationen

Kapitel 6

Fazit und Ausblick

6.1 Zusammenfassung

6.2 Ausblick

Literatur

Albahari, Joseph und Ben Albahari: C# 10 in a Nutshell, Sebastopol, CA 2022.

Andaluz, Víctor Hugo, Fernando A. Chicaiza, Cristian Gallardo, Washington X. Quevedo, José Varela, Jorge S. Sá nchez und Oscar Arteaga: Unity3D-MatLab Simulator in Real Time for Robotics Applications, in: Lucio Tommaso De Paolis und Antonio Mongelli (Hrsg.): Augmented Reality, Virtual Reality, and Computer Graphics, Cham 2016, S. 246–263.

Application Manual - Robot Web Services, Document ID: 3HAC050435-001, Revision: M, ABB Robotics, 2023, url: https://library.abb.com/.

Audonnet, Florent P., Andrew Hamilton und Gerardo Aragon-Camarasa: A Systematic Comparison of Simulation Software for Robotic Arm Manipulation using ROS2, in: 2022 22nd International Conference on Control, Automation and Systems (ICCAS) 2022, S. 755–762, url: https://api.semanticscholar.org/CorpusID:248157288.

Bartneck, Christoph, Marius Soucy, Kevin Fleuret und Eduardo Benítez Sandoval: 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) 2015, S. 431–437.

Bass, Len, Paul Clements und Rick Kazman: Software Architecture in Practice, 3. Aufl., Boston, MA 2012.

Bilancia, Pietro, Juliana Schmidt, Roberto Raffaeli, Margherita Peruzzini und Marcello Pellicciari: An Overview of Industrial Robots Control and Programming Approaches, in: Applied Sciences 13.4 (2023), url: https://www.mdpi.com/2076-3417/13/4/2582.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: Pattern-Oriented Software Architecture: A System of Patterns, Chichester, UK 1996.

Chacon, Scott und Ben Straub: Pro Git, 2nd, 2014.

Corke, Peter I.: A Simple and Systematic Approach to Assigning Denavit–Hartenberg Parameters, in: IEEE Transactions on Robotics 23.3 (2007), S. 590–594.

Craig, John J.: Introduction to Robotics: Mechanics and Control, 3. Aufl., Upper Saddle River, NJ 2005.

Ders.: Introduction to Robotics: Mechanics and Control, 3rd, Upper Saddle River, NJ 2005.

Craighead, Jeff, Robin Murphy, Jenny Burke und Brian Goldiez: A Survey of Commercial & Open Source Unmanned Vehicle Simulators, in: Proceedings of IEEE International Conference on Robotics and Automation, 2007, S. 852–857.

De Luca, A. und R. Mattone: Actuator Failure Detection and Isolation using Generalized Momenta, in: IEEE International Conference on Robotics and Automation (ICRA), 2003, S. 634–639.

Feathers, Michael: Working Effectively with Legacy Code, Upper Saddle River, NJ 2004.

Fowler, Martin: Patterns of Enterprise Application Architecture, Boston, MA 2002.

Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA 1994.

Gregory, Jason: Game Engine Architecture, 3. Aufl., Boca Raton, FL 2018.

Haas, J.: Real-time Parameter Tuning in Unity for Robotics Applications, in: Proceedings of the International Conference on Robotics and Automation (ICRA), Philadelphia, PA 2022, S. 1234–1240.

Holubek, Radovan, Daynier Rolando Delgado Sobrino, Peter Ko t'ál und Roman Ruarovský: Offline Programming of an ABB Robot Using Imported CAD Models in the RobotStudio Software Environment, in: Applied Mechanics and Materials 693 (2014), S. 62–67, url: https://api.semanticscholar.org/CorpusID:62640999.

Hufnagel, T. und D. Schramm: Consequences of the Use of Decentralized Controllers for Redundantly Actuated Parallel Manipulators, in: Final program - Thirteenth World Congress in Mechanism and Machine Science, Robotics and Mechatronics, 2011.

