Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Softwareframework für den humanoiden Roboter Gretchen

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Dominik Treßin
geboren am: 21.07.1999
geboren in: Berlin

Gutachter/innen: Prof. Dr. Verena V. Hafner
Prof. Dr. Holger Schlingloff

eingereicht am: ..... verteidigt am: .....

Dieses Bachelorarbeit beschäftigt sich mit der Erweiterung des humanoiden Roboters Gretchen um ein Softwareframework. Dafür werden verschiedene Frameworks anderer Roboter miteinander verglichen und dabei die Eignung für den Gretchen-Roboter analysiert. Als Grundlage für das neue Gretchen-Framework wird das Framework des Nao-Roboters von Berlin United verwendet. Für die Kommunikation zwischen den verschieden Hard- und Softwarekomponenten des Roboters werden mehrere Serialisierungsformate verglichen und es wird eine Empfehlung für Gretchen abgegeben. Das Simulationsmodell in Webots von Gretchen wird als neue Plattform in das Berlin United Framework integriert. Dieses wird um einen Threading-Mechanismus erweitert, der die parallele Ausführung von Modulen ermöglicht. Dafür wird eine Analyse des aktuellen Cognition-Prozesses des Nao-Roboters von Berlin United bezüglich der Abhängigkeiten der Module untereinander durchgeführt. Dabei werden einige Fehler festgestellt und es werden Empfehlungen für zukünftige Verbesserungen abgegeben.

Inhaltsverzeichnis

1 Einleitung

Humanoide Roboter sind seit vielen Jahren ein wachsender Bestandteil der Roboterforschung. Die Entwicklung dieser Roboter schreitet dabei in großen Schritten voran. Beispielsweise gibt es den Roboter Pepper [39] von SoftBank Robotics, der in der Lage ist die Mimik und Gestik von Menschen zu analysieren und auf menschliche Emotionen zu reagieren. So wird er schon in Restaurants für die Bedienung und Unterhaltung von Gästen eingesetzt [40]. Ein anderer humanoider Roboter ist Atlas [8] von Boston Dynamics. Dieser kann komplexe dynamische Bewegungen wie Sprünge und Saltos ausführen. Ein weiteres Beispiel für die großen Fortschritte in der Roboterforschung ist der Digit Roboter [21] von Agility Robotics [35], der in der Lage ist in komplexen Umgebungen zu navigieren. In Zukunft soll er für die Auslieferung von Paketen verwendet werden. Nicht vergessen sollte man den Roboter Nao [38], der seit vielen Jahren beim RoboCup [9] eingesetzt wird. Bei diesem Wettbewerb spielen mehrere Nao-Roboter vollständig autonom Fußball und treffen selbständig Entscheidungen. Hinzu kommt, dass von Jahr zu Jahr die Spielregeln immer weiter verschärft werden. Weil das Interesse an diesen Robotern immer weiter steigt, werden die Roboter auch zunehmend in der Lehre eingesetzt. Es ist wichtig, dass immer mehr Menschen Zugang zu diesem Feld finden, da dieses eine Technologie ist, die unser Leben verändern könnte. Die Zugänglichkeit wird erschwert, da diese Roboter im Allgemeinen teuer und sehr komplex in der Anwendung sind.

Dieses Bachelorarbeit beschäftigt sich mit einem Softwareframework für den humanoiden Roboter Gretchen. Gretchen ist in einem Prototypstadium und soll kostengünstig und leicht zugänglich sein, um ihn in Forschung und Lehre einsetzen zu können. Für diesen Einsatzbereich soll der Roboter verschiedene Anforderungen erfüllen. Einerseits soll es möglich sein, dass ein Student kleinere Experimente innerhalb einer Lehrveranstaltung durchführen kann und andererseits sollte der Roboter genügend Freiraum für Forschungsmöglichkeiten bieten. Das stellt besondere Anforderungen an die Struktur eines Programms. Der Einsatz in der Lehre bedeutet, dass Studenten mit unterschiedlichen Programmierkenntnissen mit dem Roboter arbeiten sollen. Gleichzeitig erfordert der Forschungseinsatz die Durchführung verschiedener komplexer Experimente. Es ist in Praxis so, dass mehrere Wissenschaftler an verschiedenen Bereichen eines Roboters arbeiten. So beschäftigt sich beispielsweise einer mit der Entwicklung neuer Algorithmen für die Bildverarbeitung und ein anderer mit verschiedenen Laufmodellen. Das heißt, es wird eine Struktur benötigt, die es ermöglicht, vielen Wissenschaftlern auf verschiedenen Ebenen zusammen zu arbeiten. Ziel ist es dabei, die Komplexität so zu abstrahieren, dass der Fokus auf der Lösung einzelner Probleme liegt. Die Struktur sollte es den Wissenschaftlern ermöglichen, ihre Algorithmen so umzusetzen, dass sie zusammenarbeiten können, ohne dass sie sich dabei in die Quere kommen. Genau so eine Struktur wird im Allgemeinen Softwareframework genannt.

Bei der Umsetzung eines Softwareframeworks für Gretchen gibt es mehrere Anforderungen zu erfüllen. Der Roboter besteht aus vielen heterogenen Komponenten, wie Sensoren, Motoren und auch verschiedene Recheneinheiten. Diese Komponenten sollen zusammenarbeiten. Daher wird ein Softwareframework benötigt, welches die Aufgaben

hat, Sensoren auszulesen und Motoren anzusteuern. Ein weiterer wichtiger Fakt ist, dass der Roboter keinen Monitor besitzt. Deshalb sollte es möglich sein, den Roboter per Remote über einen externen Rechner zu debuggen. Optional sollte das Hauptprogramm auch auf einem anderen Rechner ausgeführt werden. Diesen verschiedenen technischen Herausforderungen dienen dieser Arbeit. Als Grundlage werden dabei verschiedene Frameworks recherchiert und miteinander verglichen. Ein Roboterframework ist ein sehr komplexes Thema und kann deshalb in dieser Arbeit nicht komplett behandelt werden. Daher wird auf einzelne Aspekte genauer eingegangen, insbesondere werden die Themen Threading und Serialisierung genauer betrachtet.

1.1 Struktur der Arbeit

Als Erstes beschäftigt sich der Abschnitt 1.3 mit der Entstehung und dem aktuellen Entwicklungsstand des Gretchen-Roboters. Danach werden im Abschnitt 1.4 die Anforderungen an das zu entstehende Framework betrachtet. Im Abschnitt 2 werden verschieden Softwarearchitekturen (Abschnitt 2.1) und Kommunikationsformate (Abschnitt 2.2), die in Robotern verwendet werden oder verwendet werden können, untersucht und miteinander verglichen. Dabei wird die Eignung für Gretchen analysiert und es werden Empfehlungen für diesen Roboter abgegeben. Der Abschnitt 3 beschäftigt sich mit der praktischen Umsetzung. Diese unterteilt sich in zwei Unterabschnitte. Der Abschnitt 3.1 beschäftigt sich mit der Integration der Webots-Simulation von Gretchen in das Softwareframework von Berlin United. Im Abschnitt 3.2 wird das Berlin United Framework um einen Threading-Mechanismus erweitert. Dafür werden als Grundlage die verschiedenen Abhängigkeiten, die bei einer Parallelisierung beachtet werden müssen, untersucht. Der letzte Abschnitt 4 fasst die Ergebnisse dieser Arbeit zusammen und schlägt Möglichkeiten für weitere zukünftige Projekte vor.

1.2 Beiträge der Arbeit

Im Rahmen dieser Arbeit wurden der aktuelle Entwicklungszustand von Gretchen zusammengefasst und es wurden Anforderungen an ein Softwareframework für diesen Roboter diskutiert. Darauf basierend, wurden verschiedene Roboterframeworks anderer Roboter bezüglich ihrer Anwendbarkeit für Gretchen analysiert und miteinander verglichen. Dabei wurde das Nao-Framework von Berlin United für Gretchen ausgewählt und es wurde entschieden, dieses Framework um den Threading-Mechanismus von Nao Devils zu erweitern. Ebenfalls wurden verschiedene Serialisierungsformate recherchiert. Für diese wurden Empfehlungen für die Anwendbarkeit in der Kommunikation zwischen Hard- und Softwarekomponenten des Roboters abgegeben. Praktisch wurde im Rahmen dieser Arbeit ein vorhandenes Webots-Modell von Gretchen als neue Plattform für das Berlin United Framework integriert. Für diese Plattform wurden zur Test- und Demonstrationszwecken zwei Motion-Module und ein Cognition-Modul implementiert. Diese enthalten einen Lauf- und einen Kniebeugenalgorithmus. Des Weiteren wurde für das Berlin United Framework ein Threading-Mechanismus implementiert, der es ermöglicht, Module parallel in einer Threadpool-Bibliothek auszuführen. Dafür wurde

die Bibliothek Taskflow recherchiert, analysiert und verwendet. Zusätzlich wurden für den Cognition-Prozess von Berlin United die Parallelisierungsmöglichkeiten untersucht. Im letzten Teil werden weitere, auf diese Arbeit aufbauende Projekte, vorgeschlagen.

1.3 Projekt Gretchen

Gretchen [17] ist ein humanoider Roboter, der in einem Open Platform Projekt, an dem verschieden Teilnehmer und Teilnehmergruppen mitgewirkt haben, entstanden ist. Initiiert wurde das Projekt von der AI Brain Company [22]. Der Roboter wurde hauptsächlich von Matthias Kubisch entwickelt. Eine ausführliche Dokumentation und Testung des Roboters wurde im Rahmen der Bachelorarbeit von Anastasia Prisacaru [33] an der Humboldt-Universität zu Berlin durchgeführt. Zusätzlich erfolgten in einem Seminar [2] und einem Semesterprojekt [50] zu dem Thema „Humanoide Roboter“ mehrere Weiterentwicklungen. Eine Projektbeschreibung mit einer Timeline befindet sich hier [29] und eine ausführliche Dokumentation hier [18].

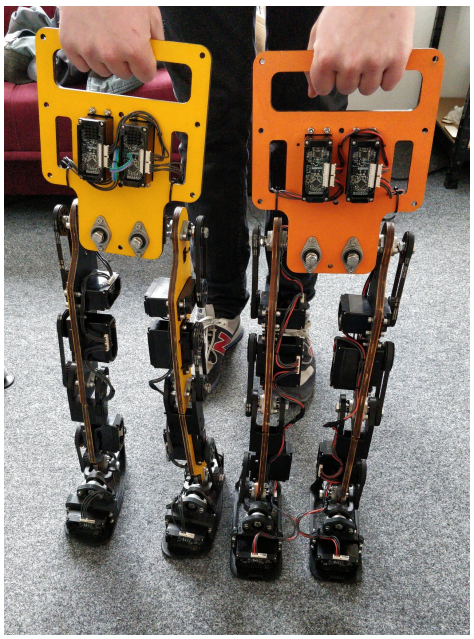

Abbildung 1: Gretchen-Roboter entnommen aus [17].

Im aktuellen Entwicklungszustand besteht der Roboter Gretchen aus einem aus Holz gebauten Unterkörper mit zwei Beinen und zehn Gelenken. Diese Gelenke werden mithilfe von zehn Servomotoren über Zahnriemen angesteuert. Die Sensorimotor-Servomotoren besitzen eine Schnittstelle für einen RS-485 Bus. Alle Motoren sind über diesen Bus miteinander verbunden. Mithilfe eines Protokolls können über den Bus Motorpositionen gesetzt und Sensordaten empfangen werden. Darauf aufbauend hat Anastasia Prisacaru in ihrer Bachelorarbeit mehrere Konzepte für eine Low-Level Softwarearchitektur entwickelt (siehe Abb. 2). An der Realisierung eines dieser Konzepte wurde in dem oben aufgeführten Semesterprojekt gearbeitet. Für die Steuerung der

Kommunikation des RS-485 Bus ist ein Mikrocontroller zuständig. Prisacaru hat in ihrer Arbeit dafür einen ESP32 Mikrocontroller vorgesehen. In dem Semesterprojekt wurden aber ein STM32 Feather verwendet. Für diesen wurde ein extra Featherwing mit der passenden RS-485 Schnittstelle entwickelt. Außerdem wurde an dem Feather eine IMU[1] angeschlossen. Ein weiterer Bestandteil der Low-Level Architektur ist ein Raspberry Pi, welcher als Hauptrechner für den Roboter agieren soll. Dieser soll die Wahrnehmung verarbeiten und den Roboter ansteuern. Dieses zu strukturieren, ist die Aufgabe der High-Level Softwarearchitektur. Damit beschäftigt sich diese Arbeit.

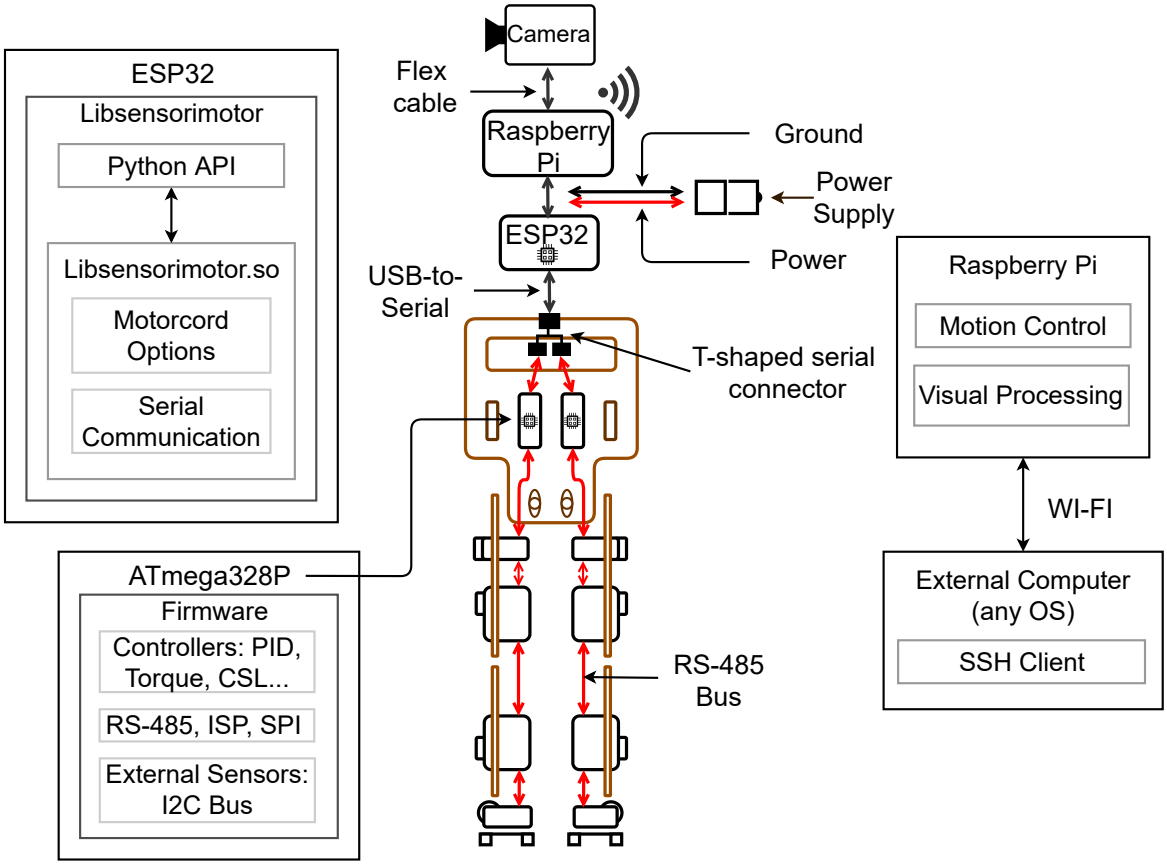

Abbildung 2: Geplante Low-Level Softwarearchitektur entnommen aus [33].

