Leichtgewichtiges Multithreading-Framework für Robotikanwendungen

Bachelorarbeit

Jan Fiedler
Matrikelnummer: 4235072

18. März 2019

Universität Bremen

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

1 Einleitung  
1.1 Motivation ................................................. 1  
1.2 Ziel ......................................................... 2  
1.3 Aufbau der Arbeit ........................................... 2  

2 Hintergrund  
2.1 RoboCup ......................................................... 3  
2.2 B-Human ......................................................... 3  
2.3 Der NAO ......................................................... 4  

3 Grundlagen  
3.1 Software-Framework ........................................... 5  
3.2 Sequenzielle und parallele Ausführung ......................... 5  
3.3 Prozesse und Threads ......................................... 6  
3.4 Synchron und asynchrone Kommunikation ......................... 6  
3.5 B-Human-Framework  
3.5.1 Entstehung .................................................. 7  
3.5.2 Aufbau ....................................................... 10  
3.5.3 Anwendungsszenarien ...................................... 15  

4 Verwandte Arbeiten  
4.1 Verwandte Arbeiten in der SPL .................................. 19  
4.2 ROS ............................................................. 21  

5 Realisierung  
5.1 Anforderungen ................................................... 23
5.2 Konzept .......................... 24
5.3 Implementierung .......................... 25
  5.3.1 Programmmstart ......................... 27
  5.3.2 Kommunikation ......................... 27
  5.3.3 Threads ......................... 28
  5.3.4 ModuleGraphCreator ......................... 30

6 Evaluation .......................... 35
  6.1 Implementierung des Konzepts ......................... 35
  6.2 Die Implementierung von B-Human ......................... 38

7 Fazit und Ausblick .......................... 41
  7.1 Bei B-Human ......................... 41
  7.2 Allgemein ......................... 43

Literaturverzeichnis ......................... 49
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


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

1.2 Ziel


1.3 Aufbau der Arbeit

Kapitel 2

Hintergrund


2.1 RoboCup


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].

2.3 Der NAO


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


3.1 Software-Framework


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
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]


3.4 Synchrone und asynchrone Kommunikation

Als Kommunikation wird in der Informatik das gegenseitige Senden und Empfangen von Daten bezeichnet. Bei synchrone 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-

### 3.5 B-Human-Framework


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.

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


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].


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].


3.5.2 Aufbau


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

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 Einstiegs punkt in die Software und das Registrieren der Prozesse dargestellt.

Prozesse als Threads

Kommunikation


Prozesse


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

```javascript
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


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.

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


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


4.2 ROS


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-
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


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.


```
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.


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).

```plaintext
defaultRepresentations = [
  RepresentationXY,
];
threads = [
  {
    name = Upper;
    priority = 0;
    debugReceiverSize = 2800000;
    debugSenderId = 5200000;
    debugSenderIdInfrastructureSize = 100000;
    executionUnit = Cognition;
    requiredThreads = [
      Motion,
    ];
    representationProviders = [
      {representation = Representation; provider = Provider;},
      {representation = Representation2; provider = Provider2;},
    ];
  },
  {
    name = Motion;
    priority = 20;
    debugReceiverSize = 500000;
    debugSenderId = 50000;
    debugSenderIdInfrastructureSize = 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


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

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 ModulePackages oder Message-Queues 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

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]).


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.


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 Aliassen. Die Schreibweise entspricht der von Konfigurationsdateien bei B-Human.
Das Klassendiagramm des `ModuleGraphCreator` und des `ModuleGraphRunner`

Die UML-Notation wurde dazu angepasst. Die äußeren Kästen beschreiben bei 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.

Die Funktion `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
2    Input: Die Konfiguration (Quelltext 5.2).
3    Output: Ob die Konfiguration gültig ist.
4    if threads == ∅ and defaultRepresentations ≠ ∅ then ERROR(); return false;
5    else if threads == ∅ then return true;
6    for index = 0 to threads.size() do /* Calculate all threads. */
7        list<index> requiredThreads;
8        foreach required : threads[index].requiredThreads do /* Find required threads. */
9            if threads.find(required) then
10               requiredThreads.add(required.index)
11            else ERROR(); return false;
12    foreach rp : threads[index].representationProviders do
13        if not modulesPerThread.find(rp.provider) then
14            modulesPerThread.add(rp.provider)
15            else ERROR(); return false;
16        if not find(rp.provider, rp.representation) then ERROR(); return false;
17        // Calculate shared.
18        if not calcShared2(config, index, ..., received[index]) then return false;
19    foreach thread : requiredThreads do /* Set the senders to the receivers. */
20        sent[thread][index] = received[index][thread];
21        for i = 0 to sent[thread][index].size() do /* Modify sent if alias. */
22            if isProvided(config, sent[thread][index][i]) then continue;
23            // Representation is alias, find right representation.
24            foreach rp : threads[thread].representationProviders do
25                if alias == rp.representation then
26                   foreach requirement : rp.provider.info do
27                        if alias == requirement.representation then
28                            sent[thread][index][i] = requirement.representation;
29                        leaveAllLoops;
30                // Default can provide everything that exists, but only that.
31            foreach representation : defaultRepresentations do
32                if not providedByAModule(modulesPerThread, representation) then
33                   ERROR(); return false;
34    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.

**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.


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.


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 instanziert 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.


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.


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 FrameExecutionUnits soweit wie möglich vereinfacht, sodass die Funktionen jetzt keine spezifischen Parameter mehr benötigen, um bestimmte Aufgaben für einen spezifischen Thread auszuführen.


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, vorge- merkt 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.

Kapitel 7

Fazit und Ausblick


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


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-
Fazit und Ausblick


7.2 Allgemein


Literaturverzeichnis

[Adikari u. a. 2017] Adikari, Darshana ; Felbinger, Georg ; Hasselbring, Arne ; Konda, Yuria ; Kost, René ; Loth, Pascal ; Peters, Lasse ; Riebeisel, Nicolas ; Schattschneider, Thomas ; Warmuth, Felix ; Wege, Felix: HULKs Team Research Report 2017. 2017.


[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


[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


[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. 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


