Leichtgewichtiges Multithreading-Framework für Robotikanwendungen

Bachelorarbeit

Jan Fiedler

Matrikelnummer: 4235072

18. März 2019

Fachbereich Mathematik / Informatik
Studiengang Informatik

1. Gutachter: Dr.-Ing. Dipl.-Inform. Thomas Röfer
2. Gutachter: Prof. Dr. Rolf Drechsler

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig angefertigt und nicht anderweitig zu Prüfungszwecken vorgelegt habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel verwendet. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 18. März 2019

.....

(Jan Fiedler)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die vorliegende Arbeit entstand im Rahmen des studentischen Projektes B-Human der Universität Bremen in Kooperation mit dem Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI).

1.1 Motivation

Seit 2009 tritt B-Human als Roboterfußballteam bei internationalen Wettbewerben der RoboCup Federation in der Standard Platform League (SPL) an und ist dabei sehr erfolgreich [B-Human, 2018]. Ein Teil des Erfolgs beruht auf einer guten Infrastruktur, die es den jährlich wechselnden Teammitgliedern erlaubt, sich schnell in die Software einzuarbeiten. Die Softwareinfrastruktur basiert auf einem Framework von B-Human selbst (Abschnitt 3.5), welches die Software sehr modular gestaltet. Dies sorgt dafür, dass ein neuer Student sich nur einen kleinen Teil anschauen muss, um einen alten Teil austauschen oder einen neuen Teil für die Software schreiben zu können. Außerdem nimmt das Framework dem Programmierer viele Aufgaben ab, indem diese automatisch umgesetzt werden. Ein Beispiel ist der interne Austausch von Daten. Das Framework wurde ursprünglich 2002 entwickelt und wird seitdem ständig weiterentwickelt. Seit vielen Jahren benutzt B-Human den in der SPL vorgegebenen Roboter *NAO* (Abschnitt 2.3). 2018 wurde eine neue Generation des *NAO* vorgestellt. Dieser besitzt erstmals einen Prozessor mit mehr als einem Rechenkern. Das aktuelle Framework kann dies jedoch nicht ausnutzen, weil es kaum auf Parallelität ausgelegt ist.

Die Motivation dieser Arbeit ist somit, zukünftigen B-Human-Generationen weiterhin ein Framework zu bieten, welches die Leistung des benutzten Roboters ausnutzen kann. Dabei soll das Framework nicht durch ein anderes ersetzt werden, wie zum Beispiel ROS (Abschnitt 4.2), da dies auf dem bisher verwendeten *NAO* zu langsam ist und den Entwicklungsprozess einschränken würde. Es ist daher gewünscht, dass das aktuelle Framework im Kern bestehen bleibt. Eine Herausforderung ist somit, einerseits den Entwicklungsprozess nicht einzuschränken und andererseits ein funktionierendes Ergebnis zu schaffen, mit dem B-Human

weiterarbeiten kann. Zusätzlich muss eine größere Komponente einer komplexen, optimierten und aktiv weiterentwickelten Software verstanden und ausgetauscht werden.

1.2 Ziel

Das Ziel dieser Arbeit ist, den Teil des Frameworks neu zu entwickeln, der sich mit Parallelität beschäftigt. Aktuell gibt es im Framework einige Features, die B-Human schon länger nicht mehr verwendet. Diese sollen bei der Neuentwicklung nicht übernommen werden. Dadurch soll die Komplexität des Frameworks sinken, sodass es einfacher zu verstehen sein wird. Um die angesprochene Parallelität flexibel zu ermöglichen, sollen parallele Ausführungseinheiten in einer Konfigurationsdatei zusammenstellbar sein, sodass verschiedene Konfigurationen benutzbar sind, ohne dass die Software dafür neu kompiliert werden muss. Eine Aufgabe, die bisher automatisch geschieht, ist der Austausch von Daten zwischen parallelen Einheiten. Dieser ist bislang nur auf zwei Einheiten ausgelegt und soll somit neu entwickelt werden, um mit einer variablen Anzahl von parallelen Einheiten umgehen zu können. Das Framework ist dabei ein allgemeines Framework für Robotik, welches in anderen Ligen des RoboCups bereits eingesetzt wurde. Diese Arbeit benutzt B-Human als Grundlage und richtet sich an Anforderungen, die B-Human an das Framework hat.

Nicht Teil dieser Arbeit ist das Ausgliedern des Frameworks aus B-Human, damit es auch in anderen Bereichen eingesetzt werden kann. Auch das Weiterentwickeln der Hauptsoftware um mehr als zwei parallele Ausführungseinheiten ist nicht Teil dieser Arbeit. Allerdings soll in der Evaluation betrachtet werden, welche Fehler es im Konzept gab und wie B-Human mit dem Framework umgeht.

1.3 Aufbau der Arbeit

Zur Einordnung der Arbeit wird in Kapitel 2 zunächst der Hintergrund erläutert. Hierauf folgen die Grundlagen (Kapitel 3), in denen neben grundlegenden Begriffen das Framework von B-Human beschrieben wird. In Kapitel 4 werden andere Frameworks in der SPL sowie das bekannte Framework ROS vorgestellt und mit dem von B-Human verglichen. Das darauf folgende Kapitel 5 erläutert das Konzept und die resultierende Implementierung. Anschließend (Kapitel 6) wird diese hinsichtlich der Zielsetzung der Arbeit evaluiert. Die Arbeit endet mit einem Ausblick (Kapitel 7), in dem reflektiert wird, was in dieser Arbeit erreicht wurde und wo noch Verbesserungspotenzial besteht.

Kapitel 2

Hintergrund

Diese Arbeit entstand im Kontext von B-Human (Abschnitt 2.2) im RoboCup (Abschnitt 2.1). Dort nimmt B-Human in der Standard Platform League (SPL) teil. In dieser spielen baugleiche *NAO*-Roboter (Abschnitt 2.3) gegeneinander Fußball.

2.1 RoboCup

Der RoboCup ist ein seit 1997 jährlich stattfindendes Event der RoboCup Federation [The RoboCup Federation, 2016b], einer internationalen wissenschaftlichen Initiative, um die Forschung im Bereich Künstliche Intelligenz und Robotik zu fördern. Hierzu wurde das Langzeitziel definiert, Mitte des 21. Jahrhunderts mit einem Team autonomer humanoider Roboter gegen den amtierenden FIFA-Weltmeister zu gewinnen [The RoboCup Federation, 2016a]. Bei den Events werden in verschiedenen Ligen Theorien, Algorithmen und Architekturen getestet und evaluiert.

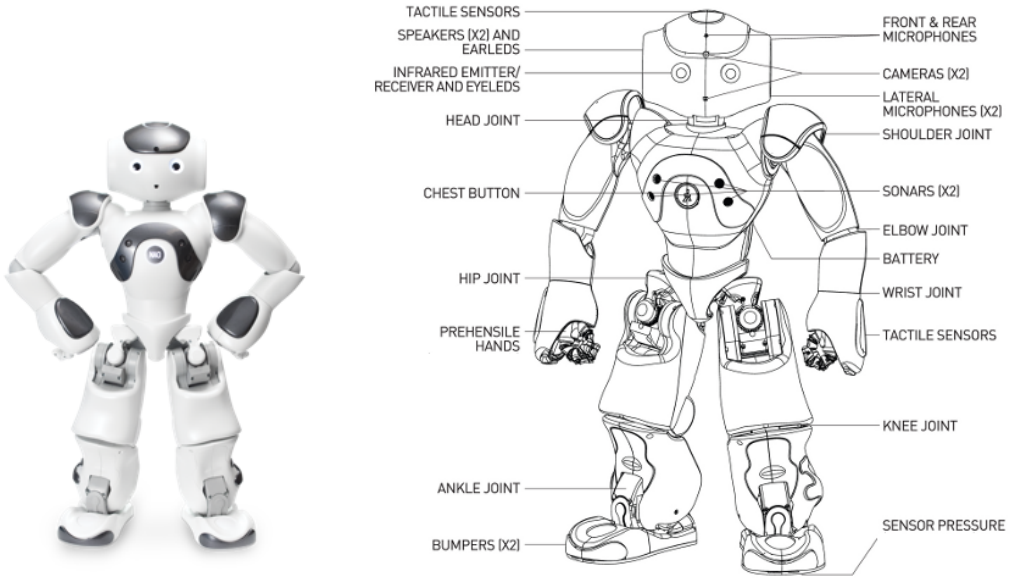
Diese Arbeit ist in der SPL angesiedelt. Dort treten baugleiche Roboter in Teams gegeneinander an. Aktuell spielen *NAO*s in Teams von fünf Robotern gegeneinander. Diese Liga ist ein reiner Softwarewettbewerb, da die verwendeten Roboter nicht modifiziert werden dürfen [The RoboCup Federation, 2016c].

2.2 B-Human

B-Human ist ein größtenteils aus Studenten bestehendes RoboCup-Team der Universität Bremen in Kooperation mit dem Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI). Seit 2009 tritt B-Human regelmäßig bei internationalen Wettbewerben der RoboCup Federation in der SPL an und ist aktuell eines der besten Teams der Liga. Das Team gewann bisher achtmal in Folge die RoboCup German Open, einmal die RoboCup European Open und sechsmal die Weltmeisterschaft [B-Human, 2018].

Außerdem veröffentlicht B-Human jährlich große Teile des Quelltextes zusammen mit einer vollständigen Beschreibung [Röfer u. a., 2017] oder den Änderungen zum Vorjahr [Röfer u. a., 2018]. Dies dient zum Austausch der Forschungsergebnisse mit anderen Teams und kann von neuen Teams als Grundlage für den Einstieg genutzt werden.

2.3 Der *NAO*



(a) Der *NAO V6*. (Entnommen aus [SoftBank Robotics, 2018e])

(b) Sensoren und Gelenke des *NAO*. (Entnommen aus [SoftBank Robotics, 2018c])

Abbildung 2.1 Der *NAO*, seine Gelenke und Sensoren.

Der in der SPL eingesetzte Roboter ist der *NAO* (Abbildung 2.1a). Dieser ist ein humanoider Roboter, der von der Firma Softbank Robotics produziert wird. Er wird stetig weiterentwickelt. Die bisher aktuellste Version ist der *NAO V5*, der zum Zeitpunkt der Arbeit vom *NAO V6* abgelöst wird. Beide sind 58 cm groß und wiegen ungefähr 5,4 kg. Sie besitzen 25 steuerbare Freiheitsgrade und diverse Sensorik (Abbildung 2.1b). Die beiden Kameras stellen jeweils 30 Bilder pro Sekunde bereit. Auf dem *NAO V5* werden die Gelenke und Sensoren 100-mal pro Sekunde aktualisiert, auf dem *NAO V6* nur noch 83-mal [SoftBank Robotics, 2018a,b,e].

Der *NAO V5* besitzt eine Intel ATOM Z530 Single Core CPU mit 1,6 GHz, sowie 1 GB RAM, 2 GB Flashspeicher und eine 8 GB Micro SDHC auf dem Motherboard [SoftBank Robotics, 2018d]. Der neue *NAO V6* hingegen besitzt eine Intel ATOM E3845 Quad Core CPU mit 1,91 GHz, sowie 4 GB DDR3 RAM und 32GB eMMC Flashspeicher [SoftBank Robotics, 2018f]. Dieser ist somit um ein vielfaches leistungsstärker als sein Vorgänger und eine Motivation (Abschnitt 1.1) für diese Arbeit.

Kapitel 3

Grundlagen

Dieses Kapitel beschreibt Grundlagen, welche für den weiteren Verlauf der Arbeit vorausgesetzt werden. Es werden zunächst die Begriffe Software-Framework (Abschnitt 3.1), sequenzielle und parallele Ausführung (Abschnitt 3.2), Prozesse und Threads (Abschnitt 3.3) sowie synchrone und asynchrone Kommunikation (Abschnitt 3.4) erläutert. Anschließend wird in Abschnitt 3.5 das bisherige Framework von B-Human beschrieben.

3.1 Software-Framework

Ein Software-Framework ist eine Rahmenstruktur, genauer ein Programmiergerüst, welches insbesondere in objektorientierter und komponentenbasierter Softwareentwicklung eingesetzt wird. Es gibt somit die Anwendungsarchitektur vor. Dies führt zu einer Umkehrung der Steuerung: Programmierer registrieren ihre Implementierungen, die das Framework steuert und benutzt. Das ist ein Gegensatz zu Klassenbibliotheken, aus denen meistens lediglich Funktionen aufgerufen werden. Ein Framework definiert auch den Kontrollfluss und die Schnittstellen der Implementierung. Dazu gehören häufig auch die Kommunikation und Schnittstellen nach außen. Damit ist es aber kein vollständiges Programm. Das Programm wird in das Framework implementiert. Meistens ist ein Framework dabei auf eine Domäne oder Anwendungstypen beschränkt [Wikipedia, 2018a; Edwin, 2014].

3.2 Sequenzielle und parallele Ausführung

In der Informatik beruht die sequenzielle Ausführung auf dem „Von-Neumann-Rechner“. Diesem liegen als mathematisches Modell deterministische endliche Automaten zugrunde. Sie besitzen als Merkmale unter anderem, dass ein Zustand genau einen deterministischen Folgezustand besitzt. Somit findet die Ausführung eines Programmes immer von oben nach unten und von links nach rechts statt, wobei keine parallelen Aktionen erlaubt sind. Parallele Ausführung beschreibt das gleichzeitige Ausführen von mehr als einer sequenziellen

Ausführungsfolge [Zöbel u. Hogenkamp, 1988].

3.3 Prozesse und Threads

Ein Prozess ist grundsätzlich ein Programm in Ausführung.

„Genauer ist ein Prozess die konkrete Instanziierung eines Programms zu dessen Ausführung innerhalb eines Rechnersystems, ergänzt um weitere (Verwaltungs-)Informationen und Ressourcenzuteilungen des Betriebssystems für diese Ausführung.“ [Wikipedia, 2018b]

Dem Programm stehen dabei die kompletten Möglichkeiten des Betriebssystems zur Verfügung. Zum Zeitpunkt der Ausführung erhält ein Prozess somit die volle Aufmerksamkeit der CPU. Allerdings ist die CPU-Zeit eines Prozesses begrenzt, da es auf einem Computer meistens mehr als einen Prozess gleichzeitig gibt. Um diese Zuteilung kümmert sich das Betriebssystem und ermöglicht somit Multitasking. Oberflächlich betrachtet besteht kein Unterschied zwischen einem Prozess und einem Thread, denn letztendlich besteht ein Prozess mindestens aus einem Thread. Allerdings können Threads nur innerhalb eines Prozesses ausgeführt werden. Während jeder Prozess seine eigene Aufgabe abarbeitet und dabei zunächst unabhängig von anderen Prozessen ist, teilen sich Threads innerhalb eines Prozesses dessen Ressourcen. Wenn ein Thread abstürzt, kann dies zur Folge haben, dass alle anderen Threads eines Prozesses auch abstürzen. Dieses Problem besteht bei Prozessen nicht. Da Threads sich Ressourcen teilen, können diese einfacher zusammenarbeiten, sind aber gegebenenfalls aufwendiger zu programmieren, da sie unbefugte Zugriffe – im Gegensatz zu Prozessen – selbst schützen müssen [Wolf, 2009].

3.4 Synchrone und asynchrone Kommunikation

Als Kommunikation wird in der Informatik das gegenseitige Senden und Empfangen von Daten bezeichnet. Bei synchroner Kommunikation werden die Daten zwischen Sender und Empfänger immer synchronisiert und die Kommunikationspartner blockieren, bis die Kommunikation abgeschlossen ist. Ein Sender sendet also etwas und wartet, bis er eine Antwort erhalten hat, bevor er weiter seine Berechnungen fortführt. Bei asynchroner Kommunikation hingegen geschieht das Senden und Empfangen zeitlich versetzt und die Kommunikationspartner können währenddessen andere Aufgaben erledigen – blockieren also nicht [Tanenbaum u. Bos, 2016].

Letzteres spielt im Kontext von Robotikframeworks eine besonders große Rolle, da diese häufig in Echtzeit agieren und somit auf eine schnelle und effektive Datenübertragung angewiesen sind. So auch im Framework aRDx vom Deutschen Zentrum für Luft- und Raumfahrt (DLR). Hier wurde die Kommunikation in mehrere Schichten aufgeteilt. Im Folgenden wird die Kommunikation innerhalb eines Prozesses und zwischen Prozessen auf einem Computer beschrieben. Innerhalb eines Prozesses wird Speicher auf dem Heap reserviert und beschrieben. An-

schließend bekommt der Empfänger eine Benachrichtigung und kann den Speicher auslesen. Dabei können mehrere Pakete zwischengespeichert werden, wobei immer das Älteste benutzt wird. Dieser Ansatz ist sehr ähnlich zu dem, den B-Human verwendet (Abschnitt 3.5.2). Für die Kommunikation zwischen Prozessen wird im Prinzip der gleiche Mechanismus genutzt. Damit Prozesse weiterhin unabhängig voneinander bleiben und nicht einer alle anderen zum Abstürzen bringen kann, gibt es einen weiteren Prozess, welcher für die Kommunikation zuständig ist. Die Nachricht wird dabei in dessen Speicher geschrieben und ist somit genauso effizient wie die Kommunikation innerhalb eines Prozesses, da die Nachricht nicht kopiert werden muss [Hammer u. Bäuml, 2013].

3.5 B-Human-Framework

Die Aufgabe eines Frameworks für B-Human besteht darin, Softwarekomponenten zu gruppieren und dabei sowohl sequenzielles als auch paralleles Ausführen in einem vorgegebenen Takt zu ermöglichen. Mit Takt ist gemeint, dass die eingehenden Daten bearbeitet sein müssen, bevor neue Daten bereitgestellt werden. Ein solches System wird als Echtzeitsystem bezeichnet [Shin u. Ramanathan, 1994]. Echtzeit spielt zwar primär für die Softwarekomponenten eine Rolle, da diese die meiste Rechenzeit beanspruchen. Trotzdem sollte auch das Framework versuchen, nur so viel Rechenzeit wie nötig zu beanspruchen.