Zusätzlich wurden für den Roboter verschiedene Erweiterungen erarbeitet. Unter anderem wurde ein schwenkbares Kameraauge entwickelt. Dieses besteht aus einer Kamera, die mit zwei Servomotoren um zwei Achsen gedreht werden kann. Auch gibt es einen ersten Prototyp für einen neue Hüfte. Diese besitzt einen zusätzlichen Freiheitsgrad und ist daher um eine zusätzliche Achse beweglich. Mit der alten Hüfte ist Gretchen nur in der Lage, sich vorwärts und seitwärts zu bewegen. Die neue Hüfte soll es ermöglichen, dass der Roboter sich auch um seine eigene Achse drehen kann. Außerdem wurde ein Simulationsmodell des Roboters für Webots erstellt. Für dieses konnten schon erste Laufmodelle entwickelt werden.

[1]Inertial Measurement Unit

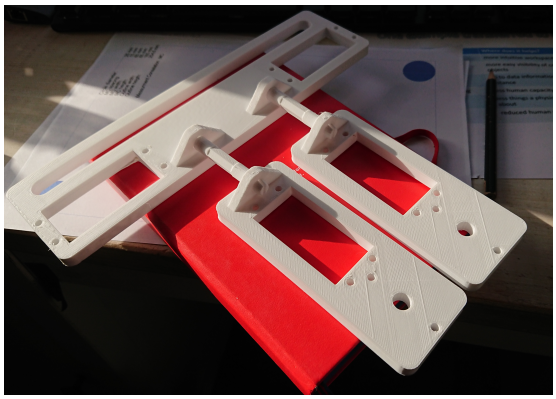
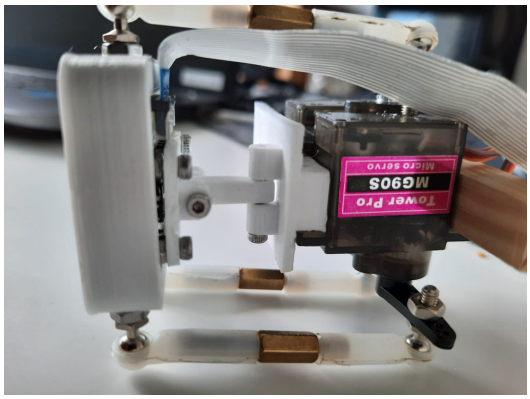

Abbildung 3: Prototypen des neuen Hüftgelenks (links) und des schwenkbares Kameraauges (rechts) entnommen aus [50].

1.4 Design Prinzipien

Bei der Entwicklung eines Roboterframeworks gibt es bestimmte Ziele, die abhängig von späteren Anwendungsbereichen des Roboters sind. Für Gretchen ist der Anwendungsbereich als Forschungs- und Lehrroboter vorgesehen. In diesem Bereich gibt es ein paar wichtige Aspekte, die weit verbreitet sind und die bei der Softwareentwicklung berücksichtigt werden sollten:

- **Verschiedenste Nutzer:** Der Roboter ist ein Universitätsprojekt bei dem häufig wechselnde Personen mitarbeiten. Diese Personen haben unterschiedliche Vorkenntnisse und damit auch unterschiedliche Anforderungen an ein Framework. Einerseits gibt es beispielsweise Studierende, die wenig Erfahrung mit dem Umgang mit Robotern besitzen und die kleinere Experimente innerhalb einer Lehrveranstaltung in einem Semester mit dem Roboter experimentieren wollen. Andererseits gibt Personen, die den Roboter für Forschungsprojekte verwenden und komplexe Experimente durchführen wollen.

- **Unterstützung der gängigsten Betriebssysteme**: Der Software sollte möglichst die gängigsten Betriebssysteme (Windows, Linux, macOS) unterstützen. Alle Nutzer sollten in der Lage sein, unabhängig von dem verwendeten System Software für den Roboter zu entwickeln und Simulationen auszuführen.

- **Modularität:** Der Roboter wird für viele Experimente verwendet und wird dabei stetig weiterentwickelt. Das bedeutet auch, dass sich die Software fortwährend ändert. Dabei können mehrere Lösungen von verschiedenen Personen parallel implementiert und ausgeführt werden. Diese dürfen sich dabei nicht gegenseitig behindern. Daher sollte das Framework eine Möglichkeit bieten, die paralleles Arbeiten unterstützt. Dafür ist es notwendig, dass Codeteile isoliert werden können. Für dieses Problem ist Modularität eine weit verbreitete Lösung. Die Anpassung von Software kann dabei durch die Entwicklung neuer Module implementiert werden.

- **Unterstützung verschiedener Roboterplattformen:** Die Software soll sowohl den Roboter als auch die Simulationen unterstützen. Dadurch können Experimente unabhängig von der verwendeten Plattform implementiert werden und vor der Ausführung auf dem Roboter simuliert werden.
- **Einfache Bedienung:** Da der Roboter als Forschungs- und Lehrroboter dient, haben die Nutzer unterschiedliche Erfahrungen und Fähigkeiten. Daher sollte die Einarbeitung und der Umgang möglichst einfach sein, damit auch Personen mit Programmiergrundkenntnissen in der Lage sind, Algorithmen für den Roboter zu implementieren.
- **Echtzeitfähig:** Der Roboter muss in der Lage sein, möglichst schnell auf Ereignisse zu reagieren und bestimmte Deadlines einzuhalten.

2 Überblick existierender Software für Roboter

Bevor mit der Entwicklung eines Softwareframeworks für Gretchen begonnen wird, empfiehlt es sich einen Überblick über existierende Software für Roboter zu verschaffen. Diese lässt sich in zwei Arten unterteilen. Einerseits gibt es speziell für den Einsatz im Roboter entwickelt Software und andererseits Software, die sich sowohl für den Einsatz im Roboter als auch für andere Anwendungsbereiche eignet. Diese beiden Arten werden in den folgenden Unterabschnitten separat betrachtet. Im ersten Unterabschnitt werden Softwarearchitekturen anderer humanoider Roboter untersucht. Diese werden untereinander verglichen und auf Eignung im Gretchen-Roboter überprüft. Im zweiten Unterabschnitt werden Softwarelösungen für die Kommunikation im Roboter betrachtet. Um Nachrichten zwischen den verschiedenen Hard- und Softwarekomponenten auszutauschen, ist es notwendig, dass diese serialisiert werden. Daher werden verschieden Bibliotheken für die Serialisierung von Nachrichten untersucht und miteinander verglichen.

2.1 Softwarearchitektur für humanoide Roboter

Bei der Entwicklung einer neuen Software ist es sinnvoll, einen Blick auf ähnliche existierende Software zu werfen. Diese können Anregungen und Lösungen für die Umsetzung der neuen Software liefern. Für die Entwicklung eines Softwareframeworks für Gretchen bietet sich eine Betrachtung von Softwarearchitekturen anderer humanoider Roboter an. Diese werden daher nachfolgend untersucht und jeweils auf Eignung für Gretchen geprüft. Insbesondere werden dabei die Frameworks des Nao-Roboters verschiedener Universitäten betrachtet. Diese haben einen ähnlichen Anwendungsbereich und damit ähnliche Anforderungen wie der Gretchen-Roboter.

2.1.1 Berlin United

Berlin United – Nao Team Humboldt [46] ist eine Forschungsgruppe der Humboldt-Universität zu Berlin, die regelmäßig an RoboCup Wettbewerben mit dem Roboter Nao teilnimmt. Laut dem Team Report von 2018 [30] wurde die Gruppe 2007 unter dem Namen NaoTH gegründet. 2011 bildeten sie ein gemeinsames Team mit FUmanoid von der Freie Universität Berlin als Berlin United. Seit 2017 besteht das Berlin United Team nur noch aus dem NaoTH Team, da FUmanoid aufgehört hat zu existieren.

Die Software des Berlin United Teams wird für Forschungs- und Bildungszwecke verwendet. Daher liegt der Fokus auf eine klare Strukturierung, die eine schnelle Einarbeitung ermöglicht. Die Software besteht aus einem plattformabhängigen und einem plattformunabhängigen Teil, die beide über ein Plattform-Interface miteinander kommunizieren (siehe Abb. 4). Diese Strukturierung ermöglicht es, dass das Framework auf unterschiedlichen Plattformen, wie zum Beispiel auf dem Roboter oder in einer Simulation, ausgeführt werden kann. Der plattformabhängige Teil implementiert für jede Plattform die Schnittstelle zwischen der Plattform und dem Plattform-Interface. Bis jetzt existieren Implementationen für den Nao-Roboter, für einen Logfile-based-

Simulator, für den Webots-Simulator [27] und für den SimSpark-Simulator [43]. Der plattformunabhängige Teil besteht aus mehreren Prozessen, die die Algorithmen implementieren. Die Prozesse laufen unabhängig voneinander in verschieden Threads. Jeder Prozess wird dabei durch das Modul-Framework definiert. Das Modul-Framework basiert auf einer Blackboard-Architektur. Das heißt, ein Prozess besteht aus mehreren Modulen, welche Daten in der Form von Representationen über ein gemeinsames Blackboard austauschen. Dabei besitzt jeder Prozess sein eigenes Blackboard. Bei der Ausführung eines Prozesses werden die einzelnen Module von Modul-Manager sequentiell aufgerufen. Es ist dabei möglich, zur Laufzeit zu konfigurieren, welche Module ausgeführt werden sollen. Aktuell besitzt das Framework zwei Prozesse. Es gibt einen Motion-Prozess, der für die Bewegung des Roboters und einen Cognition-Prozess, der für die Wahrnehmung zuständig ist. Der Datenaustausch zwischen den Prozessen untereinander sowie zwischen den Prozessen und der Plattform erfolgt durch das Plattform-Interface. Dafür werden nach dem Ausführen eines Prozesses die geteilten Representationen mithilfe von Protocol Buffers (siehe Abschnitt 2.2.1) serialisiert und in eine Message-Queue geschrieben. Eine genauere Beschreibung der Softwarearchitektur kann hier [31] und hier [30] gefunden werden.

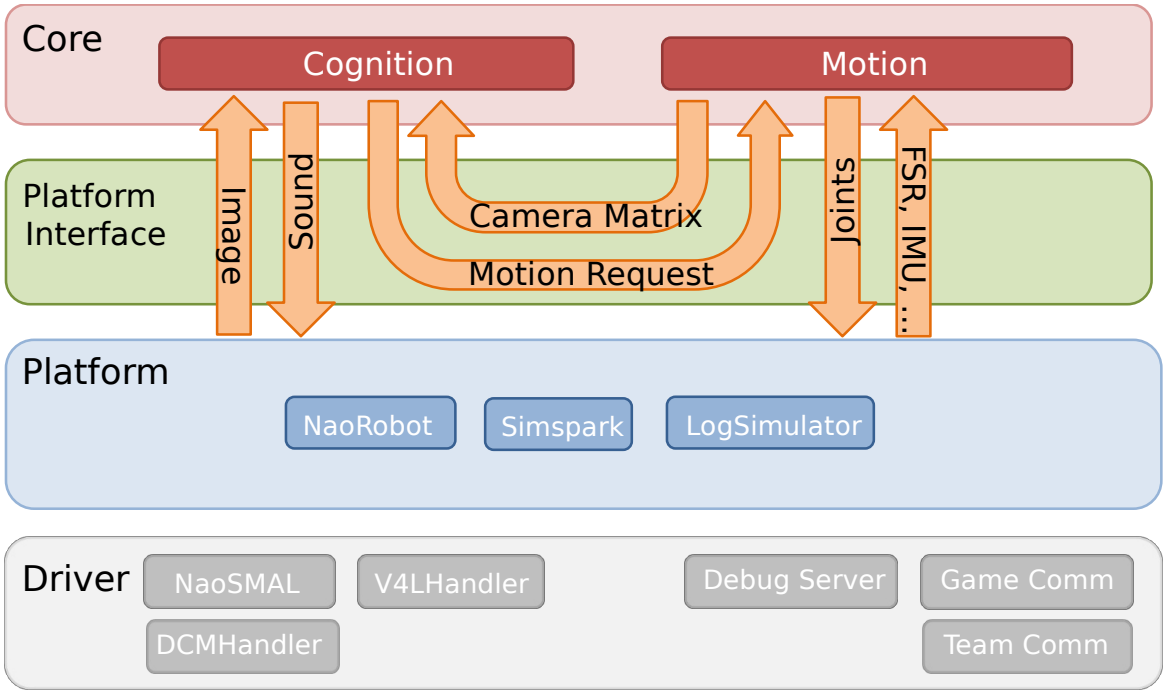

Abbildung 4: Softwarearchitektur von Berlin United entnommen aus [30].

2.1.2 B-Human

B-Human [3] ist ein RoboCup-Team von Studenten und Forschern der Universität Bremen und des Deutschen Forschungszentrums für Künstliche Intelligenz [24]. Sie haben an viele RoboCup Wettbewerben in der RoboCup Standard Platform League

[34] erfolgreich teilgenommen. Unter anderem haben sie neunmal die German Open und einmal die European Open gewonnen. Außerdem wurde sie achtmal Weltmeister.

