# Dissertation

# Constraint basierte Modellierung von Komponentennetzwerken am Beispiel eines autonomen Unterwasserfahrzeugs

**Autor:**
**Matthias Goldhoorn**

**Universität Bremen**
**Fachbereich 3, Informatik und Mathematik**

| | |
|---|---|
| **Erstgutachter:** | **Prof. Dr. Frank Kirchner** |
| **Zweitgutachter:** | **Prof. Dr.-Ing. Udo Frese** |
| **Datum des Kolloquiums:** | **17.08.2017** |

# Vorwort

Ich möchte diese Dissertation mit einer sehr harten Aussage beginnen, die vermutlich die meisten Leser überraschen wird, aber dennoch eine wichtig zu verstehende Grundlage dieser Arbeit ist.

*Diese Dissertation löst keine Probleme, die Robotersysteme sonst nicht lösen könnten, kein System wird neue Fähigkeiten erwerben.*

Warum ist das so? Weil alle Aufgaben, die Roboter mit den Konzepten, die in dieser Arbeit vorgestellt werden, auch mittels Assembler-Programmierung gelöst werden könnten. Dennoch kommen in der Entwicklung höhere Abstraktionssprachen, wie beispielsweise C++, zum Einsatz. Diese Arbeit gliedert sich somit in den Bereich der Frameworks für robotische Systeme ein. Die vorgestellten Konzepte sollen helfen die zunehmend größer werdende Komplexität von Robotersystemen effektiv zu behandeln.

Diese Dissertation ist zweigeteilt aufgebaut. Der erste Teil beschäftigt sich mit der Analyse des aktuellen Standes der Technik, um Schwachstellen zu identifizieren, während der zweite Teil der Dissertation in die Tiefe gehend ein neues Konzept zum Lösen typischer Probleme der Robotik vorstellt.

Der erste Teil stellt den Entwicklungsprozess und wesentliche Designentscheidungen dar, die nötig sind, um robuste Systeme zu entwickeln, die über längere Zeit vollautonom operieren können. Dazu wurde das System Avalon entwickelt, das als Evaluationsplattform dient.

Der zweite Teil widmet sich den Problemen, die im ersten Teil aufgezeigt wurden. Konkret werden Probleme der komponentenbasierten Softwareentwicklung gelöst und es werden bewusst bestehende Paradigmen gebrochen, die sich als nicht zielführend herausgestellt haben. Im Laufe der Arbeit werden Prinzipien der komponentenbasierten Systementwicklung dargestellt und es wird gezeigt, wie ein modellgetriebenes, constraintbasiertes Verfahren die Integration komplexer Komponentennetzwerke deutlich vereinfachen kann und zugleich zu einer Verifikation des Netzwerkes selbst führt.

# Danksagung

Ich möchte an dieser Stelle den Personen und Gruppen danken, ohne deren Hilfe diese Arbeit in dieser Form nicht möglich gewesen wäre. Einige unterstützen mich nicht nur in den Letzten Jahren während meiner Dissertation, sondern förderten mich schon in den Jahren davor. Einer dieser Personen ist mein Doktorvater Prof. Dr. Frank Kirchner. Er unterstützte sowohl meine Aufnahme an der Universität Bremen als Student, als auch als WiMi und Doktorand in seiner Arbeitsgruppe. Dank seiner Unterstützung war es mir möglich mein Studium als Diplominformatiker in seiner Arbeitsgruppe und dem DFKI Bremen erfolgreich abzuschließen und meine Entwicklung bis zu dieser Dissertation zu vollziehen. Weiterhin möchte ich Prof. Dr.-Ing. Udo Frese danken, der sowohl meine Forschung in der Robotik unterstützte, als auch stets ein offenes Ohr für akute Probleme besaß und diese Arbeit neben Prof. Dr. Kirchner maßgeblich lenkte.

Insbesondere möchte ich meiner Frau Malgorzata Goldhoorn danken. Ohne Ihr Verständnis und Unterstützung während der harten und entbehrungsreichen Zeit, sowohl im Studium als auch während der Dissertation, wäre ich nicht in der Lage gewesen diese Arbeit zu vollenden.

Da diese Arbeit durch das Graduiertenkolleg System Design "SyDe" aus Mitteln des Zukunftskonzepts der Universität Bremen im Rahmen der Exzellenzinitiative des Bundes und der Länder unterstützt wurde, möchte ich an dieser Stelle Prof. Dr. Rolf Drechsler danken. Mein Dank gilt einerseits dem Vertrauen, das er in mich setzte, indem er mich in die Graudiertenschule aufnahm, als auch für die hervorragende Leitung der Schule selbst. Ohne das Graduiertenkolleg hätte sich insbesondere der theoretische Teil dieser Arbeit in eine andere Richtung entwickelt.