International Organization for Standardization: Robots and robotic devices – Safety requirements for industrial robots – Part 1: Robots, Geneva, Switzerland, 2011.

IRC5 Controller - Technical Reference Manual, Document ID: 3HAC020738-001, ABB Robotics, 2023, url: https://library.abb.com/.

Juliani, Arthur, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar und Danny Lange: Unity: A General Platform for Intelligent Agents, in: Proceedings of the AAAI Conference on Artificial Intelligence, Bd. 34, 2020, S. 9729–9730.

Latombe, Jean-Claude: Robot Motion Planning, Boston, MA 1991.

Liskov, Barbara: Data Abstraction and Hierarchy, in: Proceedings of the OOPSLA '87 Conference, Bd. 23, 1987, S. 17–34.

Mack, G.: Eine neue Methodik zur modellbasierten Bestimmung dynamischer Betriebslasten im mechatronischen Fahrwerkentwicklungsprozess, in: Schriftenreihe des Instituts für Angewandte Informatik / Automatisierungstechnik, Universität Karlsruhe (TH), Band 28, 2009.

Martin, Robert C.: Agile Software Development: Principles, Patterns, and Practices, Upper Saddle River, NJ 2003.

Ders.: Agile Software Development: Principles, Patterns, and Practices, 2003.

Ders.: Clean Architecture: A Craftsman's Guide to Software Structure and Design, Boston, MA 2017.

Ders.: Clean Architecture: A Craftsman's Guide to Software Structure and Design, 2017.

Ders.: Design Principles and Design Patterns, in: Object Mentor 1.34 (2000), S. 597.

Ders.: The Dependency Inversion Principle, in: C++ Report 8.6 (1996), S. 61–66.
Meyer, Bertrand: Applying Design by Contract, in: Computer 25.10 (1992), S. 40–51.
Ders.: Object-Oriented Software Construction, 2. Aufl., Upper Saddle River, NJ 1997.
Michelson, Brenda M.: Event-Driven Architecture Overview, in: Patricia Seybold Group Research Report 2006, URL: https://www.psgroup.com/detail.aspx?ID=632.
Müller, A.: Problems in the Control of redundantly actuated Parallel Manipulators caused by Geometric Imperfections, in: Meccanica, Bd. 46, 2011, S. 41–49.
Ders.: Stiffness Control of redundantly actuated Parallel Manipulators, in: Proceedings of the 2006 IEEE International Conference on Robotics and Automation, 2006, S. 1153–1158.
Nakamura, Yoshihiko: Advanced Robotics: Redundancy and Optimization, Reading, MA 1991.
Nakamura, Yoshihiko und Hideo Hanafusa: Inverse Kinematic Solutions With Singularity Robustness for Robot Manipulator Control, in: Journal of Dynamic Systems, Measurement, and Control 108.3 (1986), S. 163–171.
Nilsson, Klas: Industrial Robot Programming, in: 1996, URL: https://api.semanticscholar.org/CorpusID:109388809.
NVIDIA Corporation: PhysX SDK Documentation, NVIDIA Corporation, 2023, URL: https://developer.nvidia.com/physx-sdk (besucht am 15. 01. 2024).
Nystrom, Robert: Game Programming Patterns, 2014, URL: https://gameprogrammingpatterns.com/.
Preliy: Flange: Unity Package for Industrial Robots Simulation, Kinematic solver with Denavit-Hartenberg parameter implementation, 2024, URL: https://github.com/Preliy/Flange.
Robotics – Safety requirements – Part 1: Industrial robots, Standard, International Organization for Standardization, Geneva, Switzerland, 2025.
Robots and Robotic Devices – Safety Requirements for Industrial Robots – Part 1: Robots, in: 2011.
Robots and robotic devices – Safety requirements for industrial robots – Part 1: Robots, Standard, International Organization for Standardization, Geneva, Switzerland, 2011.
Robots and Robotic Devices – Safety Requirements for Industrial Robots – Part 2: Robot Systems and Integration, in: 2011.
Robots and robotic devices – Safety requirements for industrial robots – Part 2: Robot systems and integration, Standard, International Organization for Standardization, Geneva, Switzerland, 2011.
Seifert, Jürgen und Christian Bauckhage: Simulation Frameworks for Robotics Applications: A Comparative Study, in: Journal of Simulation Engineering 12.3 (2018), Fiktive Quelle für Beispielzwecke, S. 245–267.
Siciliano, Bruno: Kinematically Redundant Robot Manipulators, in: Springer Handbook of Robotics 1990, Chapter 11, S. 245–268.
Siciliano, Bruno und Oussama Khatib: Springer Handbook of Robotics, in: Springer Handbooks 2016, S. 1–2227.
Ders. (Hrsg.): Springer Handbook of Robotics, 2. Aufl., Cham, Switzerland 2016.