Das Framework von B-Human wird im Team Report und Code Release von 2019 [41] ausführlich beschrieben. Es verwendet ähnliche Konzepte wie das Framework von Berlin United. Das B-Human Framework besteht aus mehreren Threads (siehe Abb. 5). Die Threads basieren auf einer Blackboard-Architektur. Jeder Thread besteht aus mehreren Modulen, die sequenziell ausgeführt werden. Dabei können die Module auf einen gemeinsamen, geteilten Speicher, das Blackboard, zugreifen und Daten lesen oder schreiben. Die Daten, die ein Blackboard speichert, sind Instanzen von verschiedenen Representationen. Jeder Thread besitzt dabei ein eigenes Blackboard. Für die Kommunikation zwischen Threads werden Representationen, die in mehreren Threads verwendet werden, mittels Inter-Thread-Kommunikation ausgetauscht. Das aktuelle Framework besteht aus insgesamt fünf Threads. Es gibt einen Cognition-Thread, der für die Verhaltenskontrolle und einen Motion-Thread, der für die Bewegungen des Roboters zuständig ist. Dann gibt es die zwei Threads, Lower und Upper, die jeweils die Bilder der zwei Kameras verarbeiten. Zusätzlich gibt es noch eine fünften Debug-Thread, der mithilfe von TCP und IP mit dem Host-PC kommuniziert. Die Aufteilung des ursprünglichen Cognition-Threads in Upper, Lower und Cognition erfolgte 2019 im Rahmen einer Bachelorarbeit von Jan Fiedler [10], um den Quad-Core Prozessor der neuen Nao Version besser ausnutzen zu können.

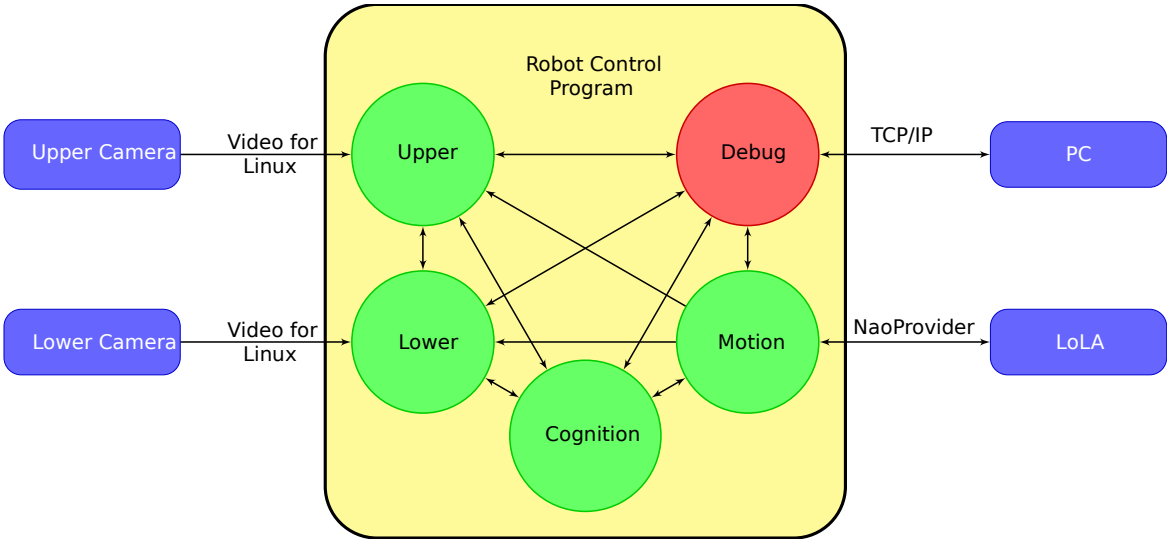

Abbildung 5: Softwarearchitektur von B-Human entnommen aus [41].

2.1.3 Nao Devils

Nao Devils [32] ist die RoboCup-Mannschaft des Instituts für Roboterforschung der Technische Universität Dortmund. Das Framework von Nao Devils basiert auf dem

B-Human Code Release von 2015 [2]. Daher ist die Softwarearchitektur von Nao Devils auch sehr ähnlich mit der B-Human Architektur. Die Änderungen, die von Nao Devils durchgeführt wurden, werden im aktuellen Team Report von 2019 [42] beschrieben. Eine große Änderung, die aber noch nicht im aktuellen Team Report enthalten ist, ist die weitere Parallelisierung des Frameworks. Diese wurde im Rahmen der Masterarbeit von Aaron Larisch [25] umgesetzt. Bei dieser Arbeit wurde zuerst eine Analyse für die Parallelisierungsmöglichkeiten des aktuellen Frameworks durchgeführt. Dabei wurden drei mögliche Ebenen für eine Parallelisierung festgestellt. Eine Möglichkeit ist auf Cycle-Ebene, wobei man die Anzahl der Threads erhöhen kann. Beispielsweise können neben den beiden klassischen Motion- und Cognition-Threads weitere Threads für die Bildverarbeitung hinzugefügt werden. Die zweite Möglichkeit ist eine Parallelisierung auf Modul-Ebene, wobei die Module innerhalb der Threads parallel ausgeführt werden. Die dritte Möglichkeit ist auf Execution-Ebene, dabei wird bei rechenaufwendigen Modulen der Code innerhalb der Module parallelisiert. Für die Umsetzung auf Modul und Execution-Ebene wurden verschiedene Threadpool-Bibliotheken untersucht. Aaron Larisch hat sich für die Umsetzung der Parallelisierung für die Taskflow-Bibliothek entschieden, da diese eine gute Performance, eine hilfreiches Debug-Interface und eine übersichtliche API aufweist. Taskflow bietet die Möglichkeit den aktuellen Task Graphen auszugeben. Außerdem gibt es ein gute graphische Darstellung der Ausnutzung der einzelnen Threads über die Zeit. Da der Motion-Thread gegenüber dem Cognition-Thread priorisiert werden soll, es aber keine Möglichkeit gibt, Tasks in Taskflow zu priorisieren, wurden für die beiden Threads jeweils ein eigener Taskflow-Threadpool implementiert. Für die Integration in das Framework wurde aus dem Modul-Representation Graphen des Frameworks ein Task Graphen von Taskflow erstellt. Außerdem wurde für die Module eine Schnittstelle implementiert, die es ermöglicht Subgraphen anzulegen. Diese realisiert die Parallelisierung auf Execution-Ebene. Abschließend hat Aaron Larisch in seiner Arbeit die optimale Anzahl an Threads pro Threadpool experimentell ermittelt. Dabei hat sich eine Kombination aus zwei Thread für den Motion-Prozess und vier Threads für den Cognition-Prozess als optimal herausgestellt.

2.1.4 Robot Operating System

Robot Operating System (ROS) [37] ist ein Open Source Meta-Betriebssystem für Roboter. Es wurde für den RP2 Roboter von Willow Garage und für Roboter, die diesem ähnlich sind entwickelt. Laut Website [37] beinhaltet das Framework eine Sammlung von Werkzeugen, Bibliotheken und Konventionen, die es vereinfachen ein komplexes Roboterverhalten zu entwickeln. Das Hauptziel von ROS ist die Wiederverwendung von Code in der Roboterforschung und -entwicklung [36].

Das ROS Framework ist ausführlich in der Dokumentation [36] beschrieben. Es Implementierungen in Python, C++, Lisp, Java und Lua. Bis jetzt werden von ROS nur unixartige Betriebssysteme unterstützt. Die Verwendung unter Windows befindet

[2] `https://github.com/bhuman/BHumanCodeRelease/tree/coderelease2015`, besucht am 12.11.2021

sich noch im experimentellen Stadium. Außerdem ist das Framework nicht echtzeitfähig. Es gibt aber die Möglichkeit ROS mit echtzeitfähigen Komponenten zu verwenden.

Das Framework ist eine servicebasierte Peer-To-Peer Architektur, die aus einem Netzwerk aus mehreren Prozessen besteht. Die Prozesse werden Nodes genannt und tauschen Informationen über eine Kommunikationsinfrastruktur aus. Die Verbindungsinformationen zu anderen Nodes erhalten sie nach Registrierung beim Master. Für die Kommunikation gibt es drei Möglichkeiten. Eine Möglichkeit ist die synchrone Kommunikation mithilfe von Services, die von einem Node angeboten werden. Ein anderer Node kann eine Anfrage zu einem Service senden. Danach muss dieser auf die Antwort des anbietenden Nodes warten. Die zweite Möglichkeit ist die asynchrone Kommunikation mithilfe von Topics. Hier erfolgt der Austausch von Daten mit dem Mechanismus des Publish-Subscribe-Patterns. Dabei können die Nodes Daten zu einem Thema abonnieren. Wenn dann ein Node zu einem Thema Daten veröffentlicht, erhalten die abonnierenden Nodes diese Daten. Eine Abstimmung untereinander und ein Warten aufeinander sind daher nicht notwendig. Die dritte Option für die Kommunikation ist der Datenaustausch über den Parameter-Server. Dieser ist ein Teil des Masters und speichert Schlüssel-Wert-Paare [36].

Neben ROS 1 gibt es noch ROS 2 [28]. Diese neue Version wurde für neue Anwendungszwecke von ROS entwickelt. ROS wird nicht mehr nur in der Forschung, sondern auch in der Industrie und in der Raumfahrt verwendet. Bei ROS 2 werden neue Technologien für die Definition, die Serialisierung und den Transport von Nachrichten verwendet. Anstatt diese Mechanismen wie bei ROS 1 selber zu entwickeln, werden Standard-Open-Source-Bibliotheken verwendet. ROS 2 ist im Gegensatz zu ROS 1 echtzeitfähig. Außerdem unterstützt es sowohl Unix als auch Windows Plattformen. Allerdings ist ROS 2 nicht mit ROS 1 kompatibel, da API-Änderungen vorgenommen wurden [13].

2.1.5 NAOqi

NAOqi ist die Software für den Nao-Roboter vom Hersteller des Roboters. Der Hersteller ist SoftBank Robotics[3] [6]. Die Software unterstützt die Entwicklung auf den drei gängigsten Betriebssystemen Linux, Windows und Mac. Das Framework besteht aus mehreren Prozessen, die auf unterschiedlicher Hardware laufen können, aber über das Netzwerk verbunden sind. Beispielsweise können einige Prozesse auf dem Nao-Roboter und einige auf einem externen Rechner ausgeführt werden. Das Framework ist eine serviceorientierte Architektur. Die Struktur der Prozesse implementiert das Broker-Architekturpattern (siehe Abb. 6). Beim Programmstart registrieren sich die Module mit ihren Methoden beim Broker. Der Broker bietet für die Module einen Lookupservice an, sodass diese die Methoden anderer Module finden und aufrufen können. Zusätzlich bietet der Broker einen Netzwerkzugang an, um Methoden von Modulen außerhalb des Prozesses aufzurufen. Die Implementierung neuer Module kann sowohl in Python als auch in C++ erfolgen. Das Framework bietet für beide Programmiersprachen dieselben

[3] früher Aldebaran Robotics

API-Funktionen an. Eine genauere Beschreibung des Frameworks findet sich in der Dokumentation [44].

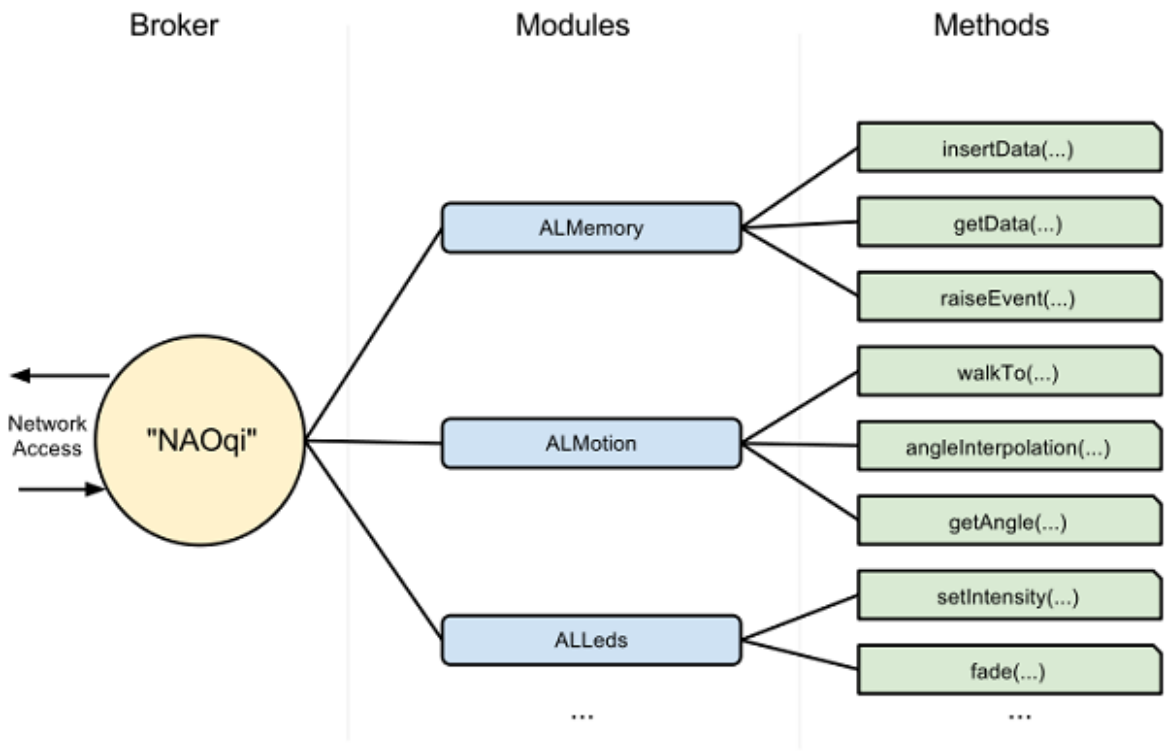

Abbildung 6: Aufbau eines Prozesses im NAOqi Framework aus [44].

2.1.6 Fazit

Die hier betrachteten Softwarearchitekturen haben verschieden Vor- und Nachteile. Alle Architekturen setzen auf eine starke Modularisierung der Software. Im Bereich der Parallelisierung gibt es dagegen Unterschiede. Alle Softwarearchitekturen unterstützen Parallelisierung, aber auf unterschiedlichem Niveau. Das Berlin United Framework erreicht Parallelisierung durch die Aufteilung der Algorithmen in einen Motion- und einen Cognition-Prozess. Eine höhere Parallelisierung findet man dagegen bei B-Human. Dort wird zusätzlich der Cognition-Prozess in drei Threads unterteilt. Dadurch kann die Verarbeitung der beiden Kamerabilder parallel erfolgen. Die höchste Parallelisierung wird bei Nao Devils umgesetzt. Die Verwendung einer Threadpool-Bibliothek ermöglicht die Parallelisierung auf Modul-Ebene. Eine genauso hohe Parallelisierung erreicht ROS. Dort findet die Modularisierung auf Prozessebene statt.

ROS und NAOqi unterstützen die Implementation neuer Module in verschieden Programmiersprachen. Diese erhöhen die Zugänglichkeit zur Software für neue Nutzer. Eine serviceorientierte Architektur wie bei ROS oder NAOqi kommt für Gretchen aber nicht infrage, da eine gute Austauschbarkeit der Module nicht gewährleistet ist.