Meinen höchsten Respekt möchte an dieser Stelle Herren Dr. Sylvain Joyeux aussprechen. Er hat durch seine Dissertation und seine Arbeiten den wissenschaftlichen Grundstein zu dieser Arbeit gelegt. Ohne seine Arbeiten sowie seine persönliche Unterstützung wäre die Basis für Inhalte dieser Arbeit nicht existent. Dank des Vertrauen von Herren Prof. Dr. Frank Kirchner, mit ihm das Projekt *Avalon* zu leiten, wo ich nicht nur vielfältige Arbeiten koordinieren sondern auch leiten konnte. Ich möchte allen *Avalon* Teammitgliedern für ihre Arbeit und ihre Engagement, welches weit über das normale Maß hinausging, danken.

Zu guter Letzt möchte ich allen Kollegen der Arbeitsgruppe Robotik des Universität Bremen und des DFKIs danken. All den genannten Personen möchte ich nochmals meinen herzlichsten Dank aussprechen. Leider werden sich manche Wege trennen. Jedoch werde ich die schöne, wenngleich turbulente und erfahrungsreiche Zeit mit Euch nie vergessen. Sie wird immer stets in positiver Erinnerung bleiben.

# Inhaltsverzeichnis

**Vorwort** III

**Danksagung** V

**1 Einleitung** 1

1.1 Komponentennetzwerke . . . 2
1.2 Plan-Manager . . . 3
1.3 Abgrenzung zu anderen Arbeiten . . . 3

**2 Untersuchung der Probleme der Unterwasserrobotik** 5

2.1 Wettbewerbe . . . 5
2.1.1 Aufgaben des SAUC-E . . . 6
2.1.1.1 Navigation . . . 7
2.1.1.2 Inspektion . . . 7
2.1.1.3 Wandinspektion und Anomalie-Erkennung . . . 8
2.1.1.4 Erkundung und Flugschreibersuche . . . 9
2.1.1.5 Basisanforderungen des Wettbewerbs . . . 9
2.1.2 Aufgaben des euRathlon . . . 11
2.1.2.1 Langstreckennavigation . . . 12
2.1.2.2 Kartierung eines Unglücksgebietes . . . 13
2.1.2.3 Finden eines Lecks einer Pipeline . . . 13
2.1.2.4 Manipulation . . . 13
2.1.2.5 Das *Grand Finale* . . . 15
2.1.3 Demonstrationsmission . . . 15
2.1.4 Zusammenfassung . . . 17
2.2 Die Evaluationsplattformen . . . 18
2.2.1 Avalon . . . 19
2.2.1.1 Aktuatoren und Ansteuerung . . . 20
2.2.1.2 Kommunikation . . . 23
2.2.1.3 Spannungsversorgung . . . 26
2.2.1.4 Sensoren . . . 27
2.2.1.5 Basiskontrolle . . . 29

2.2.2 Dagon . . . 34
2.2.2.1 Stromversorgung . . . 34
2.2.2.2 Kamerasystem und optisches SLAM-Verfahren . . . 34
2.2.2.3 Thruster . . . 34
2.2.2.4 Steuerungskonzept & Kommunikation . . . 35
2.2.2.5 Navigation und Lokalisierung . . . 36
2.2.2.6 Sensoren . . . 37
2.3 Softwarekomponenten der beiden AUV-Plattformen . . . 38
2.3.1 Lageschätzung . . . 38
2.3.2 Lageregelungstreiber . . . 38
2.4 Vorstellung der Komponentennetzwerke . . . 44
2.4.1 Einleitung . . . 45
2.4.2 Syskit . . . 47
2.4.3 Syskit-Modelle . . . 47
2.4.4 Vorstellung der eingesetzten Modelle . . . 49
2.5 Erkenntnisse . . . 57
2.5.1 Hardware . . . 58
2.5.2 Software . . . 61
2.5.3 Komponentennetzwerke . . . 63
2.5.4 Erfahrungen während der Systemeinsätze . . . 64
2.6 Zusammenfassung und Herleitung der Kernprobleme . . . 66

**3 Constraintbasierte Planung von Komponentennetzwerken 69**
3.1 Erläuterung des gewählten Lösungswegs . . . 70
3.1.1 Detaillierung der Syskit-Modelle . . . 70
3.1.2 Änderungen der Modelle . . . 71
3.1.3 Umformung der Modelle in eine formalisierte Problemstellung . . . 75
3.1.3.1 Überführung der Modelle in PDDL . . . 75
3.1.4 Überführung der Modelle in formale Constraints . . . 95
3.1.4.1 Einleitung . . . 95
3.1.4.2 Einführung constraintbasierter Modelle . . . 96
3.1.4.3 Klassenlösung . . . 98
3.1.4.4 Instanzlösung . . . 118
3.1.4.5 Die Modelle zur Verhaltenssequenzierung . . . 128
3.1.5 Zusammenfassung der formalen Modelleigenschaften . . . 143
3.1.6 Ergebnisse . . . 144
3.1.6.1 Testmodelle . . . 144
3.2 Zusammenfassung der constraintbasierten Netzwerkbehandlung . . . 160