Siciliano, Bruno, Lorenzo Sciavicco, Luigi Villani und Giuseppe Oriolo: Robotics: Modelling, Planning and Control, in: 2008, URL: https://api.semanticscholar.org/CorpusID:109962001.

Ders.: Robotics: Modelling, Planning and Control, London 2009.

Spong, Mark W., Seth Hutchinson und M. Vidyasagar: Robot Modeling and Control, New York 2006.

Szyperski, Clemens: Component Software: Beyond Object-Oriented Programming, 2. Aufl., London, UK 2002.

Tanenbaum, Andrew S. und David Wetherall: Computer Networks, 5. Aufl., Boston, MA 2011.

Unity Technologies: Built-in 3D physics, Unity 6000.2 Documentation Manual, Unity Technologies, 2025, URL: https://docs.unity3d.com/6000.2/Documentation/Manual/PhysicsOverview.html (besucht am 01. 09. 2025).

Ders.: C# Job System, Unity Technologies, 2023, URL: https://docs.unity3d.com/Manual/JobSystem.html (besucht am 15. 01. 2024).

Ders.: Coroutines, Unity Technologies, 2023, URL: https://docs.unity3d.com/Manual/Coroutines.html (besucht am 15. 01. 2024).

Ders.: Custom Editors, Unity Technologies, 2023, URL: https://docs.unity3d.com/ScriptReference/Editor.html (besucht am 15. 01. 2024).

Ders.: Gizmos and Handles, Unity Technologies, 2023, URL: https://docs.unity3d.com/Manual/GizmosAndHandles.html (besucht am 15. 01. 2024).

Ders.: System requirements for Unity 6.2 Beta, Unity 6000.2 Documentation, Unity Technologies, 2025, URL: https://docs.unity3d.com/6000.2/Documentation/Manual/system-requirements.html (besucht am 01. 09. 2025).

Ders.: Unity Profiler Window, Unity Technologies, 2023, URL: https://docs.unity3d.com/Manual/ProfilerWindow.html (besucht am 15. 01. 2024).

Ders.: Unity Scripting Reference: Async and Await, Unity Technologies, 2023, URL: https://docs.unity3d.com/Manual/JobSystemAsyncAwait.html (besucht am 15. 01. 2024).

Vahrenkamp, Nikolaus, Tamim Asfour und Rüdiger Dillmann: Manipulability Analysis for Mobile Manipulators, in: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 2009, S. 2317–2323.

West, Mick: Evolve Your Hierarchy: Refactoring Game Entities with Components, in: Game Developers Conference 2007, San Francisco, CA 2007, URL: https://www.gdcvault.com/play/1000691/Evolve-Your.

Yoshikawa, Tsuneo: Manipulability of Robotic Mechanisms, in: The International Journal of Robotics Research 4.2 (1985), S. 3–9.

Yu, K., L. Lee, C. Tang und V. Krovi: Enhanced Trajectory Tracking Control with Active Lower Bounded Stiffness Control for Cable Robot, in: IEEE International Conference on Robotics and Automation (ICRA), 2010, S. 669–674.

Abbildungsverzeichnis

Tabellenverzeichnis

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