Das Framework wurde Anfang der zweitausender Jahre entwickelt und wird seitdem durchgehend weiterentwickelt. Es ist dadurch mit der Zeit gewachsen und nicht immer aufgeräumt worden. Deshalb geht dieses Kapitel zuerst auf die Entstehung des B-Human-Frameworks (Unterabschnitt 3.5.1) ein. Im Anschluss wird der Aufbau (Unterabschnitt 3.5.2) des 2018 benutzten Frameworks dargestellt. Abschließend werden Anwendungsszenarien (Unterabschnitt 3.5.3) erläutert.

3.5.1 Entstehung

Ursprünglich wurde das Framework 2002 vom Team GermanTeam entwickelt. Das GermanTeam nahm am RoboCup in der Sony Legged Robot League teil. In dieser spielten eine Art Roboterhunde, sogenannte AIBOs [Fujita u. a., 1999], gegeneinander Fußball. Die AIBOs hatten eine Kamera und das Betriebssystem kannte nur Prozesse, keine Threads. Das GermanTeam bestand dabei aus mehreren Teams, die auf nationalen Wettbewerben gegeneinander antraten, international aber als ein Team. Teil des Teams waren dabei fünf Universitäten: Die Humboldt-Universität zu Berlin, die Universität Bremen, die Technische Universität Darmstadt, die Universität Dortmund, und die Freie Universität Berlin. Das Hauptziel der Architektur war es somit, zu ermöglichen, dass all diese Universitäten ihre eigene Software entwickeln und diese effizient zu einer Software kombinieren konnten, um an lokalen sowie internationalen Wettbewerben teilnehmen zu können. Dazu wurde eine Infrastruktur definiert, die dieses erlaubt. Jede Aufgabe wurde dabei jeweils von einem sogenannten Modul erledigt.

Damit diese zwischen Teams ausgetauscht werden konnten, wurden feste Schnittstellen definiert (Abbildung 3.1). Die Datenklassen, die Module austauschen, heißen Repräsentationen. Um die Module auch vergleichen zu können, war es möglich, diese zur Laufzeit zu wechseln. Unabhängig von diesen Schnittstellen konnte jedes Team seine Module je nach Anforderungen in mehrere Prozesse gruppieren, um sowohl sequenzielles als auch paralleles Ausführen zu ermöglichen. Dies resultierte in verschiedenen Prozesslayouts. Das Teilteam Bremen Byters (Abbildung 3.2) versuchte beispielsweise, so viel wie möglich zu parallelisieren, während die Humboldt-Universität (Abbildung 3.3) ein einfacheres Layout benutzte. Letzteres wurde auch auf dem RoboCup verwendet, da es insgesamt schneller war als das der Bremen Byters. Dies lässt sich unter anderem auf einen geringeren Kommunikationsoverhead zurückführen. Außerdem wurden einige Prozesse der Bremen Byters zur Laufzeit sequenziell ausgeführt, da sie voneinander abhingen.

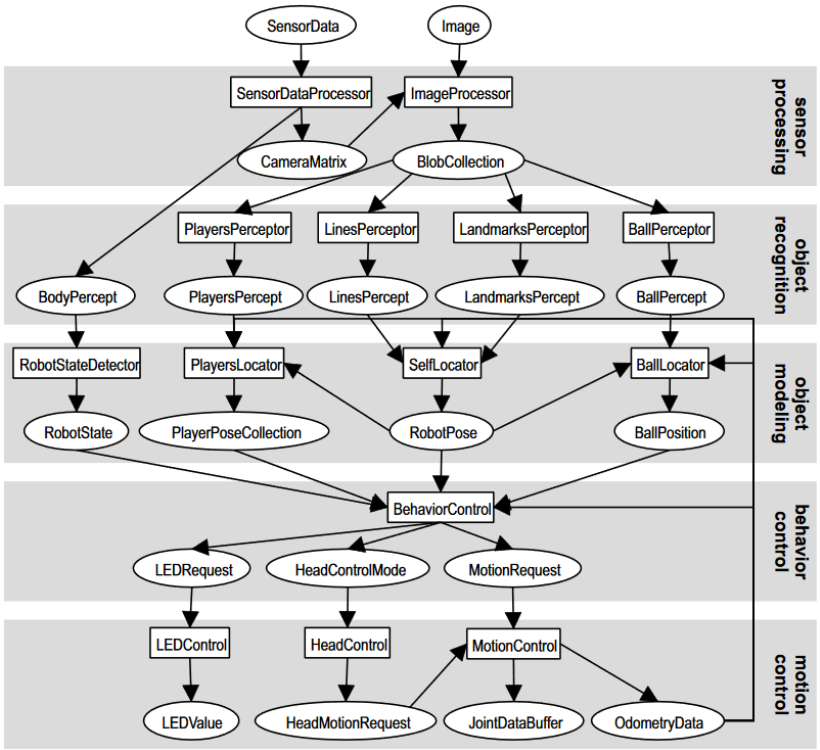


Abbildung 3.1 Struktur der Datenverarbeitung im GermanTeam.
(Entnommen aus [Burkhard u. a., 2002, S. 11])

Innerhalb eines Prozesses fand der Datenaustausch über ein sogenanntes Blackboard [Jagannathan u. a., 1989] statt. Daten zwischen Prozessen und zum Debuggen wurden gleich behandelt. Dabei tauschten Sender und Empfänger sogenannte Pakete aus. Dies wurde über Betriebssystemmechanismen realisiert. Die Kommunikation fand dabei im gleichen Takt wie die Ausführung der Module statt. Am Anfang eines jeden Zyklus erhielten alle Empfänger eines Prozesses alle aktuellen Pakete. Dann wurden alle normalen Berechnungen durchgeführt, wobei bereits Pakete versendet werden konnten. Am Ende wurden alle Pakete versendet, die vorher noch nicht versendet wurden. Dabei ist anzumerken, dass immer nur die neuesten

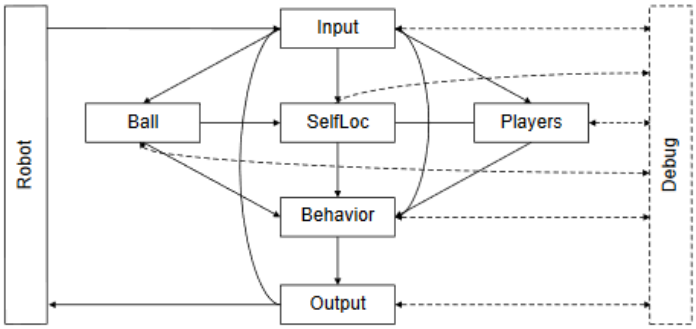


Abbildung 3.2 Das Prozesslayout der Bremen Byters.
(Entnommen aus [Burkhard u. a., 2002, S. 13])

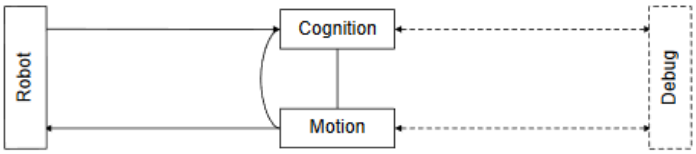


Abbildung 3.3 Das Prozesslayout der Humboldt-Universität.
(Entnommen aus [Burkhard u. a., 2002, S. 14])

Daten empfangen wurden. Hat also ein Prozess zwei Zyklen gemacht, während ein anderer nur einen geschafft hat und dieser bekommt Daten von dem anderen, bekommt er nur das neueste Paket und das vorherige geht verloren. Die ganze Kommunikation geschah dabei komplett automatisch und es reichte, die Verbindungen zwischen Sendern und Empfängern festzulegen. Dies ermöglichte eine einfachere Implementierung neuer Prozesse [Röfer, 2003; Burkhard u. a., 2002, S. 10-15].

Ein weiteres Ziel der Architektur war die Plattformunabhängigkeit. So konnte die Software auf dem AIBO laufen, aber auch unter Windows und dem Open-R Emulator mit Cygwin. Unter Windows wurden außerdem statt Prozessen Threads verwendet [Burkhard u. a., 2002, S. 6-8].

Dieses Framework stellte sich jedoch nach einigen Jahren als nicht optimal heraus und wurde deshalb für den RoboCup 2007 generalüberholt. Ein Problem war unter anderem, dass die Modulkonfiguration sehr statisch war. Somit musste jede neue Idee in die alte Struktur eingebaut werden. Da verschiedene Lösungen eines Problems verschiedene Anforderungen hatten, wurden die Schnittstellen groß. Außerdem haben verschiedene Module ihre Ergebnisse in unterschiedlicher Form bereitgestellt und somit war es in der Praxis doch nicht möglich, alle Module beliebig zu kombinieren.

Um diese Probleme zu vermeiden, wurde das Framework neu entwickelt. Das neue Framework bestand wieder aus einem Blackboard für die Kommunikation innerhalb eines Prozesses. Die Kommunikation mit den Schnittstellen zum Debugging und zwischen Prozessen ist nicht verändert worden. Die Module waren jetzt aber nicht mehr an statische Schnittstellen gebunden, sondern frei konfigurierbar. Ein Modul konnte verschiedene Repräsentationen bereitstellen und erwarten. Dabei musste ein Modul zur Laufzeit nicht alle Repräsentationen bereitstellen,

die es bereitstellen konnte. Weiterhin konnte es für jede Repräsentation mehrere Module geben, die diese bereitstellen könnten. Aus diesen Informationen konnte dann eine Reihenfolge ermittelt werden, in der die Module die Repräsentationen berechnen mussten. Der Grund dafür ist, dass alle erwarteten Repräsentationen berechnet sein mussten, bevor das Modul seine bereitzustellenden Repräsentationen berechnen konnte. Dies führte dazu, dass viele Module neu implementiert wurden, um die Software schlanker und modularer zu gestalten. Ein Indiz dafür, dass dies eine gute Entscheidung war, gab es direkt bei den RoboCup German Open 2007, welche gegen den Weltmeister gewonnen wurden [Röfer u. a., 2007].

Das Framework hat sich auch in anderen Ligen des RoboCups bewiesen und wurde deshalb auch 2008 vom neuen Team B-Human in der Humanoid League benutzt, in der humanoide Selbstbauroboter gegeneinander Fußball spielen. Im Jahr 2009 wechselte das Team dann in die SPL. Da das Framework des GermanTeams möglichst hardwareunabhängig war, war es nicht übermäßig schwer, die Software erst auf die humanoiden Roboter und dann auch auf die *NAO*s zu portieren. Seit 2011 heißt die Software offiziell B-Human-Framework und mit Microsoft Windows, Mac OS und Linux werden alle drei großen Plattformen zum Entwickeln unterstützt. Seit dem Umstieg auf den *NAO* sind Prozesse auf allen Plattformen als Threads implementiert, da der *NAO* dies im Gegensatz zu den vorherigen Robotern unterstützt. Der Begriff Prozess wird allerdings weiterhin aus Konsistenzgründen verwendet [Röfer u. a., 2008; Röfer u. Laue, 2014]. Dieses Framework benutzt B-Human mit kleinen Anpassungen bis heute und ist damit immer noch sehr erfolgreich [Röfer u. a., 2017, S. 28-45].

3.5.2 Aufbau

Im Folgenden wird der Teil des B-Human-Frameworks beschrieben, der für diese Arbeit relevant ist. Dabei wird weiterhin der Begriff Prozess verwendet. Mit diesem Framework hat B-Human 2018 am RoboCup teilgenommen [Röfer u. a., 2018]. Die Software ist in C++ geschrieben und lässt sich grob in zwei Teile aufteilen. Zuerst wird der Aufbau der Software beim Programmstart und die Prozesse als Threads beschrieben (Abbildung 3.5). Anschließend werden die Kommunikation und die Prozesse dargestellt (Abbildung 3.8). Die expliziten C++-Klassen werden hier hervorgehoben.
B-Human besitzt auch einen Simulator, mit dem es unter anderem möglich ist, Roboter zu simulieren oder sich mit einem echten Roboter zu verbinden. Dieser wird in der Arbeit verwendet.

Programmstart

Beim Programmstart (Abbildung 3.4) registrieren sich die Prozesse selbstständig in einer Liste, indem jeder `ProcessCreator` sich in dieser registriert. Hierzu wird ausgenutzt, dass globale Objekte zum Programmstart erzeugt werden. Das Registrieren wird über das C++-Makro `MAKE_PROCESS` realisiert, welches in einen `ProcessCreator` umgewandelt wird. Da diese in

Quelldateien erstellt werden, sind sie trotzdem gekapselt und nicht öffentlich sichtbar. Um das Framework zu starten, wird ein `Robot` erstellt. Dieser unterscheidet sich je nachdem, ob er auf einem *NAO* oder im Simulator gestartet wird: Der simulierte Roboter besitzt zusätzlich noch Funktionalitäten für den Simulator, die hier nicht genauer betrachtet werden. Er liest dabei eine Datei mit allen Verbindungen zwischen Prozessen ein und verbindet diese entsprechend. Ein Roboter ist dabei eine Liste von `ProcessBase`-Zeigern. Diese Liste läuft beim Erstellen eines Roboters die vorher beschriebene Liste aller zu erstellenden Prozesse durch und erstellt diese. `ProcessList` bietet die Möglichkeit, alle Prozesse zu starten und zu stoppen.

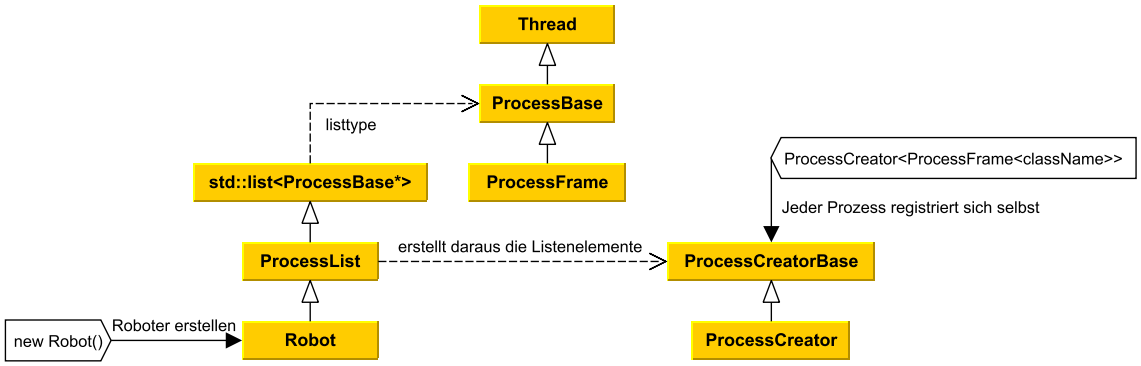


Abbildung 3.4 Das Klassendiagramm beschreibt den Aufbau des zum Starten relevanten Teils des Frameworks. Die UML-Notation wurde um Funktionen und Member der Klassen reduziert, da diese in Abbildung 3.5 beschrieben werden. Die Klassen aus Abbildung 3.8 sind gar nicht dargestellt. Dafür ist der Einstiegspunkt in die Software und das Registrieren der Prozesse dargestellt.

Prozesse als Threads

Der plattformabhängige Teil eines Prozesses ist ein `ProcessFrame`, welcher einen generischen Member hat, der die Implementierung von Prozessen im B-Human-System enthält. Somit ist dieser unabhängig davon, ob Prozesse oder Threads verwendet werden. Bei B-Human ist das `PlattformProcess`. Der nicht private Teil von `PlattformProcess` stellt auch das von `ProcessFrame` erwartete Interface dar. `ProcessFrame` hat außerdem eine `main()`-Funktion, welche die Hauptschleife eines Prozesses ist. Diese Klasse erbt von `Thread`. Sie implementiert gekapselte plattformabhängige `std::thread`s. Plattformabhängig sind sie, weil Features benutzt werden, die `std::thread` nicht unterstützt und somit für jede Plattform implementiert sind. Die hier beschriebene Abhängigkeit bezieht sich nur auf Prozesse. Abhängigkeiten zur Roboterhardware, wie zum Beispiel Kameras, Gelenke oder Sensoren, sind in Modulen umgesetzt und gekapselt, sodass bei neuer Hardware nur diese ausgetauscht werden müssten. Dies ist allerdings nicht weiter relevant für diese Arbeit und wird deshalb im weiteren Verlauf hinsichtlich der Plattformabhängigkeit nicht betrachtet, da dieser Teil des Frameworks nicht verändert wird.