**4 Zusammenfassung der Arbeit 161**

5 Diskussion und Ausblick 163
5.1 Komponentennetzwerkplanung . . . . . . . . . . . . . . . . . . . . . . . . . . . . 163
5.2 Ausblick über zukünftige Arbeiten der Verhaltensplanung . . . . . . . . . . . 164

A Referenzen 167

B Unpublished Journal Article 173

# 1 Einleitung

Moderne Robotersysteme werden zunehmend komplexer. Die Forschung im Bereich der Robotik sieht sich immer mehr mit dem Problem der Integration von Algorithmen in neue Plattformen konfrontiert. Es gibt zahlreiche innovative Verfahren und Algorithmen, die die Robotersysteme mit neuen Fähigkeiten ausstatten. Diese Erweiterungen lassen sich sowohl auf der Hardware- als auch der Softwareebene erkennen. Jedoch findet diese Vielzahl an neuen Entwicklungen nur selten Anwendung auf den aktuellen Ziel-Plattformen. Dies bremst weitere Entwicklungen, da mehr Zeit für die Integration als für die eigentliche Forschung aufgewendet wird.

Mit einem Blick über die eigene Forschung hinaus oder durch die detailliertere Beschäftigung mit den eingesetzten Verfahren lässt sich erkennen, dass die Lösungen zumeist systemspezifisch sind oder sich nur mit großer Mühe auf andere oder gar größere komplexere Maßstäbe übertragen lassen. Das Ziel dieser Arbeit ist es daher zu identifizieren, wo die größten Herausforderungen für die Forschung liegen. Im Anschluss soll das markanteste Problem gelöst werden.

Die Modularisierung sowie die modellgetriebene Entwicklung sind dabei die größten Änderungen, die sich Abzeichen. Beide Trends haben verschiedene, teils gravierende Einflüsse auf die Art und Weise, wie Entwicklung stattfindet. Modularisierung erhöht die Wiederverwendbarkeit einzelner Module und führt zu kleineren ‚Verbunden'. Modellgetriebene Entwicklung versucht mögliche Fehler in Konzepten zu einem frühen Zeitpunkt zu erkennen. Konkret bedeuten beide Verfahren jedoch Mehrarbeit im Vergleich zu der klassischen monolithischen und reaktiven Systementwicklung. Die Komponenten lassen sich zwar oftmals wiederverwenden, aber die Konsistenz und Nebeneffekte der Selektion bzw. der Abhängigkeiten erfordern viel Zeit und Expertenwissen. Die modellgetriebene Softwareentwicklung verlangt von den Entwicklern von Algorithmen bereits im Voraus Informationen darüber, wie und unter welchen Bedingungen ihr Algorithmus operieren kann.

Auf der anderen Seite, der Hardwareentwicklung, stehen immer kleinere und hoch integrierte Sensoren und Aktuatoren zur Verfügung, die wiederum die Konstruktion immer komplexerer Systeme erlauben. Die Systeme bieten oftmals ausreichend Redundanz, um Probleme auf unterschiedliche Art und Weise lösen zu können.

## 1.1 Definition der Komponentennetzwerke

Komponentennetzwerke sind allgemein Verbunde von verschiedenen Softwarekomponenten. In dieser Arbeit wird das *Verhalten* eines Robotersystems als Zustand des Komponentennetzwerkes definiert (Bild: 1.1). Zum Zustand des Komponentennetzwerkes und somit zum Verhalten gehören:

- die aktiven Komponenten im Netzwerk,
- die Verbindungen zwischen den Komponenten
- sowie die Konfiguration der Komponenten.

Je nach zugrundeliegendem Konzept untergliedern sich die Komponentennetzwerke in mehr oder minder autarke Einheiten. Die größten Vertreter der Komponentennetzwerke in der Robotik-Community sind ROS und Rock. Beide Lösungen verfolgen dabei unterschiedliche Grundkonzepte, wären aber aufeinander abbildbar. Bei Rock findet die Steuerung der einzelnen Komponenten (in Rock-Sprache: Tasks) über eine externe Einheit, die ‚Supervision', statt. ROS hingegen verfolgte lange die Idee, dass Komponenten selbst Entscheidungen über ihren Zustand bzw. ihre Rolle in einem Gesamtsystem treffen und auch andere Komponenten beeinflussen.[1]