Wenn Module ausgetauscht werden sollen, dann müssen alle Aufrufe ersetzt und bei der Entfernung von Modulen gelöscht werden.

Eine gute Erweiterbarkeit der Software um weitere Roboterplattformen ist besonders beim Berlin United Framework gegeben. Die Trennung zwischen Algorithmen und Plattformen durch das Plattform-Interface ermöglicht, dass neue Plattformen einfach hinzugefügt werden können. Insbesondere aus diesem Grund bietet es sich an, Gretchen als neue Plattform in das Framework von Berlin United zu integrieren. Dadurch werden der Nao-Roboter und der Gretchen-Roboter miteinander kompatibel. Das heißt, es können schon vorhanden Module des Nao-Roboters auch auf Gretchen ausgeführt werden. Außerdem können zukünftige Experimente parallel auf beiden Robotern getestet werden. Um den Nachteil der geringen Parallelisierung zu lösen, bietet es sich an, den Threadpool-Mechanismus von Nao Devils für das Berlin United Framework zu übernehmen. Davon würde auch der Nao-Roboter profitieren. Allerdings sollte die Umsetzbarkeit erst analysiert werden.

2.2 Kommunikation

Die Kommunikation ist eines der wichtigsten Aspekte in einem Roboter. Sie ermöglicht den Datenaustausch zwischen verschiedenen Hardware- und Softwarekomponenten. Bei Gretchen werden hardwareseitig Sensordaten von den Motoren und der IMU über den Feather an den Raspberry Pi geschickt. Umgekehrt werden Bewegungsbefehle vom Raspberry Pi an den Feather gesendet. Dieser übermittelt die Befehle wiederum an die Motoren. Softwareseitig findet ein Datenaustausch zwischen den verschiedenen Prozessen oder Threads auf dem Raspberry Pi statt. (nachzulesen im Abschnitt 1.3)

Ein Hauptbestandteil der Kommunikation ist die Serialisierung. Diese ist ein Verfahren in der Informatik, bei dem es darum geht, Datenstrukturen oder Objekte in ein Format zu übersetzen, welches gespeichert, übertragen und später wiederhergestellt werden kann [49]. Für die Umsetzung eines solchen Verfahrens gibt es schon verschiedene Bibliotheken. Um die Auswahl dabei etwas einzuschränken, werden im Folgenden nur Binäre Datenformate betrachtet. Diese sind speichereffizienter und damit schneller als menschliche gut lesbare Formate.

2.2.1 Protocol Buffers

Protocol Buffers, oder kurz Protobuf, ist ein von Google entwickeltes Format für die Serialisierung von strukturierten Daten. Das Format wird auf deren Website [23] ausführlich beschrieben. Laut dieser Website ist Protobuf schneller, kleiner und einfacher als XML. Es ist plattformunabhängig und bietet eine Schnittstelle für verschiedene Programmiersprachen, unter anderem C++, Java und Python. Das Serialisierungsformat ist schemabasierend. Das heißt, die Struktur einer Nachricht wird in einem Schema festgeschrieben, welches für die Serialisierung und Deserialisierung benötigt wird. Dieses erfordert, dass die Struktur sowohl beim Sender als auch beim Empfänger bekannt ist. Bei Protocol Buffers wird das Schema einer Nachricht vor der Ausführung des Programmcodes in einer .proto Datei festgelegt und dann in ein Format in der ge-

wünschten Programmiersprache umgewandelt. Protobuf unterstützt für eine Nachricht eine Vielzahl an Datentypen (siehe Algorithmus 1). Zusätzlich bietet es die Möglichkeit, optionale Variablen mit Defaultwerten anzulegen. Diese ermöglichen eine Aufwärts- und Abwärtskompatibilität, ohne eine Versionsverwaltung durchzuführen. Dadurch ist es möglich bei einer späteren Erweiterung des Formats einer Nachricht, den Code, der mit dem alten Format arbeitet, weiterhin auszuführen. Außerdem wird jeder Variable eine eindeutige Nummer zugeordnet. Dadurch können mit einem alten Format erstellte Nachrichten weiterhin gelesen werden.

Zusätzlich bietet Google eine extra kleine Version Nanopb [1] für Mikrocontroller bzw. für jedes speicherbeschränktes System an. Diese C Implementation von Protobuf eignet sich daher besonders für den Feather von Gretchen.

Des Weiteren gibt es die modulare C Implementation Protobluff [7]. Diese verfolgt einen etwas anderen Ansatz. Beim Lesen oder Schreiben von Werten aus Nachrichten werden die erforderlichen Decodierungs- und Codierungsschritte vollständig übersprungen, indem alle Operationen direkt in den codierten Daten ausgeführt werden.

```
message Person {
  required string name = 1;
  optional int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    optional string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}
```

Algorithmus 1: Beispiel Schema einer Nachricht in Protobuf entnommen aus [23].

2.2.2 FlatBuffers

FlatBuffers [15] ist eine ebenfalls von Google entwickelte plattformübergreifende Serialisierungsformat. Das Format wird auf der dazugehörigen Website [15] im Programmer's Guide beschrieben. Für das Format gibt es Implementationen in C++, C, Java, Python und einigen anderen Programmiersprachen. Es ist ebenfalls schemabasierend und damit relativ ähnlich zu Protocol Buffers. Im Gegensatz zu Protocol Buffers bietet FlatBuffers die Möglichkeit, Objekte direkt in den serialisierten Binärdaten zu adressieren und direkt in die Binärdaten zu schreiben. Dadurch entfällt ein aufwendiger Parsingschritt in eine weitere Repräsentation und es muss kein zusätzlicher Speicher allociert werden. Um den direkten Zugriff auf die Binärdaten plattformunabhängig umsetzen zu können,

werden bestimmte Regeln aufgestellt. So ist beispielsweise die Endianness unabhängig vom Betriebssystem immer Little Endian. Das FlatBuffers Schema beinhaltet verschachtelte Objekte wie Strukturen und Tabellen (siehe Algorithmus 2). Die Tabellen sind der Eckpfeiler von FlatBuffers und damit auch die Standardmethode, um Objekte zu definieren. Jede Tabelle besteht aus einer Liste von Feldern. Ein Feld besteht aus einen Namen, einen Datentyp und einen Defaultwert. Jede Tabelle beinhaltet eine vtable. Diese enthält Informationen darüber, welche Felder von dieser Instanz wo gespeichert werden. Für den Fall, dass einem Feld kein Wert zugewiesen wird, wird kein Wert gespeichert aber beim Auslesen der Defaultwert zurückgegeben. Anders ausgedrückt bedeutet dieses, dass Defaultwerte in einer Nachricht nicht gespeichert und daher auch nicht zugewiesen werden. Dieser Mechanismus ermöglicht Aufwärts- und Abwärtskompatibilität beim Ändern eines Schemas innerhalb einer Tabelle. Einerseits können neuen Felder einfach hinzugefügt und weiterhin alte Nachrichten gelesen werden. Andererseits verbrauchen Felder, die nicht mehr verwendet werden, keinen Speicherplatz. Die Verwendung von Tabellen verursacht jedoch ein wenig Speicheroverhead und benötigen etwas höhere Zugriffskosten gegenüber Strukturen. Diese sind einfache und kompakte Objekte in FlatBuffers und sind speichereffizienter als Tabellen, bieten aber dafür keine Versionierungsmöglichkeiten. Laut FlatBuffers eignen sie sich daher am Besten für sehr kleine Objekte wie Koordinatenpaare oder RGBA-Farben.

```
enum Color : byte { Red = 1, Green, Blue }

union Any { Monster, Weapon, Pickup }

struct Vec3 {
  x:float;
  y:float;
  z:float;
}

table Monster {
  pos:Vec3;
  mana:short = 150;
  hp:short = 100;
  name:string;
  friendly:bool = false (deprecated, priority: 1);
  inventory:[ubyte];
  color:Color = Blue;
  test:Any;
}

root_type Monster;
```

Algorithmus 2: Beispiel Schema einer Nachricht in FlatBuffers entnommen aus [15].

Auf der FlatBuffers Website [15] ist ein Benchmarkvergleich [14] zwischen der binären FlatBuffers Version und der Protocol Buffers LITE Version zu sehen. Für den Test wurde eine Menge aus zehn Objekte, die aus einem Array, vier Strings und einer großer Auswahl an verschieden Integer- und Floatwerten bestanden, verwendet. Für das eine

Million malige Codieren dieser Objekte benötigte Protobuf 185 Sekunden und damit ein Vielfaches mehr Zeit als FlatBuffers mit 3,2 Sekunden. Außerdem ist der generierte Quellcode bei FlatBuffers (4 Kilobyte) wesentlich kleiner als bei Protobuf (62 Kilobyte). Allerdings sind die generierten Binärdaten bei FlatBuffers (344 Bytes) etwa 1,5 mal größer als bei Protobuf (228 Bytes).

FlatBuffers bietet zusätzlich noch die schemalose Version FlexBuffers [16]. Dieses Format kann sowohl in Verbindung mit FlatBuffers als auch einzeln verwendet werden. FlexBuffers bietet ebenfalls einen direkten Zugriff auf die Binärdaten ohne Parsingschritt. Im direkten Vergleich ist FlexBuffers aber langsamer als die reguläre FlatBuffers Version. Dieses kann in [16] genauer nachgelesen werden.

2.2.3 Cap'n Proto

Ein weiteres Serialisierungsformat ist Cap'n Proto [47] von Kenton Varda. Dieser ist der Hauptautor von Protocol Buffers Version 2. Wie bei den beiden bisher betrachteten Formaten ist auch Cap'n Proto plattformunabhängig und schemabasierend. Es existieren unter anderem Implementationen in C++, C, Java und Python. Genauso wie bei FlatBuffers werden bei diesem Format die Daten einer Nachricht direkt ohne eine Parsingschritt in den binären Speicher geschrieben bzw. aus dem binären Speicher gelesen. Allerdings verwendet beide Formate eine unterschiedliche Codierung. Bei Cap'n Proto werden die Daten so angeordnet, dass sie effizient von modernen CPUs manipuliert werden können. Das heißt, bei der Codierung einer Nachricht wird auf feste Breiten, feste Offsets und richtiges Alignment geachtet. Für Daten mit variabler Größe werden feste auf Offsets basierende Zeiger verwendet. Um Rückwärtskompatibilität zu erhalten, sind die Element einer Struktur der Reihenfolge nach durchnummeriert. Dadurch können neue Elemente am Ende einer Struktur hinzugefügt werden. Dieses sorgt jedoch dafür, dass beim Lesen eines Elements überprüft werden muss, ob es noch innerhalb der Grenzen der Nachricht liegt. Ein Beispiel für ein Schema ist im Algorithmus 3 zu sehen. Weitere Informationen zu dem Format sind auf der Website [47] zu finden.

Neben dem Fehlen eines Parsingschrittes werden weitere Vorteile gegenüber Protocol Buffers auf der Cap'n Proto Website [47] genannt. Zum einem können Daten bei Cap'n Proto inkrementell gelesen werden. Das heißt, eine Nachricht kann schon verarbeitet werden, auch wenn sich noch nicht vollständig erhalten wurde. Zum anderem wird durch das Fehlen eines Parsingschrittes wesentlich weniger Quellcode generiert.

Im Vergleich von Cap'n Proto mit FlatBuffers hat Cap'n Proto laut der Website von FlatBuffers [15] eine geringere Flexibilität. Dieses liegt daran, dass es keine Möglichkeit gibt, optionalen Werte für veraltete Felder zu definieren oder Defaultwerte für das Serialisieren von fehlender Felder festzulegen.

```
struct Person {
  id @0 :UInt32;
  name @1 :Text;
  email @2 :Text;
  phones @3 :List(PhoneNumber);

  struct PhoneNumber {
    number @0 :Text;
    type @1 :Type;

    enum Type {
      mobile @0;
      home @1;
      work @2;
    }
  }

  employment :union {
    unemployed @4 :Void;
    employer @5 :Text;
    school @6 :Text;
    selfEmployed @7 :Void;
    # We assume that a person is only one of these.
  }
}

struct AddressBook {
  people @0 :List(Person);
}
```

Algorithmus 3: Beispiel Schema einer Nachricht in Cap'n Proto entnommen aus [47].

2.2.4 MessagePack

MessagePack [11] ist ein effizientes Serialisierungsformat, welches auf der dazugehörigen Website mit „It's like JSON. but fast and small." beschrieben wird. Das Format unterstützt über 50 Programmiersprachen. Darunter sind unter anderem C, C++, Java und Python. Im Gegensatz zu den bisher betrachteten Formaten ist dieses Format schemalos. Das heißt, beim Erstellen einer Nachricht gibt es keine vorher definierte Struktur, die vorschreibt, an welchen Stellen welche Art von Informationen gespeichert werden. Stattdessen besteht eine Nachricht aus mehreren aneinander gehängten Daten. Diese beinhalten sowohl den Datentyp als auch den Datenwert. Die Daten werden bei der Serialisierung in ein von MessagePack definiertes Format umgewandelt. Dafür wird eine effiziente Codierung verwendet. Die unterstützten Datentypen und deren genaue Codierung werden ausführlich in der Spezifikation [12] beschrieben. Ein Datenelement besteht im MessagePack Format aus einem oder mehreren Bytes. Dabei beschreiben die ersten Bits des ersten Bytes den Datentyp. Dadurch benötigt die Codierung kleiner Integers nur ein Byte oder kurze Strings nur ein zusätzliches Byte [11].

2.2.5 Concise Binary Object Representation

Concise Binary Object Representation (CBOR) ist ein weiteres Serialisierungsformat, welches in dem Internet Standard RFC 8949 [4] definiert wird. Auf der CBOR Website [5] sind Implementationen für ein große Anzahl an Programmiersprachen zu finden. Darunter sind C, C++, Java und Python. Genauso wie MessagePack ist CBOR schemalos. Das heißt, das Format benötigt für die Codierung von Nachrichten kein vorher definiertes Nachrichtenschema. Stattdessen besteht eine Nachricht aus einem Datenstrom aus mehreren Datenelementen. Für die Codierung der Datenelemente wird eine ähnliche Codierung wie bei MessagePack verwendet. Ein Datenelement setzt sich dabei aus einen ein Byte großen Header einer optionalen Headererweiterung und einer optionalen Nutzlast zusammen [48]. Das Header Byte unterteilt sich in drei Bit für den Hauptdatentyp und fünf Bit für zusätzliche Informationen. Diese zusätzlichen Informationen können für kleine Werte auch die Daten enthalten. Größere Daten werden in der optionalen Nutzlast gespeichert [4].