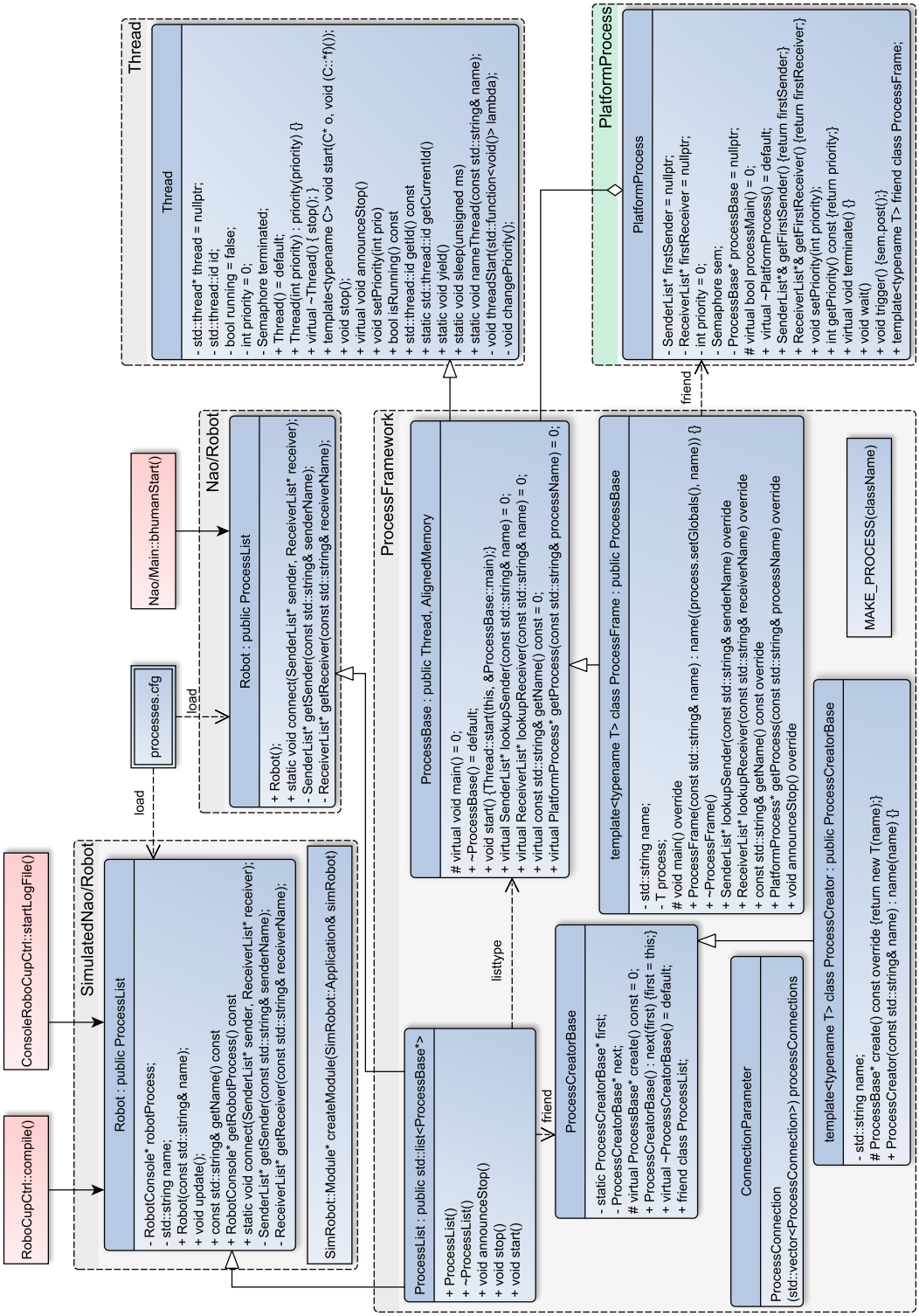


Abbildung 3.5 Das Klassendiagramm des als Threads implementierten Prozessframeworks. Die UML-Notation wurde dazu angepasst. Die äußeren Kästen beschreiben hierbei Dateien und die zweigeteilten Kästen stellen Klassen darin dar. Einfache Kästen beschreiben Besonderheiten. Die Kästen außerhalb der Dateien sind Einstiegspunkte in die Software. Doppelt umrandete Kästen sind Dateien, die eingelesen werden. Funktionen ohne Semikolon am Ende bedeuten, dass diese im Interface implementiert sind.

Kommunikation

Die Kommunikation zwischen Prozessen funktioniert bei B-Human über gemeinsam genutzten Speicher. Dabei reserviert der sendende Prozess Speicher und schreibt in diesen die Repräsentationen, die gesendet werden sollen. Anschließend wird der Speicher an den Empfänger übergeben und dieser darüber informiert. Da das Übergeben eine atomare Operation ist und erst Schreiben, dann Übergeben und anschließend Lesen ein asynchroner Ablauf ist, ist an dieser Stelle keine weitere Synchronisation nötig. Realisiert ist die Kommunikation mit den Klassen `Sender` und `Receiver`. Diese sind in Listen organisiert. Jeder `Sender` hat eine Liste von `Receivern`, denen er etwas senden kann. Auf der `SenderList` ist auch eine Funktion definiert, die das Senden aller Pakete von allen Sendern ermöglicht. Außerdem gibt es noch eine Klasse `DebugSender`. Diese hat einen Parameter für den Simulator. Weiterhin wird ein Paket nur dann gesendet, wenn es nicht leer ist. Das Senden kann blockierend sein, bis ein neues Paket angefordert wird. Jeder `Receiver` kann auf ein neues Paket prüfen, welches auch direkt eingelesen wird, falls es existiert. Er kann bis zu drei Pakete zwischenspeichern. Dies sorgt für eine verklemmungsfreie (deadlockfreie) Kommunikation, da gleichzeitig maximal ein Paket gelesen, ein anderes geschrieben und ein drittes auf das Lesen warten kann. Ansonsten wird ein Paket überschrieben, das gerade weder wartet noch gelesen wird. Gelesen wird immer das neueste fertig geschriebene Paket. Dadurch ist es zwar möglich, dass Pakete verloren gehen, aber B-Human interessieren immer nur die aktuellsten Informationen, da immer auf die neuesten Ereignisse reagiert werden soll. Die `ReceiverList` kann für alle `Receiver` in der Liste prüfen, ob sie ein neues Paket haben und dieses dann einlesen.

Pakete sind ein generischer Typ. Aktuell ist ein Paket entweder ein `ModulePackage` oder eine `MessageQueue` (Abbildung 3.6). `ModulePackage`s werden für die Interprozesskommunikation verwendet. Dabei schreibt und liest der `ModuleManager` die Repräsentationen vom Paket. `MessageQueue`s werden zum Debuggen verwendet. Der `Debug`-Prozess verwendet sie für die Kommunikation mit den Prozessen und zur Kommunikation mit dem Simulator. In einer `MessageQueue` befinden sich Nachrichten, deren Typ über eine Kennung eindeutig ist.

Prozesse

Jeder Prozess wird in einen `ProcessFrame` eingehängt und ist somit ein eigener Thread. Die einzige Schnittstelle zu dem bereits beschriebenen Teil bildet `PlatformProcess`. Dieser implementiert dabei die Schnittstelle. Außerdem verwaltet er die Sender- und Empfängerlisten. Von diesem erbt die Klasse `Process`. Sie besitzt alle prozessweiten Objekte, wie das Blackboard und diverse Debuggingfeatures. Von ihr erben bei B-Human die drei Prozesse `Cognition`, `Motion` und `Debug`. `Debug` ist dabei anders als die anderen beiden Prozesse, da es keine Module ausführt. Er ist für die Kommunikation zwischen Prozessen und dem Simulator zuständig. Hierzu nimmt er Pakete mit Nachrichten vom Simulator entgegen und leitet diese an die passenden anderen Prozesse weiter. Umgekehrt nimmt er die Antworten der Prozesse wieder entgegen und schickt diese zurück an den Simulator. Nachrichten können

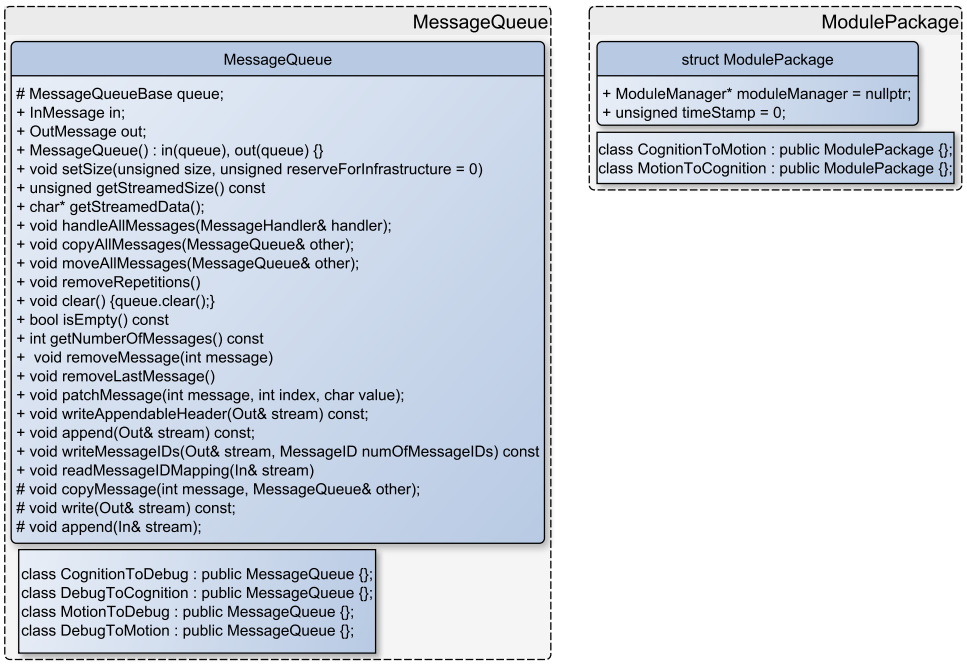


Abbildung 3.6 Das Klassendiagramm der Pakettypen. Die UML-Notation wurde dazu angepasst. Die äußeren Kästen beschreiben hierbei Dateien und die zweigeteilten Kästen stellen Klassen darin dar. Funktionen ohne Semikolon am Ende bedeuten, dass diese im Interface implementiert sind. Außerdem sind die Aliasse angegeben. Diese werden dazu benutzt, um zu verdeutlichen, wozu die Kommunikationsverbindung zuständig ist.

dabei zum Beispiel Anfragen von Daten sowie Änderungen der Module oder Konfigurationen sein. Bei Wettkampfspielen ist `Debug` allerdings nicht aktiv – einerseits um Rechenzeit zu sparen, und andererseits, da ohnehin kein Simulator angeschlossen ist und somit keine Daten ausgetauscht werden. `Motion` und `Cognition` sind die eigentlichen verarbeitenden Prozesse. Diese laufen im Takt der Sensoren: `Motion` im Takt der Gelenke und `Cognition` im Takt der Kameras. Sie besitzen jeweils eine Kommunikationsschnittstelle zu `Debug` und können sich gegenseitig Pakete schicken. In der `main()`-Funktion ist jeweils der Ablauf eines Zyklus implementiert, in dem auch die Module ausgeführt werden. Dabei ist es möglich zu loggen, welchen Inhalt Repräsentationen nach der Ausführung der Module haben, damit die Prozesse zum Debuggen später wiedergegeben werden können. Die Module und die Reihenfolge, in der sie ausgeführt werden, bestimmt der `ModuleManager`. Die beiden Prozesse haben jeweils ihren eigenen. Zusätzlich kümmert sich `Cognition` noch um die Kommunikation zwischen Robotern und mit dem Schiedsrichter.

ModuleManager

Der `ModuleManager` (Abbildung 3.7) entscheidet anhand von Kategorien, zu denen sich jedes Modul zuordnet, welche Module er ausführt. Welche Module ausgeführt werden sollen, ist in einer Konfigurationsdatei beschrieben (Quelltext 3.1). Er kennt dabei alle existierenden

Module und kann daraus berechnen, in welcher Reihenfolge seine Module ausgeführt werden müssen. Welche Repräsentationen er von dem anderen Prozess braucht und welche dieser von ihm braucht, wird auch berechnet. Hierbei wird auch geprüft, ob daraus eine valide Konfiguration erstellt werden kann. Der `ModuleManager` kann dabei mit zwei Prozessen sinnvoll umgehen, weil er zwischen sich und dem Rest unterscheidet. Dies lässt zwar mehr als einen anderen Prozess zu, führt aber dazu, dass er von allen anderen die gleichen Pakete erwartet und verschickt. Außerdem kümmert er sich um das Ausführen der Module.

```
representationProviders = [
  {representation = Representation; provider = Provider;},
  {representation = Representation2; provider = Provider2;},
];
```

Quelltext 3.1 Aufbau der Konfigurationsdatei für alle bereitgestellten Repräsentationen und von welchem Modul sie bereitgestellt werden. Die Schreibweise entspricht der von Konfigurationsdateien bei B-Human.

Über den Simulator können neue Konfigurationen eingespielt werden. Dabei ist es neben dem Umschalten der Module, die eine Repräsentation bereitstellen, auch möglich, das Bereitstellen einer Repräsentation abzuschalten. Es kann allerdings sein, dass ein anderes Modul sie erwartet. Um diesen Konflikt zu lösen und eine valide Modulkonfiguration ausrechnen zu können, gibt es ein Modul „default“. Repräsentationen, die aktuell kein Modul bereitstellt, aber von einem Modul benötigt werden, können damit trotzdem für die Berechnung angegeben werden. Somit ist es möglich, eine valide Modulkonfiguration auszurechnen. „default“ gibt an, dass die Repräsentation mit Standardwerten erstellt wird und dass diese Werte sich nicht ändern.

3.5.3 Anwendungsszenarien

In diesem Kapitel wird dargestellt, wofür das Framework noch eingesetzt wird und insbesondere, wie die Modulkonfigurationen verwendet werden. Das Framework wird neben der Robotersoftware auch im Simulator eingesetzt. Dort hat jeder simulierte Roboter seinen eigenen Thread, mit dem er kommuniziert. Dies gilt auch, wenn der Simulator mit einem echten Roboter verbunden ist.

Ein wichtiges Instrument im Framework sind Modulkonfigurationen. Diese werden vielseitig für verschiedene Aufgaben eingesetzt. Zuallererst geben sie an, welche Repräsentationen auf dem Roboter bereitgestellt werden und von wem. Es ist möglich, diese zur Laufzeit auszutauschen und somit Module direkt vergleichen zu können, ohne die Software neu bauen zu müssen oder sie neu auf den Roboter zu spielen. Die neue Konfiguration wird dabei vom Simulator an den Roboter geschickt.

Das Wiedergeben von Logs ist auch darüber realisiert. In einem Log stehen für jeden Prozess für jeden Zyklus alle Repräsentationen, die geloggt wurden. Diese werden von einem Modul

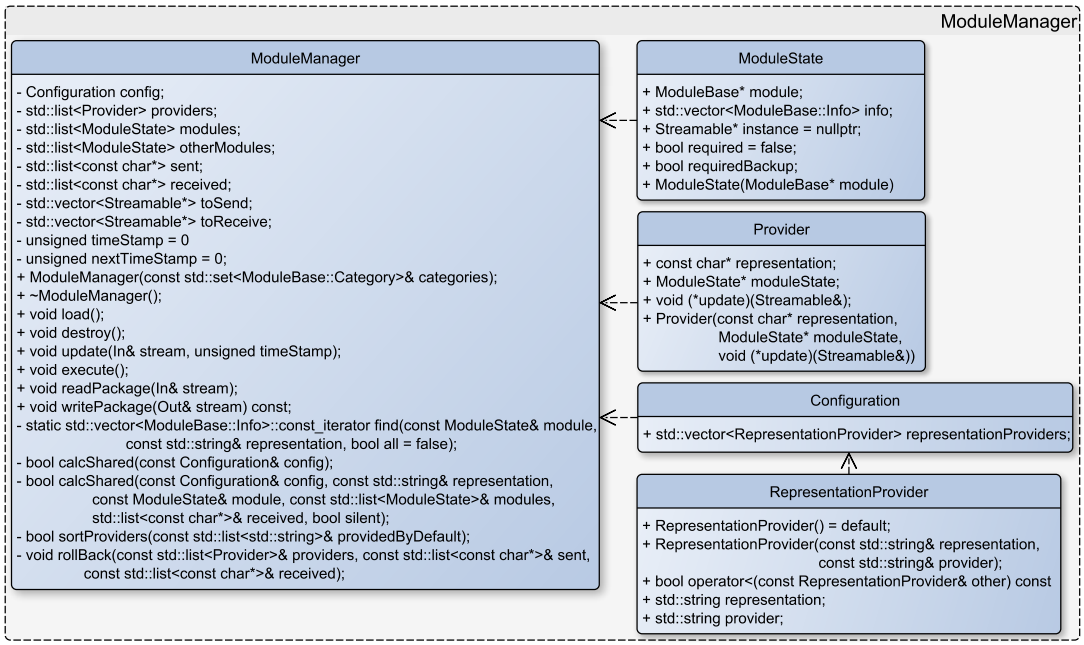


Abbildung 3.7 Das Klassendiagramm des `ModuleManager`. Die UML-Notation wurde dazu angepasst. Die äußeren Kästen beschreiben hierbei Dateien und die zweigeteilten Kästen stellen Klassen darin dar. Da das verwendete Tool die UML-Notation für innere Klassen nicht kennt, wird hier die Abhängigkeit dafür verwendet, um dies darzustellen.

bereitgestellt und die Konfiguration automatisch so umgestellt, dass der simulierte Roboter diese bekommt und damit seine normalen Berechnungen durchführt. Dadurch haben auch alle Repräsentationen, die sich direkt aus diesen berechnen lassen, wieder ihre originalen Werte, sofern sich die Module nicht verändert haben. Damit lassen sich aber auch veränderte oder neue Module mit echten Daten testen. Allerdings ist dies dadurch beschränkt, dass das Log zum Beispiel andere Entscheidungen im Verhalten mit seinen Daten überschreibt. Es eignet sich aber sehr gut, um Module der Bildverarbeitung oder der Modellierung zu testen. Ein simulierter Roboter, der kein Log wiedergibt, funktioniert dabei nach dem gleichen Prinzip. So werden Repräsentationen, die normalerweise Hardwaredaten bereitstellen, umgeschaltet, sodass sie ein anderes Modul mit Daten aus der Simulation füllt.

Eine weitere Anwendung ist, dass es sehr aufwändig ist, mehrere Roboter oder ganze Spiele zu simulieren und viele Computer nicht genug Rechenleistung haben, die Roboter in Echtzeit zu simulieren. Dafür gibt es im Simulator die Möglichkeit, dass entweder die Berechnung der Bilder oder zusätzlich auch die Modellierung nicht berechnet werden, sondern der Simulator die Daten direkt setzt. Somit ist es möglich, viel Rechenzeit zu sparen. Dies ermöglicht zum Beispiel ein effektiveres Testen von Verhalten. Insbesondere kann auch dann Verhalten getestet werden, wenn Teile der Bildverarbeitung oder Modellierung noch nicht implementiert oder nicht gut genug implementiert sind. Da all dies automatisch geschieht, muss ein Nutzer nicht alles verstehen und kann die Software einfach benutzen.