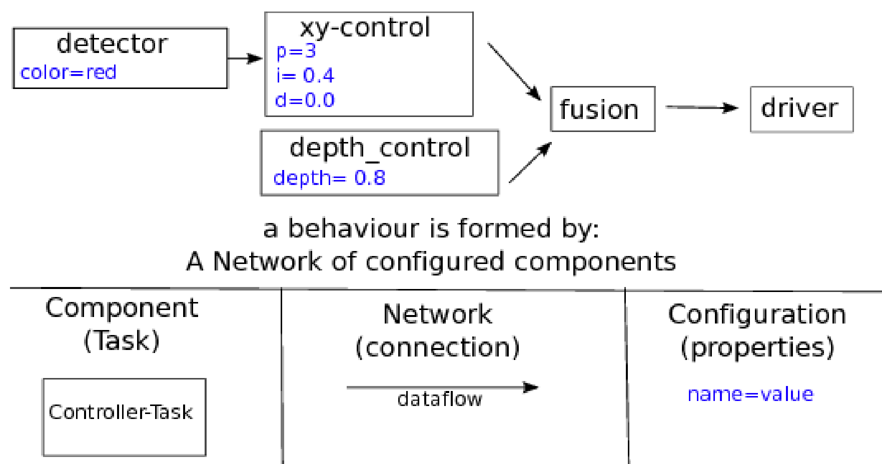



Figure 1.1: Übersicht über die Terminologie ‚Verhalten'

[1]Die *ROS-2.0*-Konzepte nähern sich jedoch an Rock an

## 1.2 Definition des Plan-Managers

Bei ROS müssen die Komponenten in sich einen natürlichen Gleichgewichtszustand (Homöostase) erreichen, um ein System stabil zu steuern. Bei Rock übernimmt dies eine höhere Schicht. Diese Supervision-Layer werden auch Plan-Manager genannt. Der Begriff Plan-Manager leitet sich jedoch nicht von der Sequenzierung von Aufgaben ab, sondern vom Planen der Komponenten. Der Fokus liegt also nicht auf dem Planen von Sequenzen, um Aktionen in der Welt auszuführen, sondern in der Planung von Komponentennetzwerken, um eine Aufgabe zu bewältigen. Plan-Manager wie Syskit bieten zwar Erweiterungen an, um auch sequentielle Aufgaben zu modellieren. Jedoch steht auch hier nicht die Ausführung von Aktionen in der Welt im Vordergrund. Unabhängig von der genauen Abgrenzung wird durch die Planung von Komponentennetzwerken implizit das Verhalten eines Systems geplant, die einzigen Unterschiede sind die Granularität sowie die Aktionskomplexität.

## 1.3 Abgrenzung zu anderen Arbeiten

Diese Arbeit grenzt sich von biologisch inspirierten (reaktiven) Ansätzen der Verhaltensplanung bzw. Verhaltensgenerierung wie der *subsumption architecture* [9] ab. Diese Methoden werden oftmals in den Bereich der *verhaltensbasierten Robotik* eingeordnet. Die verhaltensbasierte Robotik betrachtet ein Roboterverhalten als kontinuierliches Problem. Dabei sind dedizierte Unterzustände im Verhalten nicht (explizit) Teil der Planung. Auch findet eine explizite Planung, also eine deliberative Betrachtung, typischerweise nicht statt. Der gesamte Roboterzustand ist nach dieser Auffassung ein fester Verbund, der selbstständig in sich das Verhalten beeinflusst und je nach Systemzustand andere Aktionen durchführt.

Im Gegensatz dazu sieht diese Arbeit das Systemverhalten als Folge der Zustände von verschiedenen Komponentennetzwerken an. Ein Netzwerk determiniert das Verhalten des Robotersystems für einen definierten Zeitpunkt, dabei kann ein Netzwerk für einen Zeitpunkt den Konzepten der *verhaltensbasierten Robotik* folgen. Ein Netzwerk für einen Zeitpunkt beschreibt somit höherfrequente Vorgänge, die keine höheren kognitiven Fähigkeiten erfordern, die Sequenz über mehrere Komponentennetzwerke bildet im Gegensatz dazu komplexere Vorgänge ab, die i. d. R. zugleich niederfrequenter stattfinden. Bereits die Arbeit von [6] stellt diese konzeptuellen Unterschiede der beiden Ansätze heraus, wie in Bild 1.2 zu sehen ist.