Die große Ähnlichkeit zu MessagePack ist laut [4] kein Zufall. CBOR wurde von MessagePack inspiriert. Allerdings sind die beiden Formate nicht miteinander kompatibel. Laut [4] unterscheiden sich die beiden Formate an den unterschiedlichen Designzielen und Anforderungen. Bei CBOR sind die Ziele eine extrem kleine Codegröße, eine relativ kleine Nachrichtengröße und eine Erweiterbarkeit ohne die Notwendigkeit einer Versionierung.

2.2.6 Fazit

Alle in diesem Kapitel betrachteten Serialisierungsformate haben verschiedene Eigenschaften, die sich unterschiedlich für den Einsatz in dem Gretchen-Roboter eignen. Es gibt ein paar Gemeinsamkeiten, die alle betrachteten Formate besitzen. Alle Serialisierungsformate arbeiten auf binären Codierungen, sind plattformunabhängig und unterstützen die gängigen Programmiersprachen C++, Java und Python. Die betrachteten Formate lassen sich in zwei Kategorien einteilen. Zum einem gibt es die drei auf schemabasierende Formate Protocol Buffers, FlatBuffers und Cap'n Proto. Zum anderem gibt es die schemalosen Formate MessagePack und CBOR. Da es für beide Varianten sinnvolle Anwendungsbereiche in einem Roboterframework gibt, werden nachfolgend für beide Empfehlungen abgegeben.

Der Vorteil bei den auf schemabasierenden Formaten ist, dass beim Erstellen einer Nachricht die Korrektheit der Struktur vom Compiler überprüft wird. Außerdem können diese Formate im Vergleich zu schemalosen Formaten kompakter codiert werden, da die Struktur der Daten sowohl beim Sender als auch beim Empfänger bekannt ist und nicht mit übertragen wird. Diese Art der Übertragung eignet sich besonders für die Codierung von Sensordaten, da die Struktur der Daten immer gleich und von Anfang an bekannt ist. Die beiden Formate FlatBuffers und Cap'n Proto haben den Vorteil, dass beim Lesen und Schreiben einer Nachricht kein zusätzlicher Parsingschritt benötigt wird. Im Gegensatz zu dem Protocol Buffers Format, werden die Daten direkt in dem binären Format geschrieben und können daraus direkt gelesen werden. Dadurch

erfolgt eine schnellere Codierung und Decodierung. Ein weiter wichtiger Punkt bei auf schemabasierenden Formaten ist der Umgang mit zukünftigen Erweiterungen und Änderungen an der Struktur einer Nachricht. Ein Format sollte sowohl Aufwärts- als auch Abwärtskompatibilität bieten, damit beispielsweise veralte gesammelte Logdaten für Simulationszwecke weiterhin benutzt werden können, unabhängig davon, ob Datenfelder hinzugefügt oder entfernt werden. Im Vergleich der Formate bezüglich der Kompatibilität schneiden die beiden Google Bibliotheken Protocol Buffers und FlatBuffers besser ab. Sie haben durch die Möglichkeit optionale Felder und Defaultwerte zu verwenden eine hohe Flexibilität. Bei Cap'n Proto gibt es dagegen nur die Möglichkeit Nachrichten am Ende zu erweitern. Alles in allem eignet sich das FlatBuffers schemabasierendes Serialisierungsformat für die Verwendung in einem Roboter aufgrund der hohen Flexibilität und der Möglichkeit, Daten direkt und schnell zu codieren am Besten.

Bei den schemalosen Formaten hingegen hat der Verwender einer höhere Flexibilität, da eine Nachricht keine konkrete Struktur aufweist. Dafür erfolgt aber keine automatische Validierung der Struktur einer Nachricht und Datentypen müssen mit in der Nachricht codiert sein. Die hier betrachteten Formate MessagePack und CBOR sind relativ ähnlich. Beide Formate verwenden eine effiziente Binärcodierung für verschiedene Datentypen. Diese Codierungen sind beide nicht miteinander kompatibel und weisen minimale Unterschiede auf. So gibt es beispielsweise bei MessagePack die Möglichkeit innerhalb eines Bytes einen 7 Bit großen positiven Integerwert zu codieren [12]. Bei CBOR hingegen kann in einem Byte nur ein 5 Bit positiver Integerwert codiert werden [4]. Außerdem unterscheiden sich die Formate in den unterstützten Datentypen. MessagePack unterstützt nur Standarddatentypen, wie Integer, Float, String und Array [12]. CBOR hingegen unterstützt auch exotische Formate wie Bigfloat, Decimal Fraction und epoch-based Date [4]. Trotz dieser Unterschiede sind beide Formate relativ ähnlich. Daher kann an dieser Stelle für kein Format eine eindeutige Empfehlung abgegeben werden. Welches Format sich eher eignet ist abhängig vom Anwendungsfall, also welche Daten codiert werden sollen.

3 Softwarearchitektur für Gretchen

Dieser Abschnitt behandelt die praktische Umsetzung der Ergebnisse aus Abschnitt 2. Dabei unterteilt sich dieser Abschnitt in zwei Teile. Im ersten Teil wird das Webots-Modell des Gretchen-Roboters als neue Plattform in das Softwareframework von Berlin United integriert. Im zweiten Teil wird dieses Framework um ein Threading-Mechanismus erweitert,

3.1 Webots

Webots [26] ist ein Open-Source Roboter-Simulator. Die Cyberbotics Ltd. entwickelt den Roboter-Simulator seit 1998. Der Simulator ist in der Industrie, Lehre und Forschung weit verbreitet. Im Webots User Guide [27] werden die Möglichkeiten von Webots detailliert beschrieben. Webots bietet eine große Anzahl an Robotermodellen, die für die Simulation verwendet werden können. Es gibt aber auch die Möglichkeit, eigene Robotermodelle zu entwickeln. Diese Roboter können mithilfe eines Controllers gesteuert werden. Dieser kann in den gängigen Programmiersprachen C, C++, Java, Python oder MATLAB entwickelt werden.

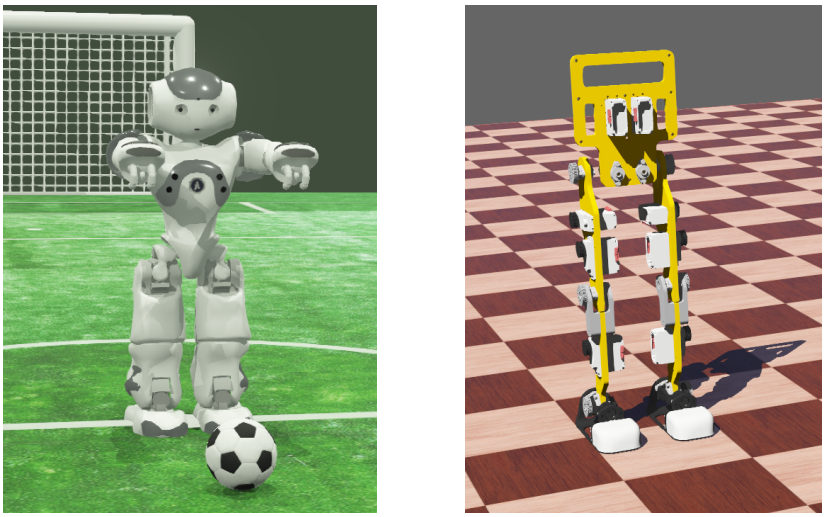

Abbildung 7: Webots-Modelle des Nao- (links) und Gretchen-Roboters (rechts).

Sowohl für den Gretchen-Roboter als auch für den Nao-Roboter wurden Modelle für Webots entwickelt (siehe Abb. 7). Die Webots-Simulation des Nao-Roboters ist als eine mögliche Plattform in das Framework von Berlin United integriert (siehe Abb. 8). Für die Integration wurde das Client-Server-Modell verwendet. Dadurch sind das Framework und der Webots Controller zwei voneinander unabhängige Programme, die separat installiert, compiliert und ausgeführt werden. Der Webots Controller agiert als Server mit dem sich das Framework als Client verbindet. Der Austausch der Sensordaten und Motorbefehle erfolgt über die Netzwerkinfrastruktur. Dafür werden die Daten mithilfe von MessagePack (siehe Abschnitt 2.2.4) serialisiert.

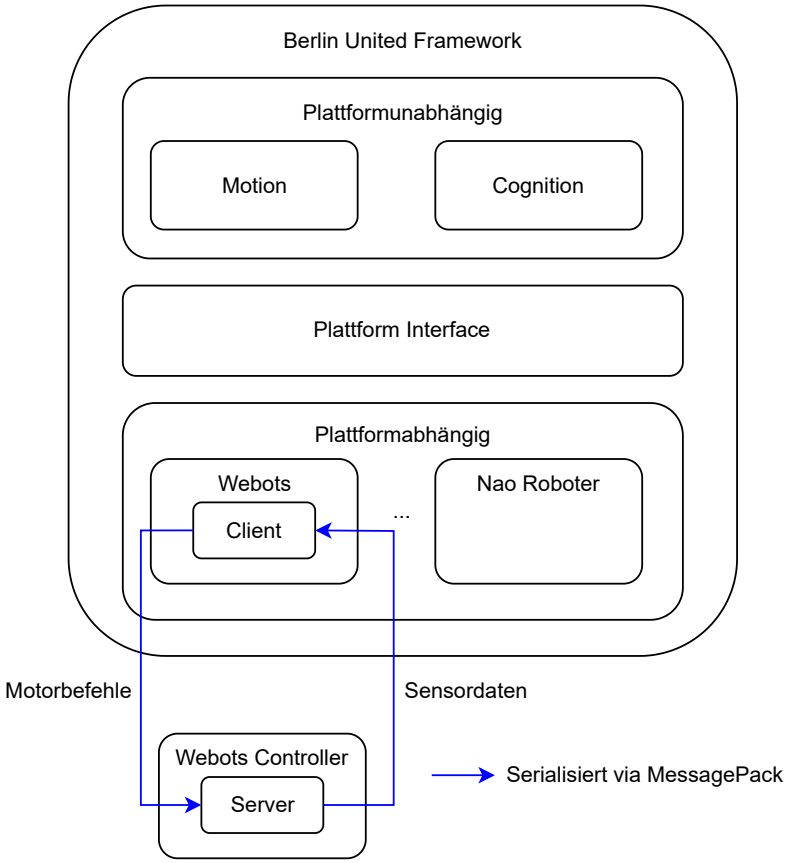

Abbildung 8: Integration der Webots-Simulation in das Berlin United Framework.

Die Integration der Gretchen Simulation in Webots als neue Plattform in das Berlin United Framework kann ähnlich umgesetzt werden. Da beide Simulation mit dem Webots-Simulator entwickelt wurden, unterscheiden sie sich nur durch die Verwendung verschiedener Motoren und Sensoren. Daher ist lediglich eine Anpassung des Datenformat für die Serialisierung mit MessagePack notwendig.

Um die Integration der neuen Plattform zu testen, wurden zwei Motion-Module für den Roboter Gretchen entwickelt. Beide Module implementieren auf sinusförmigen Funktionen basierende Bewegungsalgorithmen. Das Modul *Walk* implementiert einen Laufalgorithmus und das Modul *Squats* eine Algorithmus für Kniebeugen. Beide Bewegungen können mithilfe der Representation *MotionRequest* von einem Cognition-Modul angefordert werden. Durch die Verwendung des Datentyps *enum* im *MotionRequest* kann zu einem Zeitpunkt immer nur eine Bewegung ausgeführt werden. Bei der Änderung des *MotionRequest* wird die aktuell ausgeführte Bewegung beendet und die angefragte Bewegung ausgeführt. Der Wechsel erfolgt allerdings erst, wenn Gretchen eine stabile Pose erreicht. Dieses ist beim Laufalgorithmus beispielsweise der Fall, wenn Gretchen mit beiden Beinen parallel auf dem Boden steht. Für Demonstrationszwecke wurde ein Cognition-Modul implementiert, welches die beiden Motion-Module abwechselnd anfordert.

3.2 Parallelisierung

Wie im Abschnitt 2.1 über Softwarearchitekturen für humanoide Roboter beschrieben wurde, kann das Softwareframework von Berlin United weiter parallelisiert werden. Dafür kann der Mechanismus mit Taskflow wie bei Nao Devils verwendet werden. Dieser ermöglicht die parallele Ausführung auf Modul-Ebene durch die Verwendung eines Threadpools. Bevor das Ganze jedoch umgesetzt werden kann, werden zuerst die Abhängigkeiten der Module untereinander analysiert. Außerdem wird die Taskflow Bibliothek etwas genauer betrachtet.

3.2.1 Analyse

Wie bei der Betrachtung des Berlin United Frameworks bereits erwähnt, besteht die Software aus den beiden Prozessen Cognition und Motion. In diesen Prozessen werden mehrere Module ausgeführt. Dafür werden die Module für jeden Prozess in der *ExecutionList* gespeichert. Von dieser Liste werden sie sequentiell aufgerufen. Der Datenaustausch zwischen den Modulen erfolgt mithilfe von Representationen. Die Module können in einer *Require*- oder in einer *Provide*-Beziehung zu einer Representation stehen. Bei der *Provide*-Beziehung erzeugt ein Modul neue Daten für die Representation. Daher erhält es vollen Zugriff auf die Representationen und kann sowohl lesen als auch schreiben. Steht ein Modul in einer *Require*-Beziehung zu einer Representation, werden auf die Daten der Representation zugegriffen. Ein Modul erhält dafür nur Leserechte. Diese zwei Beziehungsarten erzeugen Abhängigkeiten zwischen den Modulen, die bei einer Parallelisierung zu beachten sind. Die Anordnung der Module in der *ExecutionList* ist so gewählt, dass ein Modul möglichst erst dann ausgeführt wird, wenn alle Representationen mit einer *Require*-Beziehungen von vorherigen Modulen geschrieben wurden. Aus diesen benötigten Representationen berechnet und erzeugt ein Modul neue Representationen. Diese können dann von den folgenden Modulen verarbeitet werden.

Neben den Representationen, die durch die Module durchgereicht werden, gibt es ein paar Representationen, die von vielen Modulen geschrieben werden. Dieses trifft überwiegend auf Representationen zu, die zum debuggen genutzt werden (siehe Tabelle 1). Insgesamt besteht die *ExecutionList* im Cognition-Prozess aus 89 Modulen. Da alleine schon 72 davon in die *DebugRequest*-Representation schreiben, muss dafür gesorgt werde, dass die Debug-Representationen nicht die parallele Ausführung von Modulen behindert.