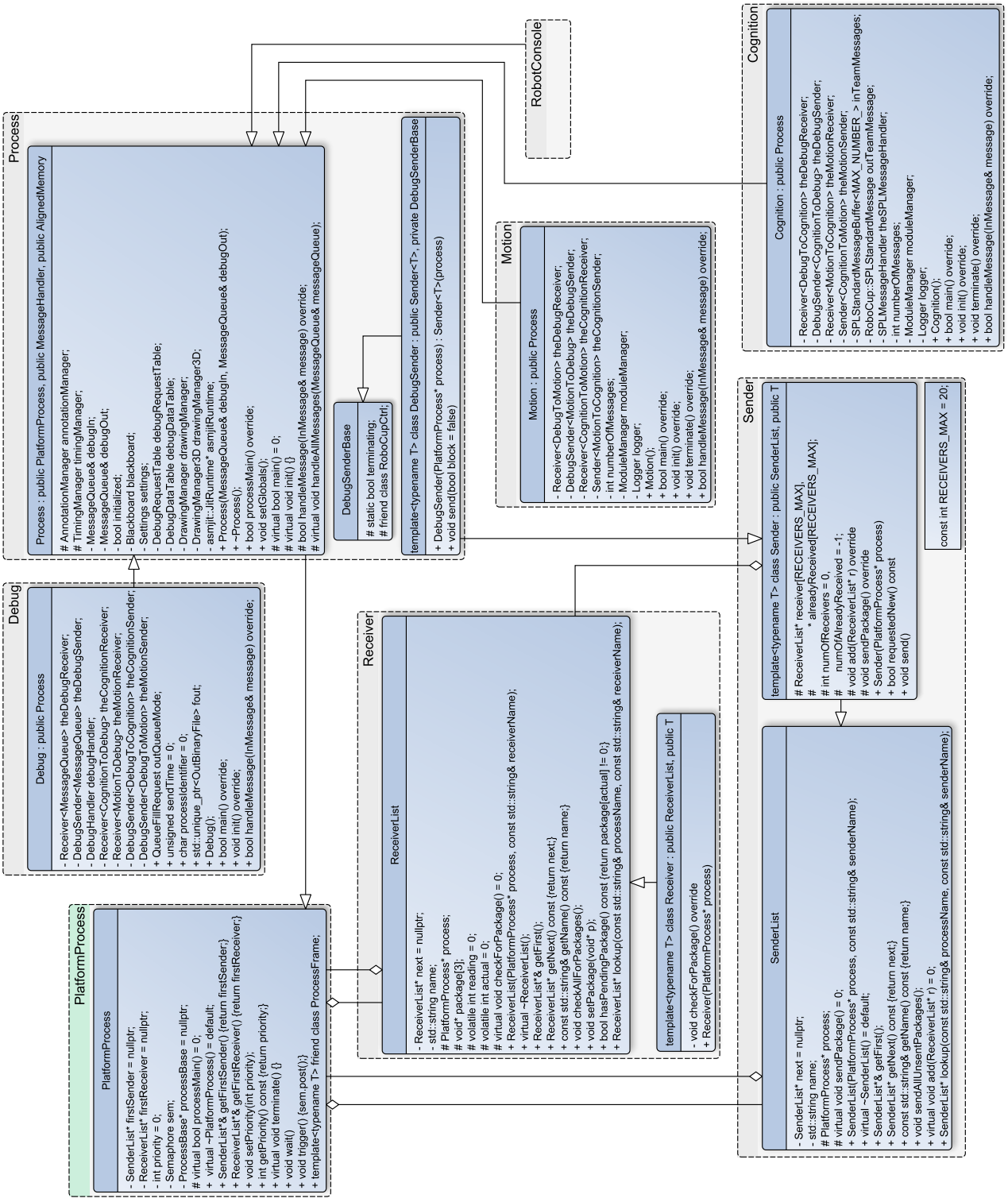


Abbildung 3.8 Das Klassendiagramm der Prozesse und Kommunikation. Die UML-Notation wurde dazu angepasst. Die äußeren Kästen beschreiben hierbei Dateien und die zweigeteilten Kästen stellen Klassen darin dar. Funktionen ohne Semikolon am Ende bedeuten, dass diese im Interface implementiert sind. Die Kompositionen zwischen den Prozessen mit der Kommunikation wurde hier aus Gründen der Übersicht ausgelassen. Die `RobotConsole` steht hierbei stellvertretend für den Simulator.

Kapitel 4

Verwandte Arbeiten

Im Folgenden werden verwandte Arbeiten und Systeme vorgestellt und mit dem in Abschnitt 3.5 beschriebenen bisherigen Framework von B-Human verglichen. Zuerst werden verwandte Arbeiten in der SPL (Abschnitt 4.1), anschließend ROS (Abschnitt 4.2) und dessen Verwendung in der SPL dargestellt.

4.1 Verwandte Arbeiten in der SPL

Adikari u. a. [2017]: Das Team HULKs hat ein eigenes Framework geschrieben. Der überwiegende Teil der Logik und Algorithmen ist in unabhängigen Einheiten implementiert, Module genannt. Diese benötigen Daten und stellen andere Daten bereit. Aus diesen Abhängigkeiten ergibt sich ein Graph, der die Ausführungsreihenfolge bestimmt, da alle benötigten Daten eines Moduls zu berechnen sind, bevor dieses ausgeführt wird. Die Daten sind in einer Datenbank pro Thread gespeichert. Da der Roboter Sensordaten in zwei unterschiedlichen Taktraten bereitstellt, gibt es zwei Threads. Einer bekommt die Kamerabilder und verwaltet alle Module, die sich mit diesen beschäftigen. Zusätzlich kümmert sich dieser Thread auch um die Modellierung der Welt und das Verhalten. Der andere Thread verarbeitet die restlichen Sensoren und steuert die Gelenke. Außerdem gibt es die Möglichkeit, Parameter zu konfigurieren, sowie die Möglichkeit, mit einem Computer zu debuggen. Diese Architektur ist sehr ähnlich zu der, die B-Human bisher verwendet, allerdings ohne C++-Makros. Ein Unterschied ist, dass es keinen expliziten Debug-Thread gibt, der sich um die Kommunikation mit der Außenwelt kümmert. Eine Singleton-Klasse übernimmt stattdessen diese Aufgabe. Weiterhin ist kein Austausch von Modulen zur Laufzeit möglich.

Wünsch [2013]: Das Team *NAO*-Team HTWK benutzt ein komplett eigenes Framework. Dieses besteht aus zwei Teilen. Der eine Teil ist eine Brücke zu NaoQI, der vom Hersteller bereitgestellten Software, die auf den *NAO*s läuft. Diese bildet eine Art API, mit der der *NAO* angesprochen und Daten empfangen werden können. Gleichzeitig werden dort auch alle Bewegungen behandelt. Der andere Teil ist der Hauptteil. Dieser ist mit mehreren Threads

und Agenten realisiert (Abbildung 4.1). Zwei Threads nehmen die Kamerabilder entgegen und übergeben diese an den *Imageprocessor*. Aus den Bildern rechnet dieser mit den weiteren Threads *Vision* und *Lokalisierung* aus, was auf den Bildern zu sehen ist und wo sich der Roboter aktuell befindet. Die gewonnenen Informationen werden an die Weltmodellierung weitergeleitet, welche diese dann verarbeitet und an die Mitspieler versendet. Aus allen Modellen berechnet jeder Roboter die Teamstrategie und gibt abstrakte Ziele, wie zum Beispiel „laufe zu Position x“, an die Agenten weiter. Diese Agenten hören auf bestimmte Ziele und berechnen, wie lange sie für die Aktion benötigen. Der beste Agent wird ausgewählt und führt die Aktion mithilfe der Brücke zu NaoQI aus. Insgesamt werden dann inklusive der Bibliotheken 19 Threads ausgeführt. Sie kommunizieren über die Bibliothek ZeroMQ und eine andere Implementierung dieser Bibliothek, die sich nullMQ nennt [Wikidot.com, 2018]. Diese abstrahiert die Kommunikation und wird sowohl intern als auch für die Kommunikation mit anderen Tools verwendet. Nachrichten sind mit einer weiteren Bibliothek namens Google Protocol Buffers (kurz Protobufs) realisiert [Google, 2018]. Diese bietet eine strukturierte Datenform mit einer threadsicheren Serialisierung und Deserialisierung für verschiedene Programmiersprachen.
Dieses System kann im Gegensatz zum bisherigen Ansatz von B-Human die Möglichkeiten des neuen *NAO V6* (Abschnitt 2.3) effizienter nutzen, da es bereits besser parallelisierbar ist. Ein möglicher, genereller Nachteil der Architektur ist allerdings, dass in den einzelnen Threads keine weitere unabhängige Modularisierung vorgenommen wird, wie in aktuelleren Veröffentlichungen zu sehen ist [Nao-Team HTWK Leipzig, 2018]. Dies erschwert somit den Austausch einzelner Komponenten, wenn diese mit mehreren Klassen interagieren. Weiterhin ist somit auch kein Austausch von Algorithmen zur Laufzeit möglich.

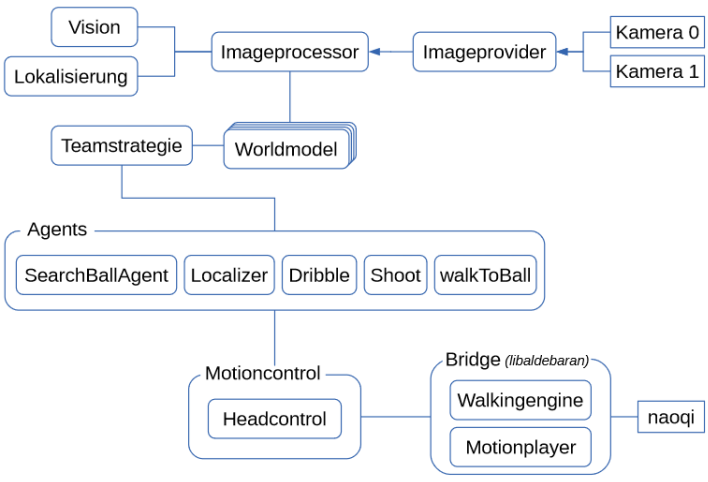


Abbildung 4.1 Schematischer Aufbau des Frameworks von HTWK. (Entnommen aus [Wünsch, 2013, S. 16])

Mamantov u. a. [2015]: Das Team Northern Bites benutzte bis 2013 kein Framework, was in einem Quelltextblock ohne getrennte Komponenten endete und somit nicht mehr zu überschauen war. Aus diesem Grund haben sie ein neues Framework entwickelt. Dieses behandelt

Module, wie sie weiter oben schon erwähnt wurden, als sogenannte Blackboxen, die eingehende und ausgehende Daten als Schnittstellen haben. Es gibt dabei synchrone und asynchrone Kommunikation. Synchroner Datenaustausch wird für Daten angewendet, die neu berechnet sind, wodurch sich auch die Ausführungsreihenfolge ergibt. Asynchroner Datenaustausch wird für Zyklen und Datenaustausch mit anderen Threads verwendet. Jede Nachricht hat dabei einen Referenzzähler, der dafür sorgt, dass die Nachricht gelöscht wird, sobald sie nicht mehr benötigt wird. Damit Nachrichten effizient zur Verfügung stehen, gibt es einen Pool, der neue Nachrichten vorhält. Analog zum *NAO*-Team HTWK wird die Protobuf-Bibliothek für den Nachrichtenaustausch verwendet.
Durch das direkte Versenden der Daten zwischen Modulen ist es im Gegensatz zum Blackboardansatz von B-Human nicht möglich, dass ein Modul auf alle Daten zugreifen könnte. Dies umgeht somit diese Fehlerquelle. Ein weiterer Vorteil können Messagepools sein, wenn gleiche Datenstrukturen für mehrere Zwecke verwendet werden. Dies ist bei B-Human allerdings nicht der Fall. Dort existieren alle Daten innerhalb eines Threads genau ein Mal im Blackboard. Zwischen Threads wird für jede Nachricht aber neuer Speicher reserviert, in den die Daten kopiert werden.

4.2 ROS

Das wohl bekannteste und populärste Framework für Robotik ist das Robot Operating System (ROS) [Open Robotics, 2018], welches auch in vielen anderen Ligen des RoboCups eingesetzt wird. Neben diesem gibt es viele weitere Frameworks, wie YARP [Metta u. a., 2006], Orocos [Bruyninckx, 2001] oder aRD [Bäuml u. Hirzinger, 2006]. In Bäuml u. Hirzinger [2008] sind diese und weitere beschrieben. Im Folgenden werden kurz ROS und Versuche damit in der SPL vorgestellt.

ROS ist ein Open Source Roboterframework, welches eine Art Meta-Betriebssystem zwischen dem Betriebssystem und der Robotersoftware darstellt. Lose gekoppelte Prozesse (Nodes) kommunizieren dabei über die ROS Kommunikationsinfrastruktur und bilden so ein gekapseltes Gesamtsystem, für das es viele weitere Werkzeuge gibt. Dies erlaubt eine hohe Wiederverwendbarkeit und Austauschbarkeit. Das System ist dabei durch veröffentlichte Pakete erweiterbar.
Es gibt drei Kommunikationsarten. Diese erlauben synchrone oder asynchrone Kommunikation sowie Kommunikation mit einem Server, welcher Parameter aus Konfigurationen bereitstellt. ROS ist dabei nicht echtzeitfähig, kann aber mit echtzeitfähigen Komponenten zusammenarbeiten [Quigley u. a., 2009].

Für viele Roboterplattformen gibt es Stacks, also Sammlungen von Paketen, welche die Anbindung der Plattform an ROS bereitstellen. So auch für den *NAO* bis zur Version *NAO V4* [Hornung u. Humanoid Robots Lab Freiburg, 2010]. Das Team UChile hat mit diesem gearbeitet, um zu testen, ob sich ROS für die SPL eignen könnte. Dies würde Teams die Mög-

lichkeit geben, alle veröffentlichten Pakete nutzen zu können. Auch der Austausch der Teams untereinander würde vereinfacht werden, wenn alle das gleiche Framework benutzten. Hierfür wurde von UChile die Laufbewegung von B-Human und weitere Bewegungen als eine Node implementiert und die Kommunikation zwischen Robotern mit ROS ausprobiert. Dabei hat ein Roboter eine Ballposition an einen blinden Roboter geschickt und dieser ist mit dem implementierten Laufen zum Ball gelaufen [Forero u. a., 2014]. Die Kommunikation über ROS-Nachrichten ist in der SPL allerdings nicht mehr erlaubt, da es ein vorgegebenes Format gibt [RoboCup Technical Committee, 2018, S. 8]. Es wurde kein Paper dazu gefunden, aber aktuell gilt in der SPL die Annahme, dass ROS auf dem *NAO V5* zu langsam wäre, um es als Framework nutzen zu können – unter anderem, da angenommen wird, dass der Overhead zu groß sei. Viele Pakete sind auch nicht auf eine so geringe Rechenleistung ausgelegt und würden somit keinen Mehrwert bieten. In Röfer u. Laue [2014] gibt es einen kurzen Vergleich zwischen ROS und B-Human.

Kapitel 5

Realisierung

In diesem Kapitel werden zunächst die Anforderungen (Abschnitt 5.1) an das neue Framework beschrieben. Darauf aufbauend wird das Konzept (Abschnitt 5.2) erläutert und anschließend wird die Implementierung (Abschnitt 5.3) vorgestellt. Ab diesem Kapitel werden Threads auch Threads genannt und es wird nicht mehr von Prozessen als parallele Einheit gesprochen.

5.1 Anforderungen

Im Folgenden werden die Anforderungen, welche das neue Framework erfüllen muss, beschrieben. Diese ergeben sich aus Kritikpunkten des aktuellen Frameworks, Anforderungen für die Zukunft und Wünschen von B-Human. Die Entwicklung des neuen Frameworks darf dabei den Entwicklungsprozess bei B-Human nicht einschränken. Somit sollten nur so wenig Änderungen wie möglich an anderen Komponenten des Systems vorgenommen werden.

Ein Kritikpunkt am aktuellen Framework ist, dass es nicht einfach zu verstehen ist. Dies hängt unter anderem mit der Entkopplung der Logik in den Threads von der Implementierung der Systemthreads zusammen. Aktuell gibt es keinen Anlass, dies weiter zu unterstützen, da mittlerweile alle verwendeten Systeme mit Multithreading umgehen können. Falls es mehrere Prozesse geben muss, ist der Grund dafür somit wahrscheinlich, dass es mehrere CPUs gibt. Das Framework hat aber nicht den Anspruch, auf mehreren CPUs zu laufen. Dafür würden sich dann andere Robotikframeworks eignen (Abschnitt 4.2). Ein weiterer Punkt ist, dass das Framework ursprünglich für den AIBO implementiert wurde und einige Komponenten spezifisch dafür umgesetzt wurden. Durch die Neuentwicklung sollen diese allgemein formuliert werden. Außerdem unterbrechen Threads aktuell die Ausführung, wenn eine neue Konfiguration eingespielt wird, da alle Threads dann die neue Konfiguration ausrechnen. Dies soll in Zukunft nach Möglichkeit parallel zum normalen Betrieb geschehen.

Um die mit den *NAO V6* zur Verfügung stehenden Rechenkerne ausnutzen zu können, soll es die Möglichkeit geben, eine variable Anzahl von Threads zu haben. Die Kommunikation soll dabei weiterhin vollautomatisch stattfinden, wobei unnötiger Datenaustausch zu vermeiden

ist. Somit sollten nur Threads miteinander kommunizieren, die auch Daten austauschen und es sollten nur Daten ausgetauscht werden, die der Empfänger benötigt.
Da der *NAO* zwei Kameras hat, möchte B-Human beide Bilder parallel verarbeiten können. Dies führt dazu, dass zum Beispiel beide Threads eine Repräsentation für die Ballposition haben. Wenn das Tracken des Balls nun von einem weiteren Thread übernommen wird, benötigt dieser jedoch beide Repräsentationen, um seine Aufgabe zu erfüllen. Bisher muss eine Repräsentation allerdings eindeutig sein, sodass in diesem Fall ein Fehler aufträte. Um mit dieser Situation umgehen zu können, sollen sogenannte Aliasse zwischen Threads ausgetauscht werden können. Somit kann jeder Kamera-Thread weiterhin mit der Repräsentation `Ball` arbeiten, ein Thread der einen oder beide Bälle benötigt, kann zum Beispiel `UpperBall` erwarten und bekommt dann die Repräsentation des Balls aus dem Thread `Upper`.

5.2 Konzept

Für das Konzept wurde zunächst eine Annahme getroffen, die die Funktionalität genauer definiert. Diese sieht vor, dass es keine neuen Threads zur Laufzeit geben kann. Dies ist damit begründet, dass die Threads in der Konfigurationsdatei angegeben werden und zum Experimentieren nicht neu gebaut, sondern nur die Datei verändert werden muss.