In der Arbeit wird somit die Annahme getroffen, dass reaktive, verhaltensbasierte Ansätze durch einzelne Komponentennetzwerke realisiert werden können. Die Planung von Verhalten erfordert jedoch eine komplexere Rekonfiguration des Komponentennetzwerkes

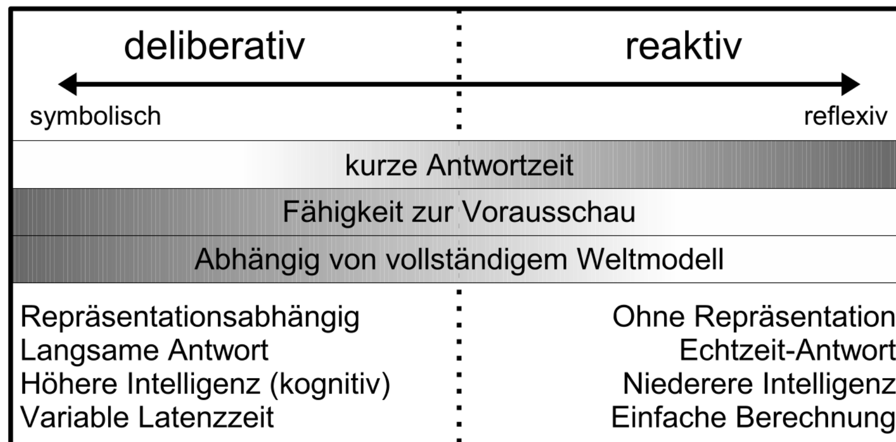



Figure 1.2: Verhaltensbasierte vs. klassische Roboterkontrolle [6] (deutsche Übersetzung [3])

und könnte somit mittels modellbasierter Verfahren, wie sie im Laufe der Arbeit entwickelt werden, besser gelöst werden. *Warum* etwas getan wird, obliegt somit dem Bereich der Komponentennetzwerkplanung, der deliberativen Ebene. *Wie* etwas getan wird, wird durch die Komponentennetzwerke selbst bestimmt.

Dabei ist die Grenze zwischen dem *Wie* und dem *Warum* fließend. Es sind Architekturen möglich, die auf der deliberativen Planungsebene Entscheidungen über Trajektorien treffen, um beispielsweise exakte Greifbewegungen durchführen zu können. Andererseits wäre diese Aufgabe auch über reaktive Lösungen innerhalb der Verhaltensnetzwerke zu bewältigen. Die genaue Abgrenzung obliegt dabei den Anforderungen oder Fähigkeiten sowie den Zielen, die ein System erreichen soll. Auch verschieben neue wissenschaftliche Arbeitsweisen, wie das *deep learning* [54], stetig die Grenzen.

# 2 Untersuchung der Probleme der Unterwasserrobotik

Dieses Kapitel widmet sich der Erstellung eines unter Realbedingungen einsetzbaren Systems, das als Evaluationsplattform dienen soll. Ziel ist es einerseits, ein System zu entwickeln, das aktiv in der Forschung eingesetzt werden kann, aber andererseits auch zu evaluieren, wo die Probleme bei der Behandlung von komplexen Systemen liegen.

Dazu wird das System *Avalon* entwickelt. Die Entwicklung beschränkt sich, im Gegensatz zu vielen anderen Arbeiten, nicht auf die Erstellung eines abstrakten virtuellen oder minimalistischen Systems. Vielmehr soll ein System entwickelt werden, das robust und vielfältig genutzt werden kann. Mittels dieses Systems werden die auftretenden Probleme detailliert untersucht, um die Herausforderungen und Schwachstellen moderner Robotikentwicklung zu identifizieren. Da diese Dissertation im Rahmen des Graduiertenkollegs *System-Design* stattfindet, nimmt die praktische und wissenschaftliche Erstellung eines Systems wie Avalon einen großen Teil dieser Abhandlung ein.

Im späteren Verlauf wird, um sowohl die Portabilität von Komponentennetzwerken zu zeigen als auch ein breiteres Spektrum an Problemen untersuchen zu können, als bereits vorhandene Plattform ein zweites System mit dem Namen *Dagon* hinzugezogen.

Ferner werden in diesem Kapitels Probleme präsentiert, die während der Umsetzung verschiedener Konzepte aufgetreten sind. Die Zusammenfassung am Ende des Kapitels unterteilt die Probleme in verschiedene Problemklassen. Die relevanten Probleme werden im Folgekapitel 3 ausführlich behandelt, dort werden auch potenzielle Lösungen für diese Probleme präsentiert.

## 2.1 Zielsetzung: Wettbewerbe

Die Zielstellung für den Hardwareentwurf und die Entwicklung der Softwarealgorithmen sollen die Wettbewerbe SAUC-E und euRathlon 2014 sein. Der SAUC-E ist der führende europäische Wettbewerb für Unterwasserrobotik, in dem die Systeme der Teilnehmer verschiedene Aufgaben wie Lokalisierung, Navigation und Inspektion bewältigen sollen. Der

euRathlon ist hingegen ein Kombinationswettbewerb aus Land-/Luft-/Unterwasserrobotik, in dem Kartographierung und Manipulation als weitere Fähigkeiten gezeigt werden sollen. Im Jahr 2014 fand jedoch nur der Unterwasserteil des Wettbewerbes statt.