Ein weitere Punkt, den es zu beachten gilt, sind Abhängigkeitszyklen zwischen den Modulen. Beispielsweise gibt es eine gegenseitige Abhängigkeit zwischen den Modulen *MultiKalmanBallLocator* und *CNNBallDetector*. Die Representation *BallModel* wird vom *MultiKalmanBallLocator* geschrieben und von *CNNBallDetector* gelesen. Bei der Representation *MultiBallPercept* ist es genau umgekehrt. Weitere Zyklen sind im Abhängigkeitsgraphen zu erkennen (siehe Abb. 9). Damit dieser Graph anschaulich bleibt, werden nicht alle Abhängigkeiten dargestellt. Es werden Abhängigkeiten weggelassen, die indirekt über andere Module schon existieren. Wenn man die Zyklen genauer

Debug-Representation	**Anzahl der *Provide*-Beziehungen**
DebugRequest	72
DebugParameterList	57
DebugDrawings	46
DebugModify	36
DebugPlot	26
DebugImageDrawings	22
DebugImageDrawingsTop	22
StopwatchManager	19
DebugDrawings3D	3
DebugCommandManager	2
DebugMessageOut	1
DebugMessageInCognition	1

Tabelle 1: Anzahl von *Provide*-Beziehungen pro Debug-Representation im Cognition-Prozess für die Webots-Plattform.

betrachtet, kann man feststellen, dass einige Abhängigkeiten von ihnen notwendig sind und andere nicht. Beispielsweise wird im Modul *TeamCommReceiver* die Representation *GameData* nicht verwendet, aber als *Require*-Beziehung angegeben. Von diesen überflüssigen *Require*-Beziehungen existieren noch mindestens fünf weitere. Möglicherweise wurden diese Beziehungen für Debug-Zwecke oder für Experimente verwendet. Weiter Zyklen lassen sich entfernen, indem Module in der *ExecutionList* anders angeordnet werden. Im Modul *PathPlanner2018* wird die Representation *MotionRequest* erzeugt. Diese wird von dem Modul *XABSLBehaviorControl* benötigt. Das Problem dabei ist, dass das Modul vor dem *PathPlanner2018* in der *ExecutionList* steht. Dadurch arbeitet das Modul immer mit veralteten Daten. Durch den Tausch der Reihenfolge bei der Ausführung der Module lässt sich dieser Zyklus verhindern. Es gibt aber auch einige Module, bei denen Zyklen erwünscht sind. Diese Module verwenden beabsichtigt Representationen aus der vorherigen Ausführung der *ExecutionList*.

3.2.2 Taskflow

Taskflow [19] ist eine Threadpool-Bibliothek, die an der University of Illinois at Urbana-Champaign entwickelt wurde. Die header-only C++ Bibliothek ist in C++17 geschrieben und erlaubt dadurch die parallele Verwendung moderner C++ Features. Sie ermöglicht dem Programmierer die Entwicklung paralleler Programme, ohne das Erstellen und Händeln von mehreren Threads mit komplexen Locking-Mechanismen. Durch die Verwendung von Taskflow genügt es, mithilfe der einfachen und ausdrucksstarken Graphenbeschreibungssprache (siehe Algorithmus 4) eine Task-basierten Abhängigkeitsgraphen zu erstellen [45]. Diese Form der Parallelisierung eignet sich ideal für modulbasierte Architekturen, da die Module mit ihren gegenseitigen Abhängigkeiten

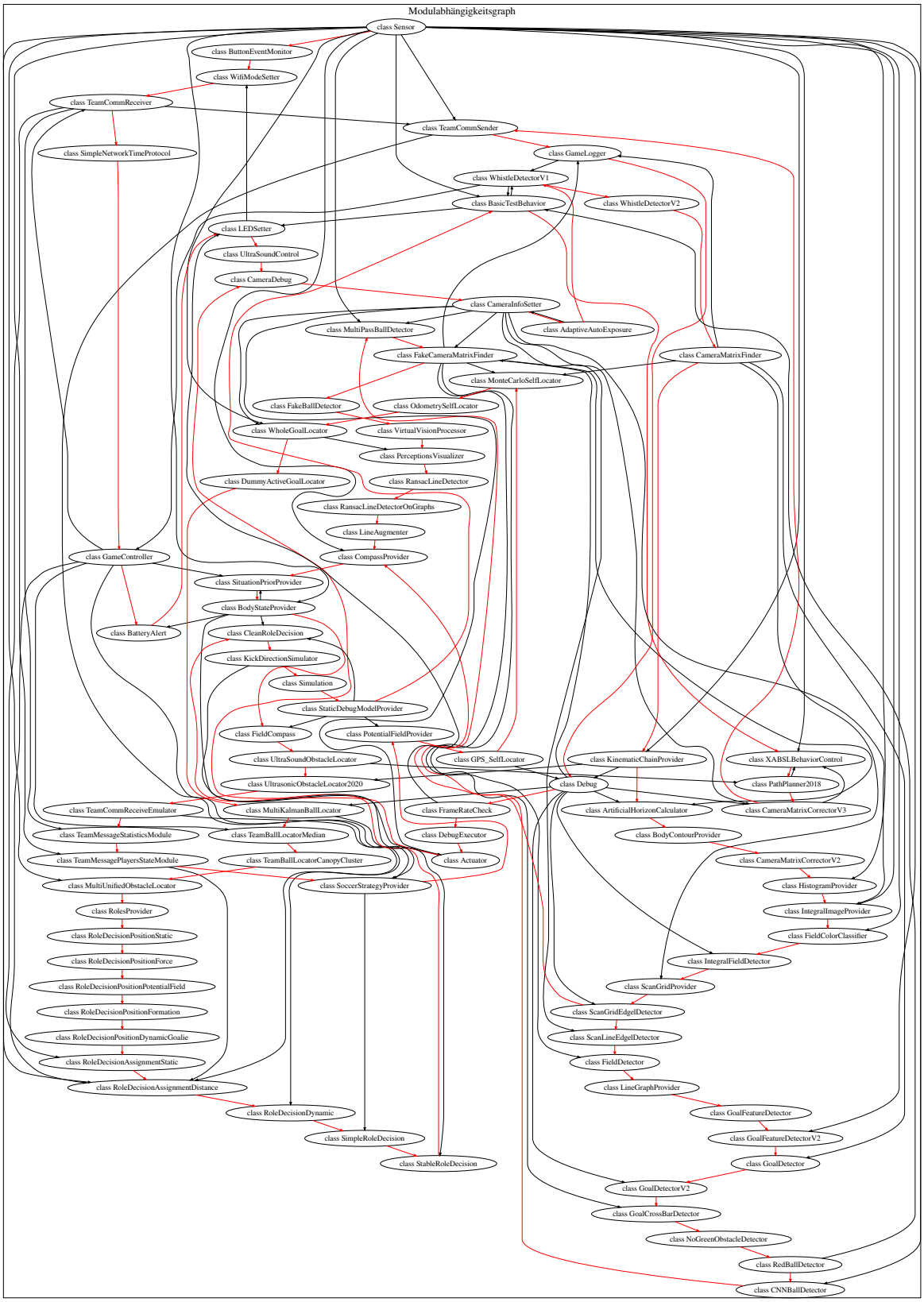

Abbildung 9: Dieser Graph zeigt Abhängigkeiten der Module durch die Representationen im Cognition-Prozess des Nao-Roboters von Berlin United. Die roten Pfeile zeigen die Anordnung der Module in der *ExecutionList*. Der Graph kann in digitaler Form genauer betrachtet werden.

auch einen Abhängigkeitsgraphen darstellen. Außerdem bietet die Bibliothek einige hilfreiche Debug-Funktionen. Es gibt die Möglichkeit, über ein Web-Interface, sich die Aufteilung der einzelnen Tasks auf die Threads über den Programmverlauf zu visualisieren. Auch kann man sich den erstellten Taskflowgraphen ausgeben lassen [20]. Die Entwickler haben die Performance von Taskflow mit den in der Industrie verbreiteten Bibliotheken OpenMP und Intel TBB [4] verglichen. Diese unterstützen ebenfalls die Parallelisierung mehrerer Tasks mit Abhängigkeiten. Sowohl bei Micro-Benchmarks als auch bei Real-World Anwendungen mit Millionen von Tasks erreichte die Taskflow Bibliothek eine 14 bis 38 % bessere Geschwindigkeit bei einer geringeren Code-Komplexität [45].

```
tf::Executor executor;
tf::Taskflow taskflow;

auto [A, B, C, D] = taskflow.emplace(
    [] () { std::cout << "TaskA\n"; },
    [] () { std::cout << "TaskB\n"; },
    [] () { std::cout << "TaskC\n"; },
    [] () { std::cout << "TaskD\n"; }
);

A.precede(B, C);   // A runs before B and C
D.succeed(B, C);   // D runs after  B and C

executor.run(taskflow).wait();
```

Algorithmus 4: Ein einfaches Taskflow Beispiel aus [19].

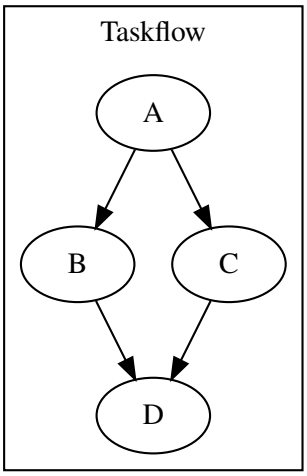

Abbildung 10: Taskflowgraph für das Beispiel in Listing 4.

Für die Parallelisierung des Nao Devils Frameworks hat Aaron Larisch in seiner Masterarbeit [25] ebenfalls die Bibliotheken OpenMP und Intel TBB zusammen mit

[4] Intel Threading Building Blocks

Taskflow untersucht. Bei der Untersuchung kam heraus, dass sich sowohl Taskflow als auch Intel TBB für die Parallelisierung des Frameworks eignen. Taskflow bietet aber eine bessere Performance, besitzt ein hilfreicheres Debug-Interface sowie eine saubere API. Daher wurde Taskflow für die Parallelisierung des Nao Devils Frameworks verwendet.

3.2.3 Umsetzung

Auf Grundlage der obigen Analyse wurde entschieden, Taskflow für die Parallelisierung der Berlin United Software zu verwenden. Bevor die Bibliothek verwendet werden kann, muss die Compiler Version von C++14 auf C++17 geändert werden. Für die Einbindung von Taskflow für die parallele Ausführung der Module werden ein paar Änderungen im Modul-Manager vorgenommen. Nach der Registrierung aller Module und der Erstellung der *ExecutionList*, wird ein Taskflowgraph generiert (siehe Algorithmus 5). Dafür wird als erstes für jedes Modul in der *ExecutionList* ein Taskobjekt erstellt. Diesem Taskobjekt wird eine Lambda-Funktion übergeben, die bei der Ausführung des Taskflowgraphen die *execute()*-Funktion des Moduls aufruft. Als nächstes müssen die Abhängigkeiten zwischen den Modulen in den Taskflowgraphen übernommen werden. Diese werden in Taskflow durch gerichtete Kanten zwischen den Taskobjekten beschrieben. Es gibt zwei verschieden Arten von Abhängigkeiten zwischen Modulen.

Die erste Art von Abhängigkeit entsteht, wenn zwei oder mehrere Module in dieselbe Representation schreiben. Wenn Module parallel ausgeführt werden, ist es wichtig, dass in eine Representation immer nur ein Modul schreiben darf. Dieses ist notwendig, um Data-Races zu verhindern. Daher werden Kanten zwischen diesen Modulen erzeugt. Um dabei keine Deadlocks durch Abhängigkeitszyklen zu generieren, werden die Kanten immer in Richtung der *ExecutionList* erzeugt.

Die zweite Art von Abhängigkeit existiert zwischen zwei Modulen, wenn für eine Representation ein Modul Schreibrechte und ein anderes Modul Leserechte besitzt. Module, die eine Representationen schreiben, sollten vor Modulen, die diese Representationen lesen, ausgeführt werden. Dieses hat zwei Gründe. Zum einen könnte es sonst zu Data-Races kommen und zum anderen könnte es vorkommen, dass ein Modul mit veralten Daten arbeitet. Daher werden gerichtete Kanten von Modulen, die eine Representation schreiben, zu Modulen, die diese Representation lesen, im Taskflowgraphen hinzugefügt. Es gibt allerdings eine Ausnahme. Um Deadlocks zu verhindern, wird eine solche Kante nur dann hinzugefügt, wenn es keine direkte oder indirekte Kante in die andere Richtung gibt. Der Grund für diese Regelung sind die in der Analyse beschriebenen Abhängigkeitszyklen.

Um den Taskflowgraphen möglichst klein zu halten, werden auch nur Kanten übernommen, die noch nicht direkt oder indirekt im Graphen vorhanden sind. Bei der Suche nach einer indirekten Kante im Graphen wird die Berechnungszeit dadurch verkürzt. Nachdem der Taskflowgraph mit allen Abhängigkeiten erzeugt wurde, kann der Graph ausgeführt werden. Dafür wird das sequentielle Ausführen der Module in der *call()*-Funktion eines Prozesses durch den Aufruf für die Ausführung des Taskflowgraphen ersetzt. Der generierte Taskflowgraph ist in Abb. 11 zu sehen.