Am Anfang des Konzepts steht die Konfigurationsdatei, in der die Threads und die dazugehörigen Module eingetragen werden (Quelltext 5.1). Diese besteht aus einer Liste von Threads. Einem Thread werden ein Name und eine Priorität zugeordnet. Das Attribut `requiredThreads` ist eine Liste von Threadnamen. Sie gibt an, von welchen Threads der Thread Daten erwartet und dementsprechend eine Kommunikationsverbindung aufgebaut werden soll. Das Attribut `representationProviders` ist eine weitere Liste. In dieser sind alle Repräsentationen eingetragen, die aktuell bereitgestellt werden und welches Modul dies tut.

```
threads = [
  {
    name = ThreadName;
    priority = 0;
    requiredThreads = [
      ThreadName2,
    ];
    representationProviders = [
      {representation = Representation; provider = Provider;},
      {representation = Representation2; provider = Provider2;},
    ];
  },{
    name = ThreadName2;
    priority = 0;
    requiredThreads = [
      ThreadName,
    ];
```

```
    representationProviders = [
      {representation = Representation3; provider = Provider2;},
      {representation = Representation4; provider = Provider3;},
    ];
  },
];
```

Quelltext 5.1 Konzeptioneller Aufbau der Konfigurationsdatei für Threads. Es sind zwei Threads mit jeweils zwei bereitgestellten Repräsentationen dargestellt, wobei beide Threads jeweils Daten des anderen erwarten. Die Schreibweise entspricht der von Konfigurationsdateien bei B-Human.

Diese soll zum Programmstart eingelesen und daraus die gewünschten Threads konstruiert werden. Dabei wird weiterhin zwischen Simulator und realem Roboter unterschieden. Außerdem soll es weiterhin eine einfache Möglichkeit geben, zusätzliche Threads, die keine Module ausführen, zu erstellen. Einer dieser Threads ist `Debug`. Außerdem könnte zu einem späteren Zeitpunkt das Loggen in einen eigenen Thread ausgelagert werden, der dies zentral übernimmt. Das Ganze soll über eine Vererbungshierarchie realisiert werden, an deren unteren Ende eine allgemeine Implementierung von modulverarbeitenden Threads, `Debug` und weitere Threads sein können. Die Kommunikation über `ModulePackage` und `MessageQueue` soll erhalten bleiben, wobei die `Sender` und `Receiver` als einzelne Klassen mit dem gleichen Übertragungsweg wie bisher neu implementiert werden. Um eine Kommunikation zwischen mehr als zwei Threads zu ermöglichen, soll der `ModuleManager` an eine zentrale Stelle wandern. Dort soll der Datenaustausch und die neue Modulreihenfolge parallel zum laufenden Betrieb ausgerechnet werden. Anschließend wird diese in den laufenden Betrieb übernommen.

Dies ist in mehreren Schritten geplant. Zuerst soll die Klassenhierarchie und die Kommunikation neu geschrieben werden. In diese wird die neue Konfigurationsdatei eingegliedert. Mit dieser sollen dann die modulverarbeitenden Threads in eine Klasse zusammengeführt werden, von der ein Thread für jeden Thread in der Konfiguration erstellt wird. Anschließend wird das Berechnen der auszutauschenden Daten mit den neuen Anforderungen implementiert.

5.3 Implementierung

Im Folgenden wird die im Rahmen dieser Arbeit entstandene Implementierung beschrieben. Dabei wird, wie in der Beschreibung des bisherigen Frameworks (Unterabschnitt 3.5.2), auf den Programmstart (Unterabschnitt 5.3.1), die Kommunikation (Unterabschnitt 5.3.2) und den Aufbau der Thread-Infrastruktur (Unterabschnitt 5.3.3) eingegangen. Anschließend wird der `ModuleGraphCreator` (Unterabschnitt 5.3.4) näher beschrieben.

Die Konfigurationsdatei (Quelltext 5.2) ist im Gegensatz zu der im Konzept gezeigten Version erweitert worden. Neu dazu gekommen ist eine Liste mit Repräsentationen, die das Modul „default“ bereitstellt. Außerdem speichert ein Thread zusätzlich noch die Größe von Debug-Sendern und -Empfängern. Ein solcher Sender erwartet zusätzlich noch eine Größe in Byte, die

angibt, wie viel Speicher für die Infrastruktur reserviert ist. Diese wird ebenfalls gespeichert. Die letzte Änderung ist eine sogenannte *executionUnit* pro Thread (Abbildung 5.3). Dies sind Klassennamen von speziellen Klassen, die Anweisungen für den Ablauf des Threads bereitstellen (vgl. Unterabschnitt 5.3.3).

```
defaultRepresentations = [
  RepresentationXY,
];
threads = [
  {
    name     = Upper;
    priority = 0;
    debugReceiverSize = 2800000;
    debugSenderSize   = 5200000;
    debugSenderInfrastructureSize = 100000;
    executionUnit = Cognition;
    requiredThreads = [
      Motion,
    ];
    representationProviders = [
      {representation = Representation; provider = Provider;},
      {representation = Representation2; provider = Provider2;},
    ];
  },{
    name     = Motion;
    priority = 20;
    debugReceiverSize = 500000;
    debugSenderSize   = 50000;
    debugSenderInfrastructureSize = 20000;
    executionUnit = Motion;
    requiredThreads = [
      Upper,
    ];
    representationProviders = [
      {representation = Representation3; provider = Provider2;},
      {representation = Representation4; provider = Provider3;},
    ];
  },
];
```

Quelltext 5.2 Aufbau der Konfigurationsdatei für Threads. Es sind zwei Threads mit jeweils zwei bereitgestellten Repräsentationen dargestellt, wobei beide Threads jeweils Daten des anderen erwarten. Die Schreibweise entspricht der von Konfigurationsdateien bei B-Human.

5.3.1 Programmstart

Analog zum bisherigen Verfahren der Registrierung von Prozessen (vgl. Abschnitt 3.5.2) wird beim Programmstart (Abbildung 5.1) automatisch eine Liste aller `FrameExecutionUnits` erstellt, indem jeder `ExecutionUnitCreator` sich selbstständig in dieser registriert. Um das Framework zu starten, wird ein `Robot` erstellt. Dieser unterscheidet sich, je nachdem, ob er auf einem *NAO* oder im Simulator gestartet wird. Der simulierte Roboter besitzt zusätzlich noch Funktionalitäten für den Simulator, die sich nicht verändert haben und hier nicht weiter betrachtet werden. Ein Roboter ist dabei eigentlich eine Liste von `ThreadFrame`-Zeigern. Er liest die Konfigurationsdatei ein und erstellt `Debug` und alle Threads der Konfiguration als `ModuleContainer`. Dieser lädt dabei jeweils die threadspezifischen Anweisungen. `ThreadList` bietet die Möglichkeit, alle Prozesse zu starten und zu stoppen.

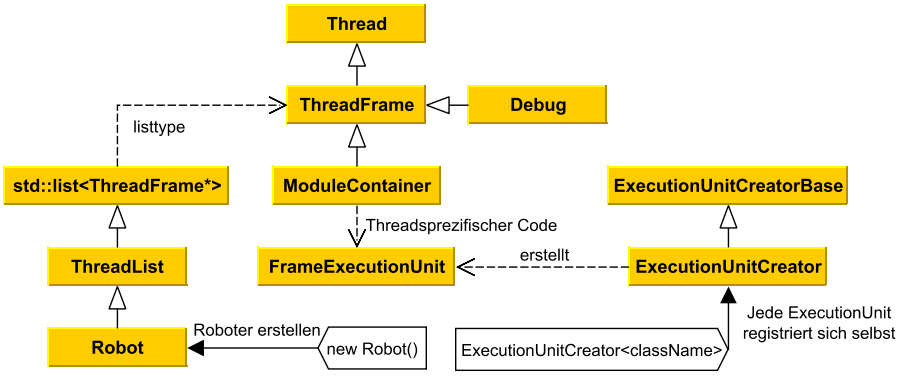


Abbildung 5.1 Das Klassendiagramm beschreibt den Aufbau des zum Starten relevanten Teil des Frameworks. Die UML-Notation wurde um Funktionen und Member der Klassen reduziert, da diese in Abbildung 5.4 beschrieben werden. Zusätzlich ist der Einstiegspunkt in die Software und das Registrieren der ExecutionUnits dargestellt.

5.3.2 Kommunikation

Die Kommunikation (Abbildung 5.2) ist in `Sender` und `Receiver` implementiert. Von diesen erben Klassen für Debug-Varianten. Dabei reserviert ein `Sender` Speicher, in den die zu sendenden Daten geschrieben werden. Anschließend wird ein Zeiger auf den Speicher an den `Receiver` übergeben und der Speicher kann wieder ausgelesen werden. Somit läuft die Kommunikation asynchron mittels gemeinsamen Speichers, ohne dass eine Synchronisation notwendig wäre. Ein `Sender` hat dabei genau einen `Receiver`, dem er Pakete schicken kann. Der `DebugSender` hat zusätzlich einen Parameter für den Simulator. Weiterhin wird ein Paket nur dann gesendet, wenn es nicht leer ist. Das Senden kann blockierend sein, bis ein neues Paket angefordert wird. Der `Receiver` kann auf ein neues Paket prüfen, welches er auch direkt einliest, falls es existiert. Er kann bis zu drei Pakete zwischenspeichern. Dies sorgt für eine deadlockfreie Kommunikation, da gleichzeitig maximal ein Paket gelesen, ein

anderes geschrieben und ein drittes auf das Lesen warten kann. Ansonsten wird ein Paket überschrieben, das gerade weder wartet noch gelesen wird. Gelesen wird immer das neuste fertig geschriebene Paket. Dadurch ist es möglich, dass Pakete verloren gehen. Es sind aber auch immer die aktuellsten Pakete am wichtigsten, da sie die neuesten Informationen enthalten und diese verarbeitet werden sollen bei B-Human.

Pakete sind dabei weiterhin generische Typen, welche entweder `ModulePackage`s oder `MessageQueue`s sind (Abbildung 3.6). Die Debug-Varianten haben bei B-Human immer `MessageQueue` als Typ und können die Größen der Queue setzen.

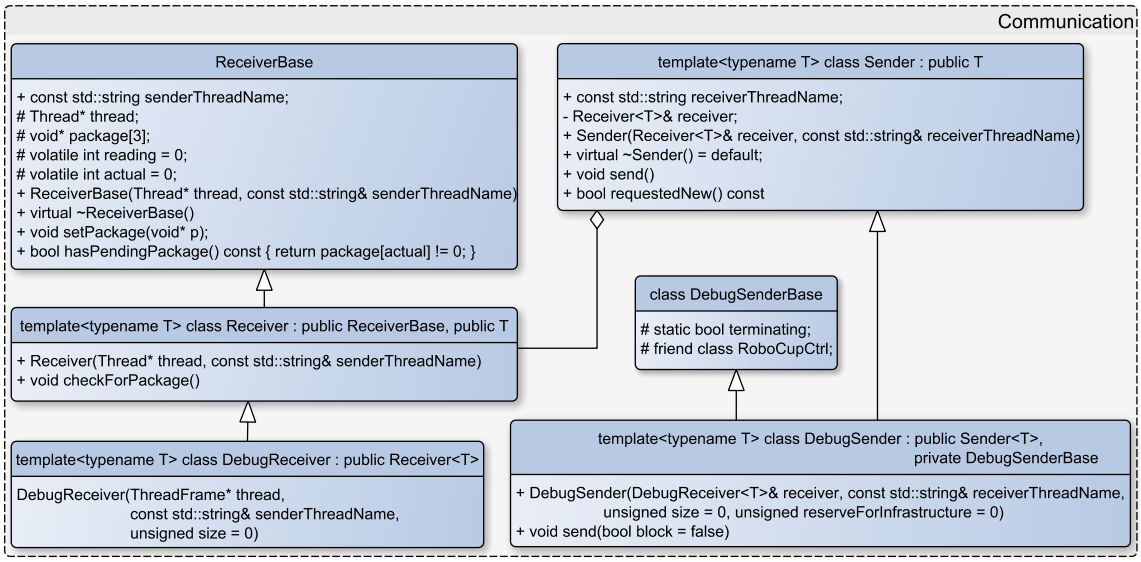


Abbildung 5.2 Das Klassendiagramm der Kommunikation. Die UML-Notation wurde dazu angepasst. Die äußeren Kästen beschreiben hierbei Dateien und die zweigeteilten Kästen stellen Klassen darin dar. Funktionen ohne Semikolon am Ende bedeuten, dass diese im Interface implementiert sind.

5.3.3 Threads

Jeder für diese Arbeit relevante Thread erbt von `ThreadFrame`. Dieser besitzt alle threadweiten Objekte. Dazu gehören unter anderem die Debugverbindungen, das Blackboard und verschiedene Debuggingfeatures. Die Debugverbindung ist dabei für einen `ModuleContainer` die Verbindung zu `Debug` und für `Debug` eine Verbindung zum Simulator. `ThreadFrame` hat außerdem die Funktion `threadMain()`, welche die Hauptschleife eines Threads ist. `ThreadFrame` erbt von der Klasse `Thread`. Diese wurde aus dem bisherigen Framework übernommen und implementiert gekapselte plattformabhängige `std::thread`s. Sie wurde dabei um die Möglichkeit erweitert, den Namen des Threads zu erfragen. Diese Funktion wurde zum aktuellen Zeitpunkt noch nicht verwendet, was in Zukunft aber noch geschehen soll. Der `ModuleContainer` erbt von `ThreadFrame`. Er repräsentiert alle Threads, die in der Konfiguration festgelegt wurden. Dabei hat er Listen von Threads, von denen er etwas sendet oder empfängt. Weiterhin

hat er den Logger. Somit loggt jeder Thread für sich selbst. Die `main()`-Funktion implementiert jeweils den Ablauf eines Zyklus. Threads haben spezielle Anweisungen, bei denen es nicht gelungen ist, sie zu abstrahieren oder in Module auszulagern. Der Grund dafür ist, dass zum Beispiel das Warten auf den nächsten Zyklus nicht einheitlich implementierbar ist oder die Bedingung, ob ein Zyklus ausgeführt werden soll, durch unterschiedliche Funktionsaufrufe geprüft wird. Deshalb hat jeder `ModuleContainer` eine `FrameExecutionUnit` (Abbildung 5.3). Diese stellen spezielle Funktionen bereit, die diese Probleme lösen.

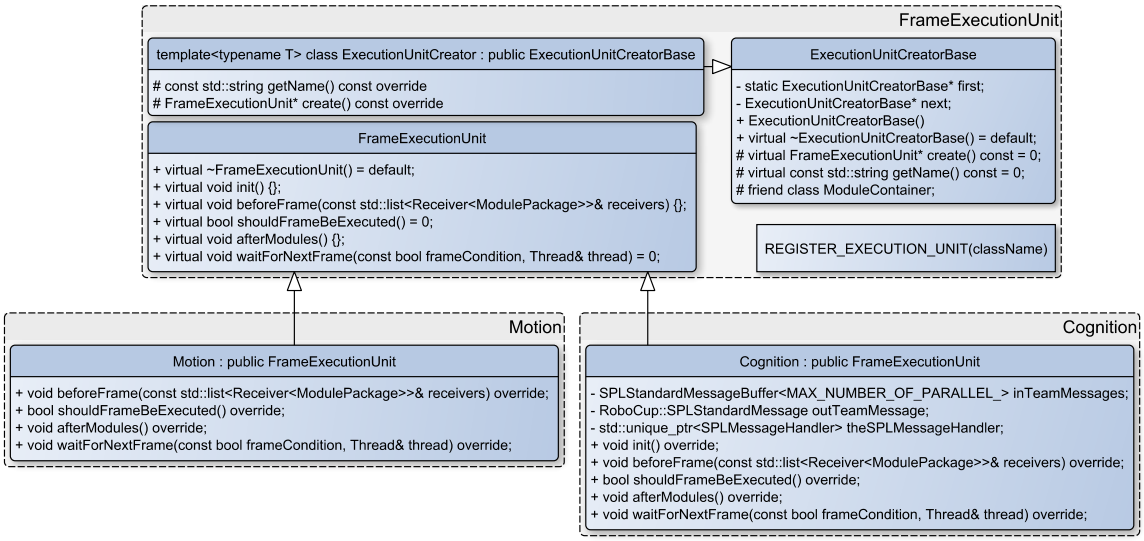


Abbildung 5.3 Das Klassendiagramm der *ExecutionUnits*. Die UML-Notation wurde dazu angepasst. Die äußeren Kästen beschreiben hierbei Dateien und die zweigeteilten Kästen stellen Klassen darin dar. Funktionen ohne Semikolon am Ende bedeuten, dass diese im Interface implementiert sind.

In einem Zyklus werden auch die Module ausgeführt. Dies übernimmt der `ModuleGraphRunner`. Er bekommt alle für seinen Thread relevanten Informationen vom `ModuleGraphCreator` und stellt daraus den nötigen Zustand her, um die Module ausführen zu können (Abbildung 5.5).

Auch `Debug` erbt von `ThreadFrame`. Die Klasse wurde übernommen und auf den Umgang mit einer variablen Anzahl an Threads erweitert. Außerdem berechnet `Debug` jetzt mit dem `ModuleGraphCreator` an einer zentralen Stelle die Modulkonfigurationen für alle Threads und wie diese Daten austauschen müssen. Dabei werden die Threads nur aktualisiert, falls die Konfiguration valide ist. Diese wird dann an die Threads verschickt. Das hat zur Folge, dass `Debug` immer initialisiert wird, allerdings beendet es sich bei Wettkämpfen sofort wieder, sobald die Modulkonfigurationen verschickt wurden. Dies geschieht einerseits um Rechenzeit zu sparen und andererseits, da ohnehin kein Simulator angeschlossen ist und somit keine Daten ausgetauscht werden.