### 2.1.1 Der SAUC-E Wettbewerb

Der Students Autonomous Underwater (Vehicle) Challenge - Europe (SAUC-E) ist ein seit 2006 jährlich stattfindender Wettbewerb der Unterwasserrobotik. Er wird seit 2010 in La Spezia (Italien) ausgetragen. Hier befindet sich ein Hafenbecken (Bild: 2.1), das durch das Centre for Maritime Research and Experimentation (CMRE) inkl. der benötigten Infrastruktur bereitgestellt wird.

Figure 2.1: Testbecken des CMRE, in dem der SAUC-E und euRathlon stattfanden.

Der SAUC-E 2014 bestand dabei aus verschiedenen Aufgaben. Die Hauptmission setzt sich aus den folgenden Aspekten zusammen:

- Navigation *Passing a Gate*
- Unterwasserstruktur-Inspektion
- Wandinspektion, Anomalie-Erkennung und Benachrichtigung eines Partnersystems
- Erkundung der Umgebung und Detektion eines Flugschreibers

Jede dieser Aufgaben soll verschiedene Fähigkeiten von autonomen, intelligenten Unterwassersystemen fordern und zeigen.

#### 2.1.1.1 Navigationsaufgabe

Die Basisfähigkeit der Navigation ist dabei eine schwierige Anforderung, die ein System absolvieren muss, um zu dem Wettbewerb bzw. zum Finallauf zugelassen zu werden. Die Anforderung dieser Aufgabe besteht dabei aus der folgenden Sequenz:

1. Starten am Randbereich des Beckens
2. Navigieren zum Zentrum des Beckens über eine Struktur
3. Passieren des *Gates* (Bild: 2.2) im inneren Bereich

Die beiden Pfosten des beleuchteten Tores stehen dabei in circa zwei Meter Abstand voneinander. Der gesamte Wettkampfbereich hat eine Große von ca. 120 x 50 Metern. Dies erschwert eine globale präzise Positionsschätzung, da zu diesem Zweck ein absoluter Referenzpunkt benötigt wird. Typische Sonare bieten jedoch unter realen Bedingungen nur eine Reichweite von ca. 30 Metern.

#### 2.1.1.2 Aufgabe der Inspektion einer Unterwasserstruktur

Die Unterwasserstruktur, die inspiziert werden soll, ist eine Anordnung von verschiedenen Fallrohren, wie sie von Dachdeckern genutzt werden (siehe Bild: 2.3). Ziel der Aufgabe ist es, eine beliebige, möglichst detaillierte Karte der Struktur zu erstellen. Dabei soll die Struktur von allen Seiten umfahren werden, um eine größtmögliche Abdeckung zu erlangen. Die minimale Punktzahl wird erlangt, wenn die Struktur erkannt wird und eine kurze Zeit über oder vor der Struktur die Position gehalten wird. Zusätzliche Punkte werden je nach Güte der Karte (bzw. Rekonstruktion) vergeben. Dabei ist es irrelevant, ob die Rekonstruktion und Routenplanung auf optischen oder akustischen Sensoren beruht. Die Herausforderung dieser Aufgabe ist einerseits die präzise Navigation um die Struktur herum, ohne sie zu berühren, und andererseits die Entwicklung eines Algorithmus, der eine detaillierte Rekonstruktion der Struktur ermöglicht.

Figure 2.2: Der beleuchtete Pylon eines Torpfeilers. (Quelle: http://sauc-europe.org/)

#### 2.1.1.3 Aufgabe der Wandinspektion und Anomalie-Erkennung

Die Aufgabe erfordert präzise Navigation in einem fest definierten Abstand von unter zwei Metern zur Wand. Das Problem ist, dass dieses der Grenzbereich für akustische Sonare ist, in dem sie operieren können. Auch muss die Inspektion um eine Ecke herum erfolgen. Erschwerend kommt hinzu, dass der Boden des Hafenbeckens uneben ist und die Wände somit kein klares Echo produzieren. Dies ist der Tatsache geschuldet, dass ein Scanning-Sonar im Gegensatz zu einem Laserscanner einen größeren Fußabdruck besitzt, der zu schwammigen bzw. verwaschenen Daten führt. Auch muss in diesem Teilabschnitt des Wettbewerbes eine Anomalie erkannt werden, die als Boje umgesetzt ist. Wenn die Boje identifiziert wurde, soll eine kurze Zeit vor ihr die Position gehalten werden, um ein Kooperationsfahrzeug zu benachrichtigen. Das Kooperationsfahrzeug kann dabei ein anderes Autonomous Underwater Vehicle (AUV) (beispielsweise eines gegnerischen Teams) oder ein Autonomous Surface Vehicle (ASV) sein. Das Kooperationssystem soll an die Zielposition fahren, um dann bei der Anomalie die Position zu halten.