Algorithmus 5 : Pseudocode für die Generierung des Taskflowgraphen

```
1  Function generateTaskflowGraph:
2      list⟨Module, Task⟩ TaskList
3      list⟨Representation, list⟨Module⟩⟩ provide
4      list⟨Representation, list⟨Module⟩⟩ require
5      for Module in ExecutionList do
6          Task = Taskflow.emplace( module.execute() )
7          Tasklist.add([Module, Task])
8          for Representation in Module.getRequire() do
9              require[Representation].add(Module)
10         for Representation in Module.getProvide() do
11             provide[Representation].add(Module)
       /* add provide dependencies                                    */
12     for [_, ModuleList] in provide do
13         Module1 = null
14         for Module2 in moduleList do
15             if Module1 != null then
16                 if findRecursivDependency(Module1, Module2) then continue
17                 Module1.precede(Module2)
18             Module1 = Module2
       /* add require dependencies                                    */
19     for [Representation, moduleList] in provide do
20         for ProviderModule in moduleList do
21             for RequireModule in require[Representation] do
22                 if ProviderModule = RequireModule then
23                     continue
24                 if findRecursivDependency(ProviderModule, RequireModule) then
25                     continue
26                 if findRecursivDependency(RequireModule, ProviderModule) then
27                     continue
28                 ProviderModule.precede(RequireModule)
29 Funktion findRecursivDependency(startTask, searchTask):
30     Queue<Task> queue queue.push(startTask)
31     while not queue.empty() do
32         task = queue.pop()
33         if task = searchTask then
34             return true
35         for successor in task.successors() do
36             queue.push(successor)
37     return false
```

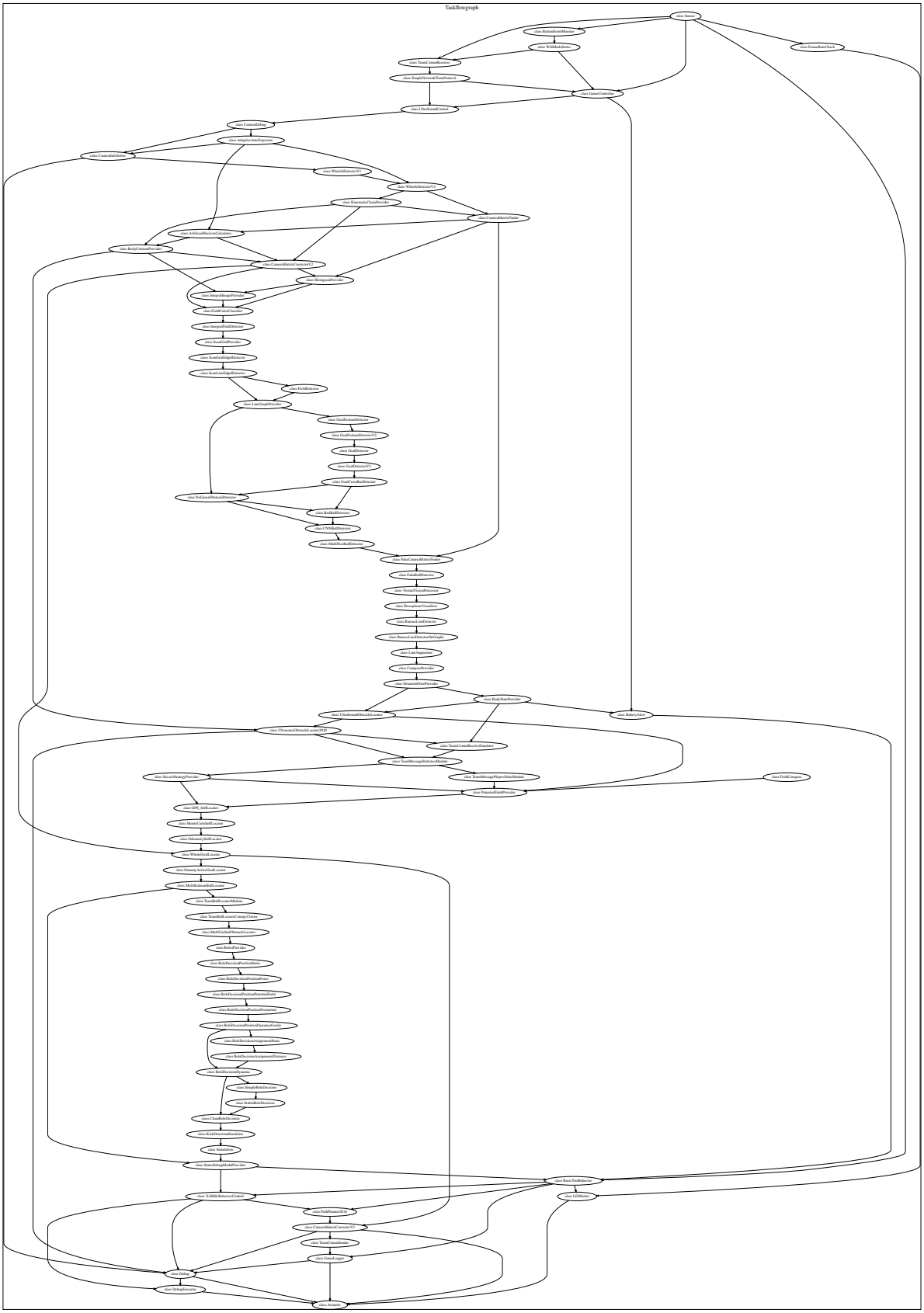

Abbildung 11: Erzeugter Taskflowgraph für den Cognition-Prozess des Nao-Roboters von Berlin United. Der Graph kann in digitaler Form genauer betrachtet werden.

4 Zusammenfassung und Diskurs

In diesem Abschnitt werden die Ergebnisse dieser Arbeit zusammengefasst. Dabei wird insbesondere auf den aktuellen Stand des Softwareframeworks für den humanoiden Roboter Gretchen eingegangen. Es werden verschiedene Herausforderungen betrachtet, die sich im Laufe dieser Arbeit ergeben haben und es wird diskutiert, wie diese in zukünftigen Projekten bearbeitet werden können.

4.1 Zusammenfassung

Das Ziel dieser Arbeit bestand darin, ein Softwareframework für den Roboter Gretchen zu entwickeln. Dafür wurden verschieden Softwareframeworks von mehreren Robotern analysiert und gegenübergestellt. Auf Basis dieser Analyse hat sich das Softwareframework von Berlin United als eine geigende Wahl herausgestellt und wurde daher für Gretchen ausgewählt. Durch die klare Trennung von Algorithmus und Plattform bietet es die Möglichkeit, dieselben Algorithmen für verschiedene Plattformen zu entwickeln. Außerdem ermöglicht die Architektur eine einfache Erweiterbarkeit des Frameworks um weitere Plattformen. Allerdings unterstützt das Framework kein Multithreading. Dieses ist jedoch notwendig, um mehrere Kerne eines Prozessors auf einem Roboter auszunutzen. Dadurch können rechenaufwändige Algorithmen, wie beispielsweise die Bildverarbeitung von mehreren Kameras, parallel ausgeführt werden.

Daher wurde beschlossen das Berlin United Framework auf Modul-Ebene zu parallelisieren. Dafür wurde jeder Prozess um einen Threadpool erweitert. Dieser ermöglicht es, dass die Module nicht mehr sequenziell sondern parallel ausgeführt werden können. Um Data-Races zu verhindern, war dabei zu beachten, dass ein Modul nur dann eine Representation schreiben darf, wenn kein anderes Modul auf diese zugreift. Für die Umsetzung wurde die Threadpool-Bibliothek Taskflow verwendet. Da für den Gretchen-Roboter noch keine komplexe Cognition existiert, wurde die Cognition des Nao-Roboters von Berlin United bezüglich der Parallelisierungsmöglichkeiten untersucht. Dabei hat sich herausgestellt, dass eine nutzbare Parallelisierung erst dann entsteht, wenn der Zugriff auf Debug-Representationen asynchron erfolgen kann.

Da die Kommunikation, der Datenaustausch zwischen verschiedenen Hard- und Softwarekomponenten, in einem Roboterframework eine entscheidende Rolle spielt, wurden verschiedene Serialisierungsformate bezüglich ihrer Eignung im Gretchen-Roboter analysiert und verglichen. Bei den schemabasierenden Formaten, die kompakter codiert und vom Compiler auf Korrektheit der Struktur überprüft werden können, hat sich FlatBuffers als am besten geeignet herausgestellt. Dieses Format bietet Zero-Copy-Möglichkeiten und eine hohe Flexibilität bezüglich spätere Änderungen an. Eine noch höhere Flexibilität versprachen die schemalosen Formate. Sowohl MessagePack als auch CBOR, verwenden eine effiziente Binärcodierung für verschiedene Datentypen. Da diese beiden Formate relativ ähnlich sind, konnte hierfür keine eindeutige Empfehlung abgegeben werden.

Praktisch wurde in dieser Arbeit die Webots-Simulation von Gretchen als neue Plattform in das Berlin United Framework integriert (siehe Abb. 12). Für Testzwe-

cke wurden für den Motion-Prozess zwei Bewegungsalgorithmen zum Laufen und Kniebeugen implementiert. Diese können mithilfe der Representation *MotionRequest* von anderen Modulen angefordert werden. Für Demonstrationszwecke wurde zusätzlich ein Cognition-Modul implementiert, welches abwechselnd die beiden Bewegungen anfordert. Wenn das Berlin United Framework mit diesen Modulen auf der Gretchen-Webots-Plattform ausgeführt wird, dann läuft der Gretchen-Roboter in der Simulation abwechselnd ein paar Schritte und macht dann ein paar Kniebeugen.

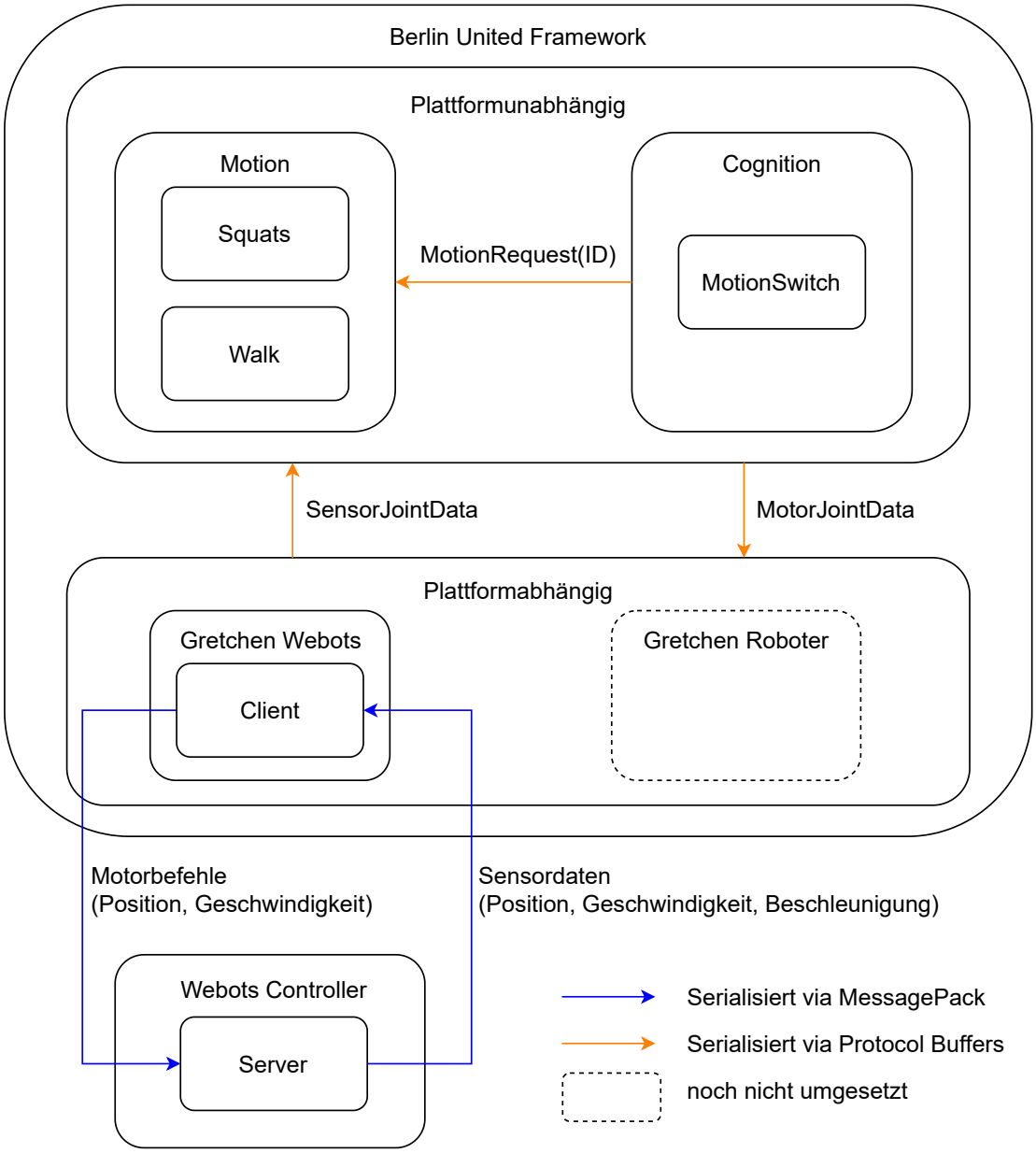

Abbildung 12: Integration des Gretchen-Roboters in das Berlin United Framework.

4.2 Zukünftige Projekte

Ein wichtiges Projekt für die Entwicklung von Gretchen ist die Fertigstellung der Low-Level Architektur des Roboters. Dieser Schritt ist notwendig, um den Roboter als Plattform in das Softwareframework von Berlin United integrieren zu können. Dafür muss insbesondere die Kommunikation zwischen den verschiedenen Hardwarekomponenten realisiert werden. Für die Umsetzung dieser Kommunikation können die Ergebnisse zum Thema Serialisierung aus dem Abschnitt 2.2 verwendet werden.

In dieser Arbeit wurden die Ergebnisse der Analyse der Kommunikation noch nicht in das Framework integriert. Bisher wurde die Analyse verschiedener Serialisierungsformate nur literaturbasiert durchgeführt. Eine praktische Untersuchung dieser Formate bezüglich ihrer Benutzerfreundlichkeit erfolgte bisher nicht. Das Berlin United Framework verwendet für die Serialisierung das Protocol Buffers Format, diese Arbeit empfiehlt jedoch FlatBuffers. Praktische Tests und ein möglicher Wechsel der Formate stehen noch aus.

Bei der manuellen Analyse des Cognition-Prozesses des Nao-Roboters von Berlin United im Abschnitt 3.2.1 hat sich gezeigt, dass die *Require*- und *Provide*-Beziehungen zwischen den Modulen und den Representationen einige Widersprüche aufweisen. Beispielsweise gab es mehrere Module, die eine *Require*- oder *Provide*-Beziehungen auf eine Representation besitzen, aber nie auf diese zugreifen. Diese sorgen bei der Parallelisierung für unnötige Abhängigkeiten im Taskflowgraphen und verhindern damit die parallele Ausführung von Modulen. Des Weiteren gibt es mehrere Abhängigkeitszyklen zwischen den Modulen. Für die optimale Nutzung des Threading-Mechanismus sollte analysiert werden, ob diese unbedingt notwendig sind oder ob diese beispielsweise nur durch eine falsche Anordnung in der *ExecutionList* erzeugt werden. In der Analyse wurden einige dieser Fehler durch manuelle Suche gefunden. Da die Analyse manuell durchgeführt wurde, können weitere Fehler nicht ausgeschlossen werden. Daher empfiehlt es sich, in einer künftigen Arbeit eine genaue Analyse dieser Abhängigkeiten durchzuführen. Insbesondere ist eine automatische Suche nach dieser Art von Fehlern zu empfehlen, um weitere zukünftig zu verhindern.