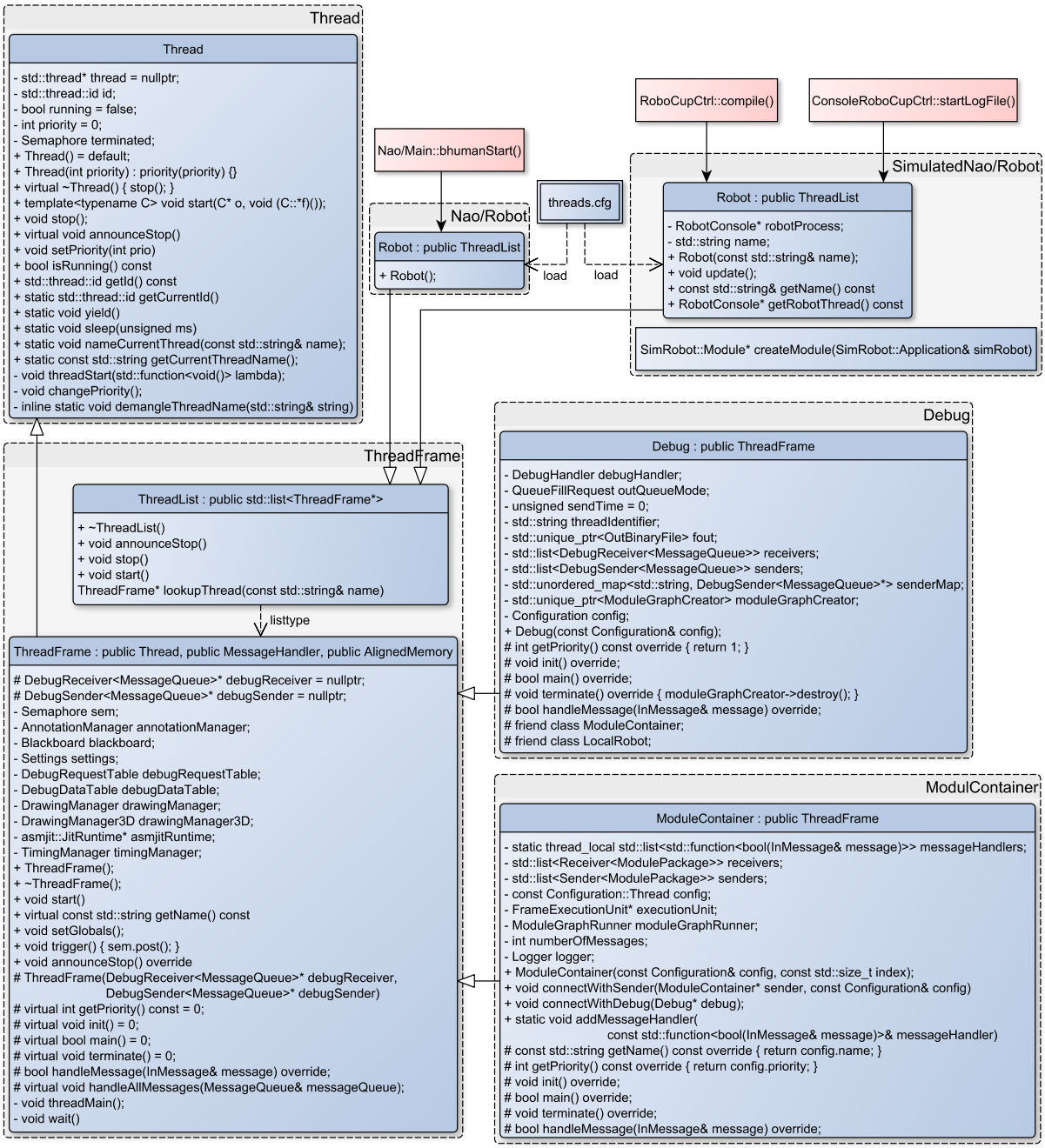


Abbildung 5.4 Das Klassendiagramm der Threads. Die UML-Notation wurde dazu angepasst. Die äußeren Kästen beschreiben hierbei Dateien und die zweigeteilten Kästen stellen Klassen darin dar. Einfache Kästen beschreiben Besonderheiten. Die Kästen außerhalb der Dateien sind Einstiegspunkte in die Software. Doppelt umrandete Kästen sind Dateien, die eingelesen werden. Funktionen ohne Semikolon am Ende bedeuten, dass diese im Interface implementiert sind.

5.3.4 ModuleGraphCreator

Der `ModuleManager` (Abbildung 3.7) ist in den `ModuleGraphCreator` und den `ModuleGraphRunner` (Abbildung 5.5) aufgeteilt worden. Der `ModuleGraphRunner` kümmert sich dabei

nur um die Ausführung der Module. Der `ModuleGraphCreator` berechnet für die gesamte Thread-Konfiguration, ob diese valide ist und in welcher Reihenfolge jeder Thread welche Module ausführen muss. Außerdem berechnet er den nötigen Datenaustausch zwischen den Threads. Aus dem `ModuleManager` wurde `execute()` in den `ModuleGraphRunner` übernommen und an die neue Struktur angepasst. In den `ModuleGraphCreator` wurden die Funktionen `rollBack(...)`, `find(...)` und `sortProviders(...)` übernommen und an die neue Struktur angepasst. Der Rest ist neu implementiert. Insbesondere wurde die Berechnung der Abhängigkeiten zwischen Threads im Hinblick auf zu schickende Nachrichten neu geschrieben (Algorithmus 5.1). So wird jetzt nur noch berechnet, welche Repräsentationen ein Thread benötigt und nicht, welche er verschicken muss. Die erwarteten Repräsentationen werden anschließend für den sendenden Thread gesetzt. Dabei werden auch die in den Anforderungen (Abschnitt 5.1) angesprochenen Aliasse aufgelöst.

Ein Alias ist eine Repräsentation, welche einen Threadnamen als Präfix vor dem eigentlichen Namen hat. Die Repräsentation muss dabei von der eigentlichen Repräsentation erben, ohne sie um Member zu erweitern. Wenn eine Repräsentationen von mehreren Threads bereitgestellt wird und ein Modul diese erwartet, muss ein Alias benutzt werden, um anzugeben, welche Repräsentation es benutzt. Dabei gilt die Annahme, dass eine Repräsentation, die ein Alias sein kann, nicht zugleich selbst bereitgestellt wird, wenn einer ihrer Aliasse bereitgestellt wird. Wenn `ModulC` in Quelltext 5.3 `AlphaA` erwartet, könnte ein Programmierer davon ausgehen, dass er `A` aus `Alpha` bekommt oder möglicherweise `AlphaA` aus `Beta` bekommt. Dies wird somit verhindert. Gleiches gilt auch für den Fall, dass `AlphaA` statt in `Beta` in einem anderen Thread bereitgestellt wird. Dies soll Mehrdeutigkeit und falsche Annahmen von Programmierern verhindern.

```
threads = [
  {
    name = Alpha;
    representationProviders = [
      {representation = A; provider = ModulA;},
    ];
  },{
    name = Beta;
    requiredThreads = [
      Alpha,
    ];
    representationProviders = [
      {representation = AlphaA; provider = ModulB;},
      {representation = B; provider = ModulC;},
    ];
  },
];
```

Quelltext 5.3 Minimalbeispiel für einen möglichen Konflikt von Aliasen. Die Schreibweise entspricht der von Konfigurationsdateien bei B-Human.

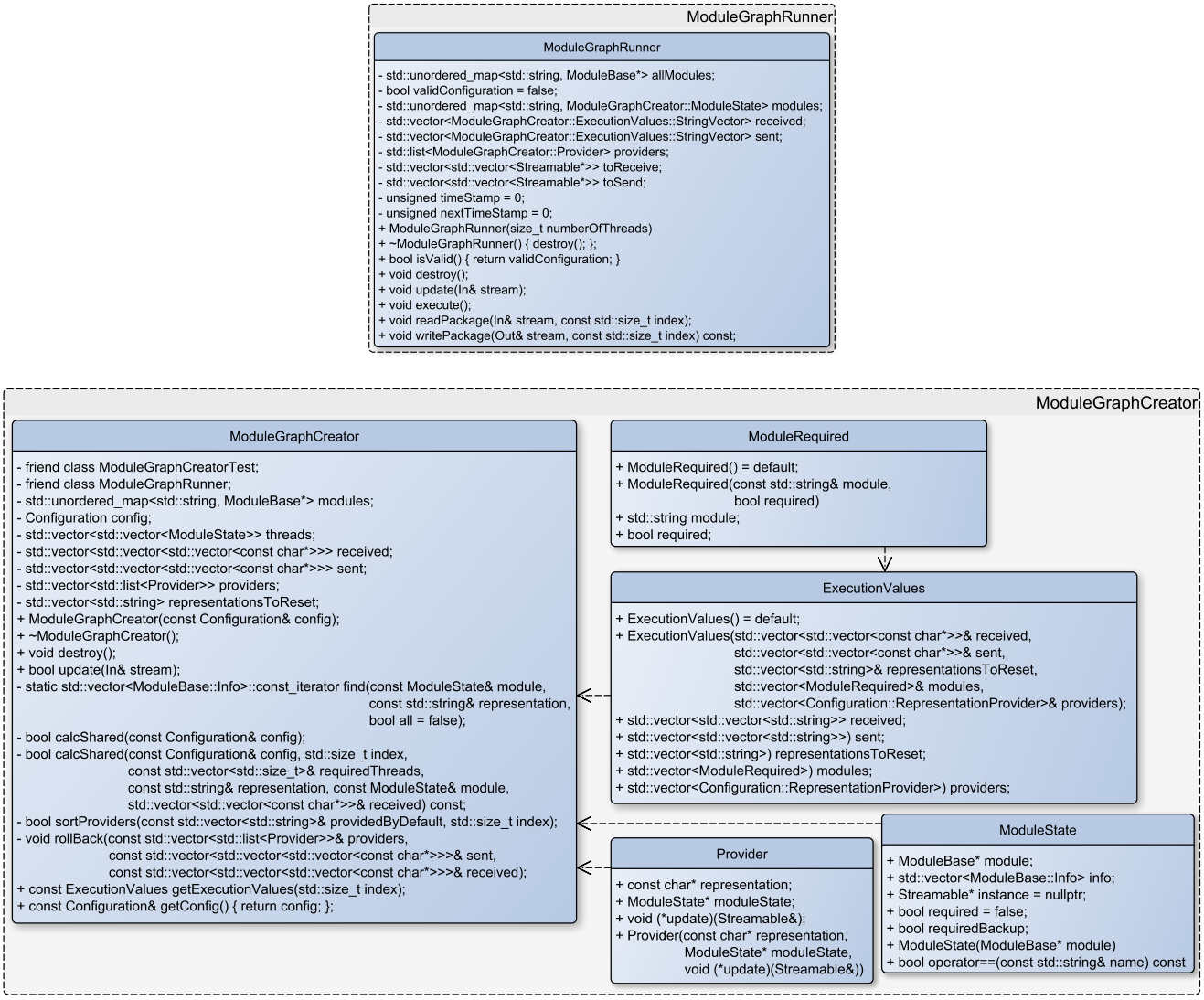


Abbildung 5.5 Das Klassendiagramm des `ModuleGraphCreator` und des `ModuleGraphRunner`. Die UML-Notation wurde dazu angepasst. Die äußeren Kästen beschreiben hierbei Dateien und die zweigeteilten Kästen stellen Klassen darin dar. Da das verwendete Tool die UML-Notation für innere Klassen nicht kennt, wird hier die Abhängigkeit dafür verwendet, um dies darzustellen.

Im Folgenden wird die Berechnung der Abhängigkeiten zwischen Threads in Hinblick auf zu schickende Nachrichten genauer erläutert. Die beiden Funktionen `calcShared(...)` (Algorithmus 5.1, 5.2) implementieren diese Funktionalität. Um eine eindeutige Zuordnung zu schaffen, wird im Folgenden die Funktion mit mehr Parametern mit `calcShared2` bezeichnet. `calcShared` bekommt eine Konfiguration als Eingabe und prüft, ob diese valide ist. Falls die Software an dieser einen Fehler feststellt, beendet die Funktion sich und wirft eine Fehlermeldung. Dabei wird die Liste der Threads in der Konfiguration durchgegangen und ruft für jedes `representationProvider`-Paar `calcShared2(...)` auf. Anschließend werden alle zu empfangenden Repräsentationen entsprechend zum Senden gesetzt. Hier wird geprüft, ob der sendende Thread die Repräsentation bereitstellt, oder ob es sich um einen Alias handelt.

Falls es sich um einen Alias handelt, wird dieser durch den Namen der äquivalenten Repräsentation ersetzt, die in diesem Thread bereitgestellt wird. Dies ist möglich, da Aliasse den gleichen Inhalt haben und somit Repräsentationen in das Paket geschrieben und als deren Alias-Repräsentationen wieder ausgelesen werden können.

```
1  Function calcShared(const Configuration& config) : bool
       Input: Die Konfiguration (Quelltext 5.2).
       Output: Ob die Konfiguration gültig ist.
2      if threads == ∅ and defaultRepresentations ≠ ∅ then ERROR(); return false;
3      else if threads == ∅ then return true;
4      for index = 0 to threads.size() do                    /* Calculate all threads. */
5          list<index> requiredThreads;
6          foreach required : threads[index].requiredThreads do    /* Find required threads. */
7              if threads.find(required) then
8                  requiredThreads.add(reqired.index)
9              else ERROR(); return false;
10         foreach rp : threads[index].representationProviders do
11             if not modulesPerThread.find(rp.prvider) then
12                 modulesPerThread.add(rp.provider)
13             else ERROR(); return false;
14             if not find(rp.provider, rp.representation) then ERROR(); return false;
               // Calculate shared.
15             if not calcShared2(config, index, ..., received[index]) then return false;
16         foreach thread : requiredThreads do        /* Set the senders to the receivers. */
17             sent[thread][index] = received[index][thread];
18             for i = 0 to sent[thread][index].size() do          /* Modify sent if alias. */
19                 if isProvided(config, sent[thread][index][i]) then continue;
                   // Representation is alias, find right representation.
20                 foreach rp : threads[thread].representationProviders do
21                     if alias == rp.representation then
22                         foreach requirement : rp.provider.info do
23                             if alias == requirement.representation then
24                                 sent[thread][index][i] = requirement.representation;
25                                 leaveAllLoops;
26                         .
       // Default can provide everything that exists, but only that.
27     foreach representation : defaultRepresentations do
28         if not providedByAModule(modulesPerThread, representation) then
29             ERROR(); return false;
30     return true;
```

Algorithmus 5.1: Dieser Algorithmus berechnet, welche Daten zwischen allen Threads ausgetauscht werden müssen. `modulesPerThread` steht hier für `threads` (Abbildung 5.5).

`calcShared2` prüft von dem Modul, welches die angegebene Repräsentation bereitstellt, ob jede von dem Modul erwartete Repräsentation auch bereitgestellt wird. Falls eine von diesen aus einem anderen Thread kommt, wird sie als zu empfangende Repräsentation aus dem

sendenden Thread hinzugefügt. Dabei werden ebenfalls Fehlermeldungen geworfen und die Berechnung abgebrochen, sofern ein Fehler festgestellt wurde. Für jede erwartete Repräsentation wird zudem überprüft, ob sie von „default“, dem eigenen Thread oder einem anderen Thread bereitgestellt wird. Falls die Repräsentation als Präfix einen Threadnamen hat, wird zusätzlich überprüft, ob dieser Thread das Modul bereitstellt.

```
1  Function calcShared2(config, index, requiredThreads, representation, module, received) : bool
       Input: Die Konfiguration (Quelltext 5.2), das Modul, dass geprüft werden soll, die Liste der zu
              empfangenden Repräsentationen und weitere Parameter zur Validierung.
       Output: Ob alle erwarteten Repräsentationen verfügbar sind.
2      if profidedByAnotherModule(config, representation, module) then ERROR(); return false;
3      foreach requirement : module.info do      /* Check if all requirements are provided */
4          if not requirement.update then
5              provided = maybeAlias = false;
6              aliasThreadIndex = -1;
7              if providedByDefault(config, requirement.repr) then continue;
8              if threadIndex = startWithThreadName(requiredThreads, requirement.repr) then
9                  maybeAlias = true;
10                 aliasThreadIndex = threadIndex;
11             if providedInOwnThread(config, requirement.repr) then provided = true; break;
12             if not provided then
13                 if providedByOneOtherThread(requirement.repr) then
14                     provided = true;
15                     received[thread].add(requirement.repr);
16                 else ERROR(); return false;
17             if maybeAlias then
18                 foreach rp : threads[aliasThreadIndex].representationProviders do
19                     if threads[aliasThreadIndex].name + rp.repr == requirement.repr then
20                         if provided then ERROR(); return false;
21                         if not areTypesAlias(requirement.repr, rp.repr) then
22                             ERROR(); return false
23                         provided = true;
24                         received[thread].add(requirement.repr);
25                         break;
26             if not provided then ERROR(); return false;
27     return true;
```

Algorithmus 5.2: Dieser Algorithmus überprüft alle erwarteten Repräsentationen eines Moduls und fügt diese, sofern nötig, als Erwartung vom richtigen Thread an. Er wird für jeden Eintrag in der Konfiguration aufgerufen.

Kapitel 6

Evaluation

Im Folgenden wird die Implementierung des Konzepts (Abschnitt 6.1) und anschließend die von B-Human implementierte Software (Abschnitt 6.2) hinsichtlich der Anforderungen und tatsächlichen Verwendung bei B-Human evaluiert.

Die Implementierung fand parallel zum normalen Entwicklungsprozess bei B-Human statt. Dieser wurde dabei nicht merklich negativ beeinflusst. So konnten Änderungen an Modulen und der Infrastruktur vorgenommen werden, ohne dass dies die Arbeit negativ beeinflusst hat und umgekehrt. Auch die Integration in den aktuellen Entwicklungsstand verlief ohne Probleme. Die daraus resultierenden Änderungen für das Projekt beschränken sich auf die neue Konfigurationsdatei, welche Parallelen zur vorherigen Datei aufweist, und die Änderung der Syntax eines Kommandos im Simulator. Die Anforderung, das Projekt B-Human nicht einzuschränken, wurde somit erfolgreich umgesetzt. Dies umfasst allerdings keine Änderungen, die nötig sind, um die Möglichkeiten des Frameworks auch komplett nutzen zu können. Derartige Änderungen muss das Projekt umsetzen und nach seinen Anforderungen anpassen. Mittlerweile wurden bereits einige Experimente und Änderungen mithilfe des Frameworks vom Projekt vorgenommen. So spielt B-Human zum Zeitpunkt der Evaluation mit vier Threads und nicht mehr mit zwei. Auch die dabei aufgedeckten Probleme und Fehler sowie weitere Aspekte der Evaluation sollen im Folgenden betrachtet und erläutert werden.