Figure 2.3: Zielstruktur der Inspektionsaufgabe. (Quelle: http://www.eurathlon.eu/)

#### 2.1.1.4 Erkundung der Umgebung und Detektion eines Flugschreibers

Die letzte Aufgabe des Wettbewerbes besteht in der Kartographierung des gesamten Hafenbeckens bzw. des Wettkampfbereichs. Hierbei entfallen 33 % der Punkte auf das Erstellen einer Karte, wie beispielsweise eines Mosaiks, 33 % auf das Erkennen der Blackbox (irrelevant, ob mit Sonar oder optisch) und 33 % auf das Auftauchen innerhalb eines 3-m-Radius um die Blackbox. Die Blackbox ist dabei deutlich rot gefärbt und emittiert aktiv jede Sekunde einen 10 ms langen akustischen Puls von 15 kHz (*Ping*).

#### 2.1.1.5 Basisanforderungen des Wettbewerbs

Während der gesamten Mission muss die Position des Fahrzeugs in ein Logfile geschrieben werden. Das System muss somit in der Lage sein, sich selbst in dem gesamten Bereich zu lokalisieren. Das Fahrzeug darf zu keinem Zeitpunkt auftauchen oder mit der Basis kommunizieren. Dabei sind auch jegliche Arten von Antennen oder externen Systemen zur Positionsbestimmung wie Ultra Short Baseline (USBL)-,Long Baseline (LBL)-,Short Baseline (SBL)- oder Global positioning System (GPS)-Lösungen verboten. Das Gesamtgewicht des Fahrzeugs darf 90 kg Gewicht nicht überschreiten. Bereits ab 70 kg gibt es Strafpunkte. Ein geringeres Gewicht als 45 kg bringt hingegen Bonuspunkte. Die optische Sichtweite

Figure 2.4: Eine Boje, die auch als eindeutige Anomalie vor einer Wand dient. (Quelle: Team Avalon 2011)

beträgt zwischen ein und zwei Metern. Strömung im Hafenbecken ist näherungsweise nicht vorhanden, jedoch kann es zu Wellengang durch vorbeifahrende Schiffe kommen. Der Wettbewerb selbst läuft über eine Woche. Sämtliche Tage vor dem Finale dienen dazu als Training. Hier können vorhandene Algorithmen an das Szenario angepasst und eventuelle Fehler beseitigt werden. Die eigentliche Mission darf eine Zeitdauer von 60 Minuten nicht überschreiten, ansonsten würde der gesamte Lauf als Fehlschlag gewertet werden.

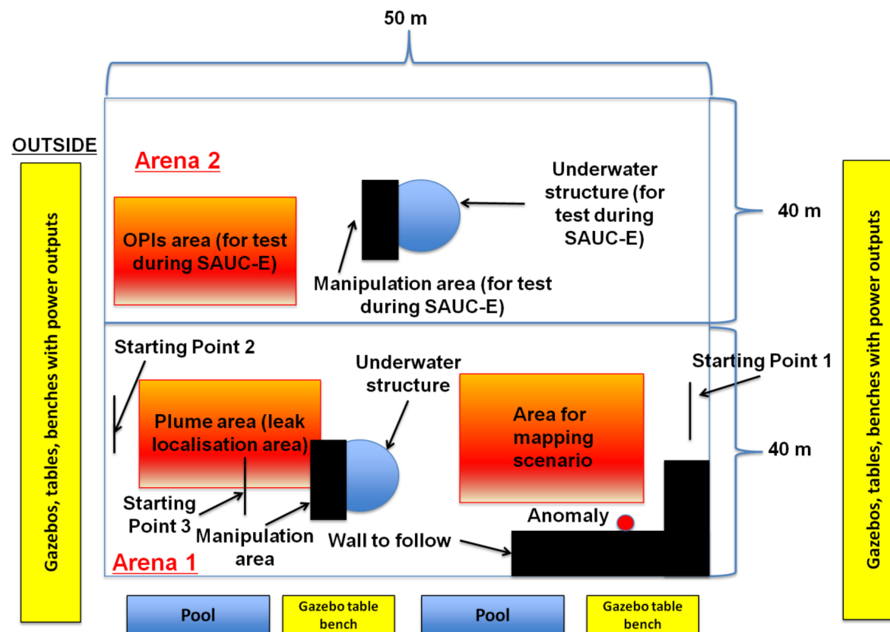



Figure 2.5: Wettkampfbereich der *euRathlon sea Challenge 2014*.
(Quelle: www.eurathlon.eu)