Weiter hat sich herausgestellt, dass einige Representationen von einer hohen Anzahl von Modulen erzeugt werden. Dieses trifft überwiegend auf Representationen zu, die zum debuggen genutzt werden. In diese Debug-Representationen wird jedoch in der Regel im Normalbetrieb nur selten geschrieben. Der Zugriff auf eine Representation wird aber für die gesamte Laufzeit eines Moduls reserviert. Die Debug-Representationen sorgen nun dafür, dass ein großer Teil des Cognition-Prozesses sequentiell ausgeführt wird (siehe Abb. 11). Um den Nutzen des Threadpool-Mechanismus zu erhöhen, sollte es zukünftig eine Möglichkeit geben, auf diese Representationen asynchron zugreifen zu können. Die Abb. 13 zeigt den generierten Taskflowgraphen, wenn die Abhängigkeiten der Debug-Representationen vernachlässigt werden. Es ist zu erkennen, dass ein besserer Grad an Parallelisierung erreicht werden kann. Ein paar Möglichkeiten für eine solche Umsetzung bietet die Taskflow Bibliothek. So gibt es einen integrierten Semaphor Mechanismus [20]. Für Representation, auf die asynchron zugegriffen werden soll, kann jeweils ein Semaphor der Größe eins definiert werden. Bei einem Zugriff auf diese

Representation wird dieser Semaphor erst reserviert und nach dem Zugriff wieder freigegeben. Eine andere Möglichkeit ist die Verwendung von *CriticalSection* [20] von Taskflow. Diese abstrahieren den Semaphor Mechanismus. Einer *CriticalSection* können mehrere Tasks von einem Taskflowgraphen hinzugefügt werden. Innerhalb dieses kritischen Bereiches kann die Anzahl der parallellaufenden Tasks begrenzt werden. Für einen asynchronen Zugriff auf eine Representation kann pro Representation eine *CriticalSection* mit nur einem parallel laufenden Task erzeugt werden. Jeder Zugriff auf einen Representation kann dann in einem separaten Task durchgeführt werden, welcher der jeweiligen *CriticalSection* hinzugefügt wird. Bei der Umsetzung eines solchen Mechanismus sollte darauf geachtet werden, dass die Verwendung für den Programmierer möglichst einfach gehalten wird. Dieses bedeutet, dass die Beschränkung der Zugriffe auf Representationen automatisiert erfolgen sollte. Dadurch können Fehler, wie das einfache Zugreifen ohne Erlaubnis, mit der Folge von undefinierten Verhalten oder das Vergessen der Freigabe eines Semaphors, verhindert werden.

Die in dieser Arbeit für die Parallelisierung des Cognition- und des Motion-Prozesses verwendete Bibliothek Taskflow, bietet die Möglichkeit für jeden Threadpool die Anzahl der Threads festzulegen. Wie im Abschnitt 2.1.3 beschrieben wurde, hat Aaron Larisch in seiner Masterarbeit experimentell die optimale Anzahl der Thread pro Prozess bei Nao Devils ermittelt. Als Ergebnis hat sich die Kombination aus zwei Thread für den Motion-Prozess und vier Threads für den Cognition-Prozess als optimal herausgestellt. Ob dieses auch für die Prozesse des Nao-Roboters von Berlin United gilt, sollte in Zukunft ermittelt werden. Ebenfalls sollte dieses bei einem eigenen Cognition-Prozess bei Gretchen beachtet werden.

Ebenso sollten zukünftig die Ausführungszeiten der einzelnen Plattformen analysiert werden. Dadurch kann man bestimmen, für welche Plattform die Parallelisierung auf Modul-Ebene Vorteile bietet.

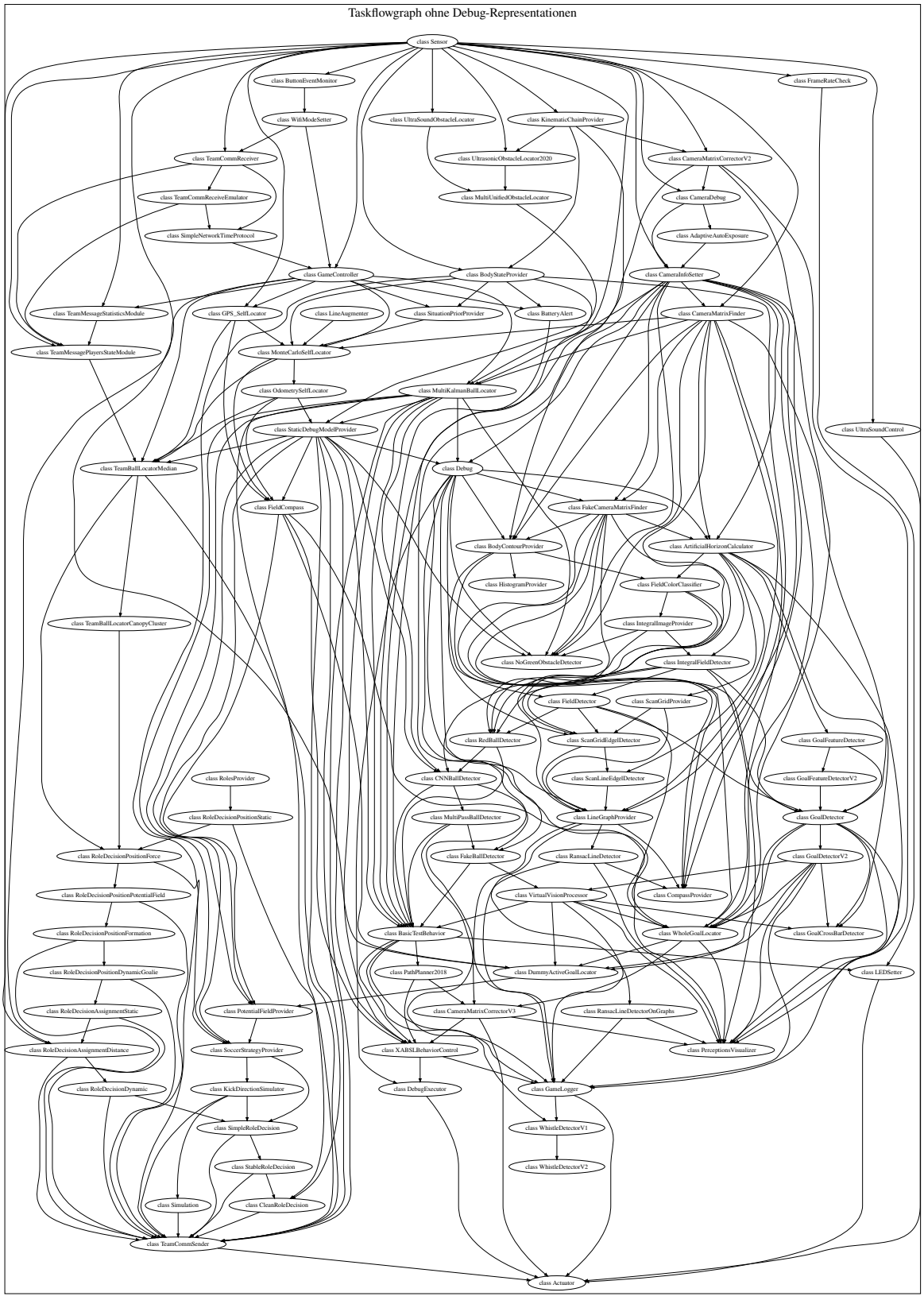

Abbildung 13: Erzeugter Taskflowgraph für den Cognition-Prozess des Nao-Roboters von Berlin United mit Vernachlässigung der Abhängigkeiten der Debug-Representationen. Der Graph kann in digitaler Form genauer betrachtet werden.

Literatur

[1] Petteri Aimonen. *Nanopb - protocol buffers with small code size*. https://jpa.kapsi.fi/nanopb/. besucht am 14.10.2021.

[2] Emil Bergold u. a. *Proseminar Humandoide Roboter, Gretchen - Abschlussbericht*. Sep. 2020.

[3] *B-Human*. https://www.b-human.de/. besucht am 12.11.2021.

[4] C. Bormann und P. Hoffman. *RFC 8949 – Concise Binary Object Representation (CBOR)*. https://www.rfc-editor.org/rfc/rfc8949.html. besucht am 15.10.2021.

[5] Carsten Bormann. *CBOR — RFC 8949 Concise Binary Object Representation*. https://cbor.io/. besucht am 15.10.2021.

[6] SoftBank Robotics Group Corp.("SBRG"). *SoftBank Robotics*. https://www.softbankrobotics.com/. besucht am 03.08.2022.

[7] Martin Donath. *protobluff*. https://squidfunk.github.io/protobluff/. besucht am 14.10.2021.

[8] Boston Dynamics. *ATLAS*. https://www.bostondynamics.com/atlas. besucht am 07.12.2021.

[9] RoboCup Federation. *RoboCup*. https://www.robocup.org/. besucht am 07.12.2021.

[10] Jan Fiedler. "Leichtgewichtiges Multithreading-Framework für Robotikanwendungen". Bachelorarbeit. Universität Bremen, 2019.

[11] Sadayuki Furuhashi. *MessagePack*. https://msgpack.org/. besucht am 08.10.2021.

[12] Sadayuki Furuhashi. *MessagePack specification*. https://github.com/msgpack/msgpack/blob/master/spec.md. besucht am 08.10.2021.

[13] Brian Gerkey. "Why ROS 2?" In: (besucht am 27.10.2021). URL: https://design.ros2.org/articles/why_ros2.html.

[14] Google. *C++ Benchmarks — FlatBuffers, An open source project by FPL*. https://google.github.io/flatbuffers/flatbuffers_benchmarks.html. besucht am 18.10.2021.

[15] Google. *FlatBuffers — Flatbuffers, An open source project by FPL*. https://google.github.io/flatbuffers/. besucht am 18.10.2021.

[16] Google. *FlexBuffers — FlatBuffers, An open source project by FPL*. https://google.github.io/flatbuffers/flexbuffers.html. besucht am 18.10.2021.

[17] *Gretchen – An Open-Source Humanoid Robot Development Platform.n*. https://github.com/aibrainag/Gretchen, besucht am 21.09.2021.

[18] *Gretchen - Documentation*. https://berlinunited.github.io/gretchen.github.io/. besucht am 02.12.2021.

[19] Dr. Tsung-Wei Huang. *Taskflow*. besucht am 04.08.2022. https://taskflow.github.io/: University of Utah, 2019-2022.

[20] Dr. Tsung-Wei Huang. *Taskflow QuickStart*. besucht am 04.08.2022. https://taskflow.github.io/taskflow/pages.html, 2018-2022.

[21] IEEE. *Digit — ROBOTS YOUR GUIDE TO THE WORLD OF ROBOTICS*. https://robots.ieee.org/robots/digit/. besucht am 07.12.2021.

[22] AIBrain Inc. *AIBRAIN, INC — THE INTELLIGENT COMPANY*. https://aibrain.com/. besucht am 10.09.2021.

[23] Google Inc. *Protocol Buffers*. https://developers.google.com/protocol-buffers. besucht am 14.10.2021.

[24] Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI). *Deutsches Forschungszentrum für Künstliche Intelligenz*. https://www.dfki.de/. besucht am 12.11.2021.

[25] Aaron Larisch. "An efficient real-time capable multi-core module framework for the humanoid robot NAO". Masterarbeit. Technische Universität Dortmund, 2020.

[26] Cyberbotics Ltd. *Cyberbotics*. https://cyberbotics.com/. besucht am 06.10.2021.

[27] Cyberbotics Ltd. *Webots User Guide*. https://cyberbotics.com/doc/guide/index. besucht am 08.10.2021.

[28] Steven Macenski u. a. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[29] Heinrich Mellmann. *Humanoider Roboter Gretchen*. https://berlin-united.org/project-gretchen.html. besucht am 25.11.2021.

[30] Heinrich Mellmann u. a. *Berlin United - Nao Team Humboldt Team Report 2018*. Techn. Ber. Adaptive Systeme, Institut für Informatik, Humboldt-Universität zu Berlin, Unter den Linden 6, 10099 Berlin, Germany, 2018.

[31] Heinrich Mellmann u. a. "NaoTH Software Architecture for an Autonomous Agent". In: *Proceedings of the International Workshop on Standards and Common Platforms for Robotics (SCPR 2010)*. Darmstadt, Nov. 2010, S. 316–327. URL: http://www.naoteamhumboldt.de/wp-content/papercite-data/pdf/scpr-mellmannxuetal-10.pdf.

[32] *Nao Devils TU Dortmund*. https://naodevils.de/. besucht am 08.12.2021.

[33] Anastasia Prisacaru. "Gretchen - a humanoid robot for research and education". https://www.naoteamhumboldt.de/wp-content/papercite-data/pdf/2020_prisacaru_bachelorarbeit.pdf. Bachelorarbeit. Humboldt-Universität zu Berlin, 2020.

[34] *RoboCup Standard Platform League*. https://spl.robocup.org/. besucht am 05.06.2022.

[35] Agility Robotics. *AGILITY ROBOTICS*. https://www.agilityrobotics.com/. besucht am 07.12.2021.

[36] Open Robotics. *Documentation — ROS.org*. https://wiki.ros.org/. besucht am 25.10.2021.

[37] Open Robotics. *ROS*. https://www.ros.org/. besucht am 21.10.2021.

[38] SoftBank Robotics. *NAO*. https://www.softbankrobotics.com/emea/de/nao. besucht am 07.12.2021.

[39] SoftBank Robotics. *pepper*. https://www.softbankrobotics.com/emea/en/pepper. besucht am 07.12.2021.

[40] SoftBank Robotics. *Pepper PARLOR*. https://pepperparlor.com/en/. besucht am 07.12.2021.

[41] Thomas Röfer u. a. *B-Human Team Report and Code Release 2019*. Only available online: http://www.b-human.de/downloads/publications/2019/CodeRelease2019.pdf. 2019.

[42] Ingmar Schwarz u. a. *Nao Devils Team Report 2019*. Techn. Ber. https://github.com/NaoDevils/CodeRelease/blob/CodeRelease2019/TeamReport2019.pdf. 2019.

[43] *SimSpark*. http://robocup-sim.gitlab.io/SimSpark//. besucht am 12.11.2021.

[44] *SOFTBANK ROBOTICS DOCUMENTATION*. http://doc.aldebaran.com/2-5/dev/naoqi/index.html, besucht am 15.06.2021.

[45] Chun-Xun Lin Tsung-Wei Huang and, Guannan Guo und Martin Wong. *Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++*. Techn. Ber. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 974-983. https://taskflow.github.io/papers/ipdps19.pdf: Department of Electrical und Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA, 2019.

[46] Berlin United. *Berlin United – Nao Team Humboldt*. https://berlin-united.org/. besucht am 12.11.2021.

[47] Kenton Varda. *CAP'N Proto — cerealization protocol*. https://capnproto.org/. besucht am 15.10.2021.

[48] Wikipedia. *CBOR — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/CBOR. besucht am 15.10.2021.

[49] Wikipedia. *Serialization — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Serialization. besucht am 13.10.2021.

[50] Raphael Winkel u. a. *ws2021-humanoide-roboter*. 2021.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 15. August 2022 .....