6.1 Implementierung des Konzepts

Hier wird zunächst die Implementierung des Konzepts evaluiert. Die Idee, die Realisierung in mehrere Teilschritte zu gliedern, hat sich als sinnvoll herausgestellt, da so der Überblick nicht verloren ging und funktionsfähige Zwischenergebnisse die Fehlersuche vereinfachten sowie die Motivation hochhielten. Der daraus entstandene Overhead für die Kompatibilität hat Zeit gekostet, allerdings überwiegt dies nicht die Vorteile, die daraus entstanden sind. Dies ermöglichte das Debuggen jeder Komponente vor und nach der Implementierung, um auf gleiches Gesamtverhalten zu prüfen. Ein Beispiel ist die Kommunikation. Dabei wurden

die Nachrichten vorher und hinterher auf ihre Gleichheit geprüft. Da sich so eine Prüfung nicht gut darstellen lässt, wird hier darauf verzichtet.

Die Anforderung, dass das Framework einfacher zu verstehen sein soll, lässt sich nicht trivial beantworten, da dies aus den UML-Diagrammen in Abschnitt 3.5 und Kapitel 5 nicht einfach entnommen werden kann. Was jedoch entnommen werden kann, ist, dass die Tiefe geringer geworden ist. So ist die Anzahl der Klassen von acht auf vier geschrumpft (Abbildung 6.1).

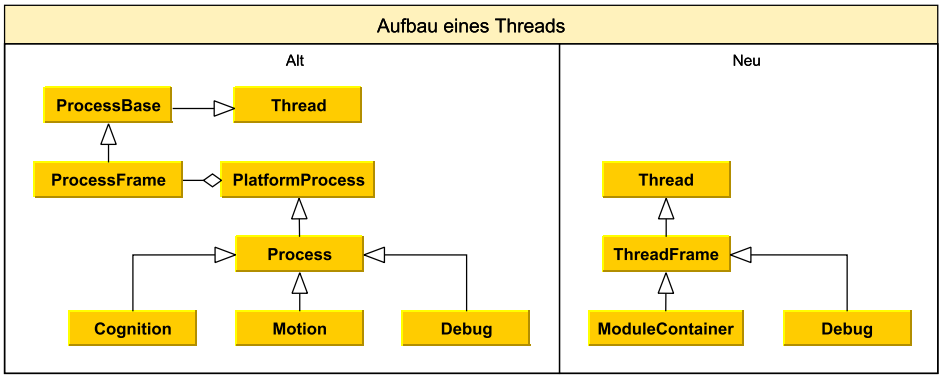


Abbildung 6.1 Das Klassendiagramm beschreibt links den alten und rechts den neuen Aufbau eines Threads. Die UML-Notation wurde um Funktionen und Member der Klassen reduziert, da diese in Abbildung 3.5, 3.8 und 5.4 beschrieben werden.

`ThreadFrame` ist zwar größer als `Process`, es handelt sich dabei aber hauptsächlich nur um kleine Funktionen, die vorher teilweise in `PlatformProcess` untergebracht waren. Auch `Sender` und `Receiver` sind einfacher geworden, da sie nun einfache Objekte und keine selbst verwaltenden Listen mehr sind. Außerdem wurden nicht genutzte Funktionalitäten entfernt. Modulverarbeitende Threads sind nun deutlich einfacher zu verwalten. Zum Hinzufügen eines solchen Threads genügt es, ihn in der Konfigurationsdatei zu definieren, statt wie bisher eine neue Klasse zu implementieren. Zur Laufzeit wird im neuen Framework dann pro Thread die gleiche Klasse instanziiert. Auch die Zuordnung von Modulen zu Threads ist jetzt flexibler und leichter nachvollziehbar, da sie ebenfalls nicht länger im Quelltext festgeschrieben ist, sondern sowohl offline, als auch zur Laufzeit konfiguriert werden kann. Es ist nicht gelungen, alle threadspezifischen Anweisungen komplett in Modulen unterzubringen. Aus diesem Grund ist mit `FrameExecutionUnit` ein neues Konstrukt dazu gekommen, welches sich – wie bisher Threads – registriert, woraufhin eine in der Konfigurationsdatei angegebene Instanz erstellt wird. Diese stellen aber nur ein Übergangskonstrukt dar, bis eine bessere Lösung gefunden wurde, da dies im Rahmen der Arbeit nicht gelungen ist. Außerdem sind sie bisher klein und gut überschaubar.

Die Konfigurationsdatei erlaubt es nun zusätzlich, nicht alle Threads laufen zu lassen, da diese dort angegeben werden, zusammen mit einer passenden `FrameExecutionUnit`. Dabei können `FrameExecutionUnits` auch mehrfach instanziiert werden. Somit ist es möglich, einen Thread einfach aus der Datei zu entfernen und die Software neu zu starten, und diese läuft ohne diesen, sofern weiterhin alle Abhängigkeiten aufgelöst werden können.

Die Anzahl der Threads ist zur Laufzeit aber weiterhin fest. Diese Entscheidung wurde getroffen, da sie einfacher zu implementieren ist. Ebenfalls kann die Anzahl der Threads, durch das Ändern der Konfiguration, gefolgt von einem einfachen Neustart der Software, geändert werden. Das ist ausreichend, da diese Funktionalität bei B-Human nur für Testphasen und nicht in Wettkampfspielen benötigt wird. Während der Implementierung wurde allerdings nicht bedacht, dass es zur Laufzeit nötig sein könnte, neue Kommunikationsverbindungen aufzubauen. Dies wird aber nötig, sobald Module zur Laufzeit ausgetauscht werden, die Daten über eine nicht existierende Verbindung benötigen. Beide angesprochenen Probleme könnten zum Beispiel über einen statischen Zugriff auf `Robot` gelöst werden, falls dies gewünscht ist. Dies hätte aber unter anderem Folgen für den `ModuleGraphCreator`, welcher annimmt, dass die Anzahl konstant ist.

Dadurch, dass `Debug` jetzt das Ausrechnen der Modulreihenfolge und der Kommunikation übernimmt, konnte der Ausfall von mehreren Zyklen beim Wechsel der Konfiguration zur Laufzeit behoben werden. Dies wurde getestet, indem mithilfe der Debug-Verbindung ein Modul in der Software eines gehenden Roboters angeschaltet wurde. Das führt dazu, dass alles neu berechnet wird. Bei der Verwendung des alten Frameworks kommt der Roboter dabei ins Stocken und würde Gefahr laufen umzufallen. Mit dem neuen Framework ist keine Beeinträchtigung mehr zu beobachten. Dadurch ist es auch, im Gegensatz zum bisherigen Framework, nicht mehr möglich, dass ein Thread die neue Modulkonfiguration nutzt und ein anderer auf der vorherigen bleibt, da die neue für ihn nicht valide ist. Es werden somit nur alle Threads zugleich aktualisiert, und zwar nur, sofern die aktuelle Konfiguration für jeden Thread gültig ist.

Im Folgenden wird näher auf die Entwicklung des `ModuleGraphCreator` eingegangen. Vor der Entwicklung des Algorithmus (Unterabschnitt 5.3.4) zum Datenaustausch wurden Unit-Tests geschrieben, die das Verhalten des bisherigen Algorithmus prüfen. Dazu wurde Google Test ([Google, 2019]) verwendet. Diese Tests sind entsprechend der neuen Struktur erweitert worden. Die Implementierung geschah dann zunächst gemäß dieser Tests, um die Funktionalität wiederherzustellen. Mit weiteren Tests, die mehr Threads und Aliasse verwenden, wurde dann die neue Funktionalität implementiert. Während der Entwicklung wurde auch mit dem B-Human-System getestet, ob keine Fälle übersehen wurden. Außerdem gibt es Tests für alle Fehlerfälle. Die Tests weisen aktuell eine 100% Pfadabdeckung auf. Dies bedeutet, dass jede Quelltextzeile von mindestens einem Test ausgeführt wird. Nach Beendigung der Implementierung wurde die Unterstützung der bisherigen Implementierung für die Tests mit nur zwei Threads entfernt, da diese nicht mehr benötigt wird.

Der Austausch läuft somit automatisch ab. Ein Problem dabei ist, dass jeder Alias eine Klasse im Quelltext ist, wodurch die Konfigurierbarkeit eingeschränkt wird. Hierfür ist aktuell aber auch keine andere Lösung bekannt. Folglich sollte diese Funktionalität zum größtmöglichen Erhalt der Konfigurierbarkeit so sparsam wie möglich eingesetzt werden. Die in der Implementierung getroffene Annahme, dass es eine Repräsentation geben könnte, die zwar selbst bereitgestellt wird, aber auch ein Alias sein könnte, gibt es bei B-Human zum Beispiel mit der Re-

präsentation `MotionInfo` aus dem Thread `Motion`. Sollte es einmal eine Repräsentation `Info` geben, wäre `MotionInfo` nicht mehr namentlich eindeutig. Eine vorgeschlagene Lösung ist, dass `MotionInfo` dann als `MotionMotionInfo` erwartet wird. Hierfür müsste der Algorithmus zum Ausrechnen der Aliasse entsprechend angepasst werden. Dies würde dazu führen, dass das Erwarten einer Repräsentation, die zufällig `<Thread-Name><Repräsentations-Name>` als Namensschema hat, nicht mehr intuitiv ist, sobald es eine Repräsentation `Repräsentations-Name` gibt. Außerdem würde dies möglicherweise zu Problemen mit der Programmiersprache führen, wenn es mehrere Klassen mit dem gleichen Namen gibt.

Eine weitere Einschränkung der Konfigurierbarkeit stellt die Möglichkeit im B-Human-System dar, dass ein Modul auch angeben kann, eine Repräsentation zu erwarten, ohne dass sie zwangsläufig bereits im selben Zyklus berechnet sein muss. Dieser Sonderfall ist nötig, da sonst Abhängigkeitszyklen zwischen Modulen nicht aufgelöst werden könnten. Diese besonderen Anforderungen eines Moduls werden beim Verschicken nicht betrachtet, da sie aktuell nicht verwaltet werden. Wenn ein Modul somit in einen anderen Thread geschoben wird, in dem die von dem Modul erwartete Repräsentation bisher nicht bereitgestellt wird, stehen keine aktuellen Daten zur Verfügung. Ob dies der Fall ist, kann aktuell auch nicht geprüft werden. Wenn diese Besonderheit verwaltet werden würde, wäre es aber möglich, dies zu prüfen.

6.2 Die Implementierung von B-Human

B-Human hat das Framework zunächst so, wie es im Rahmen dieser Arbeit implementiert wurde, getestet. Dann wurde die Anzahl der Threads auf drei erweitert, indem die Bildverarbeitung und Modellierung getrennt wurden. Als das funktionierte, wurde die Bildverarbeitung noch mal für beide Kameras getrennt. Somit sind es aktuell vier Threads. Dabei wurden die `FrameExecutionUnit`s soweit wie möglich vereinfacht, sodass die Funktionen jetzt keine spezifischen Parameter mehr benötigen, um bestimmte Aufgaben für einen spezifischen Thread auszuführen.

Mit der Aufteilung in mehr Threads gibt es einen Thread, welcher Modellierung, Lokalisierung und Verhalten berechnet, der nicht im Takt der Hardware läuft. Dieser wartet auf eingehende Nachrichten. Dabei wird aktuell jeweils auf Daten von beiden Kamera-Threads gewartet und dann die Daten mit dem älteren Zeitstempel zuerst berechnet und direkt im Anschluss die anderen. Der Grund dafür ist, dass das B-Human-System davon ausgeht, dass die Zeitstempel fortlaufend sind und die Zeitstempel vom Anfang der Berechnung sind. Somit könnte ein Thread später anfangen zu rechnen, aber früher fertig sein und die Daten des anderen Threads würden nie verarbeitet werden, da diese immer den älteren Zeitstempel haben.

Ein weiteres Problem ist, dass ein Thread mit einer neuen Konfiguration einmal durchlaufen muss, bevor dieser valide ist und Daten empfangen kann, da vorher nicht sichergestellt ist, dass alle Datenstrukturen existieren. Aus diesem Grund werden Threads nach einer Aktualisierung ihrer Konfiguration jetzt immer sofort für einen Zyklus ausgeführt. Wäre das nicht

der Fall, käme es zu Deadlocks, da keine Daten mehr empfangen werden könnten und somit Threads nicht mehr aufwachen würden, sofern sie nicht von anderen ankommenden Daten ausgelöst würden. Die Folgen davon wurden zum aktuellen Zeitpunkt noch nicht evaluiert, sind aber auch nicht negativ aufgefallen. Eine mögliche Lösung wäre, dass die nötigen zu empfangenden Repräsentationen direkt vom Framework erstellt werden, sodass stets Daten empfangen werden können und der erste Durchlauf nach Ändern der Konfiguration problemlos abläuft.

Besonders beim Abspielen von Logs ist aufgefallen, dass nachträglich eingerichtete Verbindungen zwischen Threads unverzichtbar sind, da Repräsentationen ausgetauscht werden müssen, die normalerweise nicht ausgetauscht werden. Dies wurde gelöst, indem jetzt alle Threads eine Verbindung haben, aber nur kommunizieren, sofern sie auch Daten austauschen.

Im `ModuleGraphCreator` konnte es zu einem Fehler kommen, obwohl die Konfiguration valide war. Der Grund dafür ist, dass alle Module, die in einem Thread laufen oder liefen, vorgemerkt werden und neue erst nach Bedarf zu dieser Liste hinzugefügt wurden. Diese Liste wird allerdings unter anderem dazu benutzt, zu prüfen, ob ein Thread eine Repräsentation bereitstellen kann. Somit ist sie für einen Thread, der erst noch berechnet wird, nicht korrekt, falls dieser um neue Repräsentationen erweitert wird. Das wurde gelöst, indem jetzt alle Module vorgemerkt werden. Außerdem wurde die Bedingung für Aliasse aufgeweicht. Diese müssen jetzt nur noch strukturell identisch sein und nicht mehr voneinander erben.

Außerdem hat sich die Weiterführung der Trennung zwischen realem und simuliertem Roboter als nicht notwendig herausgestellt, da der reale Roboter eine Teilmenge des simulierten Roboters darstellt, sodass sie zusammengefasst werden können.

Ein weiteres nicht frameworkspezifisches Problem ist, dass einige Module der Bildverarbeitung Informationen aus beiden Kamerabildern verwendet haben. Hierfür mussten die Module angepasst werden. Module, die bisher in beiden Threads Infrastruktur bereitgestellt haben, wurden dabei zusammengefasst und für eine variable Anzahl an Threads generalisiert. Auch das Debugging musste an diversen Stellen angepasst und generalisiert werden, damit es mit der neuen Herausforderung umgehen kann. So muss die Modellierung zum Beispiel die Möglichkeit haben, Dinge in Koordinatensystemen der Kamera-Threads zu zeichnen. Außerdem sind viele Filter, wie die in der Modellierung genutzten Modelle, implizit auf die Anzahl der Durchläufe des Threads pro Sekunde abgestimmt. Diese beiden Faktoren schränken auch die Konfigurierbarkeit des Produktivsystems ein.

Kapitel 7

Fazit und Ausblick

Ziel der vorliegenden Arbeit war es, Teile des B-Human-Frameworks neu zu entwickeln, um mit den neuen Anforderungen wie Parallelisierbarkeit jetzt und auch in Zukunft umgehen zu können. So wurde die Thread-Infrastruktur und deren Kommunikation neu geschrieben, wobei einige Teile und Prinzipien – wie die Datenübertragung – übernommen wurden. Weiterhin wurde ein Algorithmus entwickelt, der berechnet, welche Daten Threads untereinander austauschen müssen.

Im Folgenden werden die dargelegten Ergebnisse der Arbeit kritisch analysiert und überprüft, welche Folgen sich daraus für B-Human ergeben (Abschnitt 7.1). Abschließend wird noch ein allgemeiner Blick darauf geworfen, welchen Nutzen die Arbeit außerhalb von B-Human bringen kann (Abschnitt 7.2).

7.1 Bei B-Human

Es wurde das Ziel erreicht, dass die Arbeit gut genug funktioniert, um aktiv eingesetzt zu werden. Sie hat das bisherige Framework abgelöst, es sind aber noch nicht alle Möglichkeiten integriert. Dies bezieht sich aber insbesondere auf Debuggingfeatures und Komfortfunktionalitäten. Während der Entwicklung wurde stets darauf geachtet, dass sich das Verhalten aller Komponenten im Zusammenspiel nicht verändert. Somit blieben größere Komplikationen bei der Einführung aus. Durch Tests für den entwickelten Algorithmus wurde einerseits sichergestellt, dass dieser sich für zwei Threads analog zum bisherigen verhält. Andererseits konnte durch Tests das Verhalten für mehr als zwei Threads schon im Vorhinein definiert werden. Die Tests sind dabei jedoch nicht vollständig. So wurde bereits ein Fehler im Algorithmus gefunden, welcher auf einen fehlenden Testfall zurückzuführen ist. Allerdings können die Tests als Grundlage für eine Weiterentwicklung genutzt werden, sollte der Algorithmus in Zukunft noch angepasst und optimiert werden.

Die bisherigen Entwicklungen mit dem Framework bei B-Human haben gezeigt, dass es die aktuellen Anforderungen erfüllen kann. Dazu sind allerdings kleinere Anpassungen nötig gewe-

sen. Um das Framework auf Dauer gut nutzbar zu machen, sind auch noch weitere Anpassungen und Optimierungen nötig. So muss aktuell nach dem Einspielen einer neuen Konfiguration immer ein Zyklus durchlaufen werden, um wieder einen validen Zustand herzustellen. Dies ist nötig, da alle Repräsentationen konstruiert sein müssen, bevor wieder Daten empfangen werden können. Das Erstellen könnte auch das Framework übernehmen und so verhindern, dass es möglicherweise ein Zyklus ausführen muss, der eigentlich nicht ausgeführt werden soll. Auch Aliasse haben noch Optimierungspotenzial. Aktuell wird jedes Mal eine Klasse erstellt und diese dem Modul als Erwartung gegeben. Wenn ein Thread in einer Konfigurationsdatei umbenannt wird oder es mehrere Konfigurationen gibt, die unterschiedliche Aliasse erwarten, geht das aktuell nicht. Eine mögliche Lösung wäre, dass in der Konfigurationsdatei angegeben werden kann, welche Repräsentationen unter welchem Bezeichner in einem anderen Thread verwendet werden. Dies löst direkt das Problem, wenn in einem Thread nur eine Repräsentation benötigt wird. Einige Module benötigen bei B-Human die gleiche Repräsentation von beiden Kameras. Dafür könnte `Other<Repräsentations-Name>` weiter als Klasse angegeben werden. Somit ist aber der Name des Threads nicht mehr relevant und das Framework kann das richtig setzen. Den schwierigen Fall stellt ein Modul dar, welches zum Beispiel die gleichen Daten beider Kameras erwartet. Eine mögliche Lösung wäre, dass das Framework stets die jeweils passenden Daten bereitstellt. Dies hieße bei B-Human beispielsweise, dass, wenn ein Modul in der Modellierung Daten aus der oberen Kamera verarbeitet, sämtliche dem Modul bereitgestellten Daten aus dem Thread der oberen Kamera stammen. Für dieses Problem gibt es möglicherweise aber auch noch andere und bessere Lösungen.

In der Geschichte des Frameworks (Unterabschnitt 3.5.1) ist es nicht das erste Mal, dass mehr Parallelisierung versucht wird. Dabei hatte sich zuvor herausgestellt, dass die bisherige Anzahl an Threads die Beste war. Die Notwendigkeit eines erneuten Versuchs mit mehr Parallelität ist durch die gestiegene Rechenleistung des *NAO V6* begründet. Ein weiterer Grund ist, dass es damals nur eine Kamera gab. Der *NAO* verfügt aber schon seit mehreren Generationen über zwei Kameras. Diese können mit mehr Rechenkernen vollkommen parallel bearbeitet werden, wodurch der Zusatzaufwand für mehr Kommunikation relativiert wird. Weiterhin können komplexe Aufgaben, die nicht für jede Eingabe zu berechnen sind, mit einem Thread auf einen anderen Rechenkern ausgelagert werden. Somit wird das Benutzen von komplexeren Algorithmen zum Beispiel in der Modellierung oder Lokalisierung erlaubt. Außerdem könnte es Threads gegeben, die Analysen oder Statistiken führen und somit zur Effektivität in der Fehlersuche und Bewertung von Änderungen und Spielverläufen beitragen. Dadurch müssen für diese Aufgabe nicht extra Daten gespeichert und ausgewertet werden, sondern nur deren Ergebnisse, die direkt vorliegen. Die nötige freie Rechenleistung war dafür bisher nicht gegeben und hätte nicht genutzt werden könnten.

Die Berechnung der Reihenfolge, in der Module ausgeführt werden sollen, wurde nicht verändert. Diese war bisher der langsamste Teil, wenn sich die Modulkonfiguration verändert hat. Dadurch, dass die Anzahl der Module pro Thread gesunken ist, ist dies jetzt schneller. Trotzdem wäre eine algorithmisch optimalere und schnellere Lösung wünschenswert.

7.2 Allgemein

Abschließend ist zu analysieren, inwiefern das Ergebnis der vorliegenden Arbeit außerhalb von B-Human Anwendung finden kann. Zuallererst wird die entstandene Software im Rahmen des jährlichen B-Human-Coderelease veröffentlicht werden und ist somit öffentlich verfügbar. Es gibt immer wieder neue Teams, die diese Veröffentlichungen als Grundlage für ihre eigene Entwicklung benutzen. Somit stellt die Veröffentlichung für diese einen Mehrwert dar. Außerdem wird die Veröffentlichung möglicherweise auch außerhalb des RoboCups verwendet. Prinzipiell kann jeder sich somit die Software herunterladen und mit dem Framework seine eigene Software für Roboter schreiben. Ein Schritt, der dies begünstigen würde, wäre, wenn das Framework von der Implementierung der Fußballsoftware getrennt werden würde. Dies haben zum Beispiel die HULKs getan [Adikari u. a., 2017]. Somit müsste nur noch dieser Teil aus der Veröffentlichung entnommen werden, um mit einem komplett unabhängigen Projekt starten zu können. Dies würde auch die Möglichkeit bieten, das Framework zum Beispiel für Multithreading in ROS-Nodes zu nutzen. So ist es möglich, die Aufgaben einer Node sehr fein zu unterteilen und einfach zur Laufzeit auszutauschen, wie es bei B-Human der Fall ist und die Vorteile von ROS nutzen zu können.

Es bleibt abzuwarten, ob sich dieses Framework auf Dauer durchsetzen kann, oder ob es schlussendlich doch von einem anderen abgelöst wird. Zum aktuellen Zeitpunkt bildet die in dieser Arbeit entwickelte Erneuerung des Frameworks eine gute Grundlage für B-Human, um weiterhin erfolgreich Fußball spielen zu können.

Literaturverzeichnis

[Adikari u. a. 2017] Adikari, Darshana ; Felbinger, Georg ; Hasselbring, Arne ; Konda, Yuria ; Kost, René ; Loth, Pascal ; Peters, Lasse ; Riebesel, Nicolas ; Schattschneider, Thomas ; Warmuth, Felix ; Wege, Felix: *HULKs Team Research Report 2017*. 2017. – Nur online verfügbar: `https://www.hulks.de/_files/TRR_2017.pdf`

[B-Human 2018] B-Human: *B-Human | RoboCup Standard Platform League. Offizielle Webseite des Teams B-Human*. `https://www.b-human.de`, 2018. – Abgerufen am 17.11.2018

[Bäuml u. Hirzinger 2006] Bäuml, Berthold ; Hirzinger, Gerd: Agile Robot Development (aRD): A Pragmatic Approach to Robotic Software. In: *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006. – ISSN 2153–0858, S. 3741–3748

[Bruyninckx 2001] Bruyninckx, Herman: Open robot control software: the OROCOS project. In: *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)* Bd. 3, 2001. – ISBN 0–7803–6576–3, S. 2523–2528

[Bäuml u. Hirzinger 2008] Bäuml, Berthold ; Hirzinger, Gerd: When hard realtime matters: Software for complex mechatronic systems. In: *Robotics and Autonomous Systems* 56 (2008), Nr. 1, S. 5 – 13. `http://dx.doi.org/10.1016/j.robot.2007.09.017`. – DOI 10.1016/j.robot.2007.09.017. – ISSN 0921–8890

[Burkhard u. a. 2002] Burkhard, Hans-Dieter ; Düffert, Uwe ; Hoffmann, Jan ; Jüngel, Matthias ; Lötzsch, Martin ; Brunn, Ronnie ; Kallnik, Martin ; Kuntze, Nicolai ; Kunz, Michael ; Petters, Sebastian ; Risler, Max ; Stryk, Oskar von ; Koschmieder, Nils u. a.: *GermanTeam RoboCup 2002*. 2002

[Edwin 2014] Edwin, Njeru M.: Software Frameworks, Architectural and Design Patterns. In: *Journal of Software Engineering and Applications* 07 (2014), S. 670–678. `http://dx.doi.org/10.4236/jsea.2014.78061`. – DOI 10.4236/jsea.2014.78061

[Forero u. a. 2014] Forero, Leonardo L. ; Yáñez, José M. ; Solar, Javier Ruiz-del: Integration of the ROS Framework in Soccer Robotics: The NAO Case. In: Behnke, Sven (Hrsg.) ; Veloso, Manuela (Hrsg.) ; Visser, Arnoud (Hrsg.) ; Xiong, Rong (Hrsg.): *RoboCup 2013: Robot World Cup XVII*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2014. – ISBN 978–3–662–44468–9, S. 664–671

[Fujita u. a. 1999] Fujita, Masahiro ; Zrehen, Stephane ; Kitano, Hiroaki: A Quadruped Robot for RoboCup Legged Robot Challenge in Paris '98. In: Asada, Minoru (Hrsg.) ; Kitano, Hiroaki (Hrsg.): *RoboCup-98: Robot Soccer World Cup II*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1999. – ISBN 978–3–540–48422–6, S. 125–140

[Google 2018] Google: *Google Protocol Buffers*. `https://developers.google.com/protocol-buffers/`, 2018. – Abgerufen am: 31.01.2019

[Google 2019] Google: *Google Test*. `https://github.com/google/googletest`, 2019. – Abgerufen am: 06.03.2019

[Hammer u. Bäuml 2013] Hammer, Tobias ; Bäuml, Berthold: The highly performant and realtime deterministic communication layer of the aRDx software framework. In: *2013 16th International Conference on Advanced Robotics (ICAR)*, 2013, S. 1–8

[Hornung u. Humanoid Robots Lab Freiburg 2010] Hornung, Armin ; Humanoid Robots Lab Freiburg: *ROS Aldebaran Nao*. `http://wiki.ros.org/nao`, 2010. – Abgerufen am: 31.01.2019

[Jagannathan u. a. 1989] Jagannathan, Vasudevan (Hrsg.) ; Dodhiawala, Rajendra (Hrsg.) ; Baum, Lawrence S. (Hrsg.): *Blackboard Architectures and Applications*. Academic Press, Boston, 1989

[Mamantov u. a. 2015] Mamantov, Elizabeth ; Silver, William ; Dawson, William ; Chown, Eric: RoboGrams: A Lightweight Message Passing Architecture for RoboCup Soccer. In: Bianchi, Reinaldo A. C. (Hrsg.) ; Akin, H. L. (Hrsg.) ; Ramamoorthy, Subramanian (Hrsg.) ; Sugiura, Komei (Hrsg.): *RoboCup 2014: Robot World Cup XVIII*. Cham : Springer International Publishing, 2015. – ISBN 978–3–319–18615–3, S. 306–317

[Metta u. a. 2006] Metta, Giorgio ; Fitzpatrick, Paul ; Natale, Lorenzo: YARP: Yet another robot platform. In: *International Journal of Advanced Robotic Systems* 3 (2006). `http://dx.doi.org/10.5772/5761`. – DOI 10.5772/5761

[Nao-Team HTWK Leipzig 2018] Nao-Team HTWK Leipzig: *HTWKVision*. `https://github.com/NaoHTWK/HTWKVision`, 2018. – Abgerufen am: 21.01.2019

[Open Robotics 2018] Open Robotics: *ROS - Documentation*. `http://wiki.ros.org`, 2018. – Abgerufen am: 14.09.2018

[Quigley u. a. 2009] Quigley, Morgan ; Conley, Ken ; Gerkey, Brian P. ; Faust, Josh ; Foote, Tully ; Leibs, Jeremy ; Wheeler, Rob ; Ng, Andrew Y.: ROS: an open-source Robot Operating System. In: *ICRA Workshop on Open Source Software*, 2009

[RoboCup Technical Committee 2018] RoboCup Technical Committee: *RoboCup Standard Platform League (NAO) Rule Book*. `https://spl.robocup.org/wp-content/uploads/downloads/Rules2018.pdf`, 2018. – Abgerufen am: 31.01.2019

[Röfer 2003] Röfer, Thomas: An Architecture for a National RoboCup Team. In: Kaminka, Gal A. (Hrsg.) ; Lima, Pedro U. (Hrsg.) ; Rojas, Raúl (Hrsg.): *RoboCup 2002: Robot Soccer World Cup VI*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003. – ISBN 978–3–540–45135–8, S. 417–425

[Röfer u. Laue 2014] Röfer, Thomas ; Laue, Tim: On B-Human's Code Releases in the Standard Platform League – Software Architecture and Impact. In: Behnke, Sven (Hrsg.) ; Veloso, Manuela (Hrsg.) ; Visser, Arnoud (Hrsg.) ; Xiong, Rong (Hrsg.): *RoboCup 2013: Robot World Cup XVII* Bd. 8371, Springer Berlin Heidelberg, 2014 (Lecture Notes in Artificial Intelligence), S. 648 – 655

[Röfer u. a. 2007] Röfer, Thomas ; Laue, Tim ; Brose, Jörg ; Göhring, Daniel ; Jüngel, Matthias ; Risler, Max: GermanTeam 2007. In: Visser, Ubbo (Hrsg.) ; Ribeiro, Fernando (Hrsg.) ; Ohashi, Takeshi (Hrsg.) ; Dellaert, Frank (Hrsg.): *RoboCup 2007: Robot Soccer World Cup XI Preproceedings*. Atlanta, GA, USA : RoboCup Federation, 2007

[Röfer u. a. 2017] Röfer, Thomas ; Laue, Tim ; Bülter, Yannick ; Krause, Daniel ; Kuball, Jonas ; Mühlenbrock, Andre ; Poppinga, Bernd ; Prinzler, Markus ; Post, Lukas ; Roehrig, Enno ; Schröder, René ; Thielke, Felix: *B-Human Team Report and Code Release 2017*. 2017. – Nur online verfügbar: `http://www.b-human.de/downloads/publications/2017/coderelease2017.pdf`

[Röfer u. a. 2008] Röfer, Thomas ; Laue, Tim ; Burchardt, Armin ; Damrose, Erik ; Fritsche, Martin ; Müller, Judith ; Rieskamp, Andrik: B-Human Team Description for RoboCup 2008. In: Iocchi, Luca (Hrsg.) ; Matsubara, Hitoshi (Hrsg.) ; Weitzenfeld, Alfredo (Hrsg.) ; Zhou, Changjiu (Hrsg.): *RoboCup 2008: Robot Soccer World Cup XII Preproceedings*. Suzhou, China : RoboCup Federation, 2008

[Röfer u. a. 2018] Röfer, Thomas ; Laue, Tim ; Hasselbring, Arne ; Heyen, Jannik ; Poppinga, Bernd ; Reichenberg, Philip ; Roehrig, Enno ; Thielke, Felix: *B-Human Team Report and Code Release 2018*. 2018. – Only available online: `http://www.b-human.de/downloads/publications/2018/CodeRelease2018.pdf`

[Shin u. Ramanathan 1994] Shin, Kang G. ; Ramanathan, Parameswaran: Real-time computing: a new discipline of computer science and engineering. In: *Proceedings of the IEEE* 82 (1994), Nr. 1, S. 6–24. `http://dx.doi.org/10.1109/5.259423`. – DOI 10.1109/5.259423. – ISSN 0018–9219

[SoftBank Robotics 2018a] SoftBank Robotics: *NAO - Construction*. `http://doc.aldebaran.com/2-4/family/robots/dimensions_robot.html`, 2018. – Abgerufen am: 26.11.2018

[SoftBank Robotics 2018b] SoftBank Robotics: *NAO - Developer Guide*. `http://doc.aldebaran.com/2-4/family/index.html`, 2018. – Abgerufen am: 12.09.2018

[SoftBank Robotics 2018c] SoftBank Robotics: *NAO H25*. http://doc.aldebaran.com/2-4/family/nao_h25/index_h25.html, 2018. – Abgerufen am: 26.11.2018

[SoftBank Robotics 2018d] SoftBank Robotics: *NAO V5 - Motherboard*. http://doc.aldebaran.com/2-4/family/robots/motherboard_robot.html, 2018. – Abgerufen am: 26.11.2018

[SoftBank Robotics 2018e] SoftBank Robotics: *NAO V6 - Developer Guide*. http://doc.aldebaran.com/2-8/family/nao_technical/index_naov6.html, 2018. – Abgerufen am: 12.09.2018

[SoftBank Robotics 2018f] SoftBank Robotics: *NAO V6 - Motherboard*. http://doc.aldebaran.com/2-8/family/nao_technical/motherboard_naov6.html, 2018. – Abgerufen am: 26.11.2018

[Tanenbaum u. Bos 2016] *Kapitel* 8. In: Tanenbaum, Andrew S. ; Bos, Herbert: *Moderne Betriebssysteme*. Pearson Education Deutschland GmbH, 2016. – ISBN 978–3–86894–270–5, 628 – 683

[The RoboCup Federation 2016a] The RoboCup Federation: *A Brief History of RoboCup*. https://www.robocup.org/a_brief_history_of_robocup, 2016. – Abgerufen am: 22.11.2018

[The RoboCup Federation 2016b] The RoboCup Federation: *RoboCup Objective*. https://www.robocup.org/objective, 2016. – Abgerufen am: 20.11.2018

[The RoboCup Federation 2016c] The RoboCup Federation: *Soccer Standard Platform League*. http://www.robocup.org/leagues/5, 2016. – Abgerufen am: 20.11.2018

[Wikidot.com 2018] Wikidot.com: *ZeroMQ - Distributed Messaging*. http://zeromq.org/, 2018. – Abgerufen am: 31.01.2019

[Wikipedia 2018a] Wikipedia: *Framework — Wikipedia, Die freie Enzyklopädie*. https://de.wikipedia.org/w/index.php?title=Framework&oldid=179886028, 2018. – Abgerufen am: 13.03.2019

[Wikipedia 2018b] Wikipedia: *Prozess (Informatik) — Wikipedia, Die freie Enzyklopädie*. https://de.wikipedia.org/w/index.php?title=Prozess_(Informatik)&oldid=181106308, 2018. – Abgerufen am: 13.03.2019

[Wolf 2009] *Kapitel* 26.3. In: Wolf, Jürgen: *C von A bis Z*. 3. Rheinwerk Computing, 2009. – ISBN 978–3–8362–1411–7

[Wünsch 2013] Wünsch, Michael: *Portierung und Emulation individueller Roboter-Firmware auf Fremdsystemen*, HTWK Leipzig, Masterarbeit, 2013

[Zöbel u. Hogenkamp 1988] *Kapitel* 1. In: ZÖBEL, Dieter ; HOGENKAMP, Horst: *Sequentielle und parallele Prozesse*. Wiesbaden : Vieweg+Teubner Verlag, 1988. – ISBN "978–3–322–94670–6, 11–18