### 2.1.2 Der euRathlon Wettbewerb

Der euRathlon ist ein europäisch geförderter Wettbewerb, der u. a. durch die Programme n° 601205 (FP7/2007-2013) und n° 688441 (Horizon 2020) finanziert wird. Er umfasst verschiedene Szenarien und Disziplinen der Robotik. Sein Ziel ist es, die Entwicklung von Systemen zu fördern, die in Katastrophenszenarien in der Lage sind, entweder Folgen einzudämmen oder durch ihre Flexibilität schnell für Aufklärung zu sorgen. Die folgende Beschreibung bezieht sich auf die *euRathlon 2014 sea competition*, die verschiedene Teilbereiche der Unterwasserrobotik umfasste. Im Gegensatz zum SAUC-E sind die Aufgaben komplexer und schwieriger zu absolvieren. Das Ziel des gesamten euRathlon-Wettbewerbes ist eine Kombination aus allen Teilbereichen der Robotik, also der Land-, Wasser- und Flugrobotik, um die Systeme in einem Fukushima-ähnlichen Szenario in der *euRathlon 2015 Grand Challenge* gegeneinander antreten zu lassen und so den Stand der Technik weiter voranzutreiben.

Die *sea competition* setzt sich aus folgenden Unteraufgaben zusammen:

- Langstreckennavigation

- Kartierung eins Unglücksgebietes
- Finden eines Lecks einer Pipeline
- Manipulation an einer Unterwasserstruktur
- Kombination aller Aufgaben

#### 2.1.2.1 Aufgabe der Langstreckennavigation

Das Ziel dieser Aufgabe ist es, mit einem Fahrzeug möglichst schnell und präzise Zielwegpunkte sequenziell anzufahren. Dabei darf das AUV zwischendurch auftauchen, um zusätzliche Referenzposition mittels GPS zu erhalten. Jedoch werden beim Auftauchen Strafpunkte vergeben. Ein zweites Hilfssystem, ein ASV, darf an der Oberfläche bereitstehen oder mitfahren, um mit dem Unterwassersystem zu kommunizieren. Es ist somit erlaubt, dem AUV Positionen mitzuteilen. Eine direkte Kommunikation mit dem AUV von der Küste aus darf jedoch nicht erfolgen. Diese Aufgabe findet im Außenbereich des CMRE statt. Hier können Strömungen von bis zu einem Knoten auftreten. Die Schwierigkeit bei der Aufgabe liegt daher auch in der robusten Navigation unter Wasser. Strömungen, die präzise Schätzung der Orientierung sowie der gefahrenen Strecke sind dabei die größten Herausforderungen.

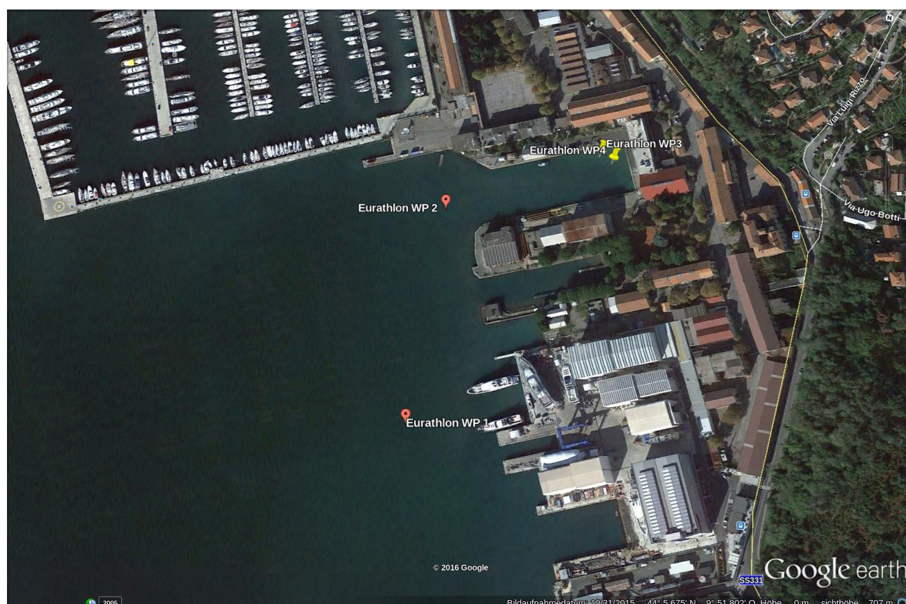



Figure 2.6: Langstreckennavigationsaufgabe des euRathlon. (Quelle: GoogleEarth)

#### 2.1.2.2 Kartierung eines Unglücksgebietes

Die Aufgabe beginnt, analog zu der SAUC-E Aufgabe *Wandinspektion und Anomalieerkennung* (siehe Kapitel 2.1.1.3), mit dem Erkennen einer Boje. Es kommt jedoch ein Licht oberhalb der Boje hinzu. Das AUV muss erkennen, ob das Licht oberhalb der Boje an- oder ausgeschaltet ist. Diese Information muss in den Logdaten, die das AUV aufzeichnet, annotiert werden. Im Anschluss muss der Bereich des Hafenbeckens inspiziert werden. Dazu kann eine Karte des Beckens erstellt werden. Über das gesamte Gelände sind verschiedene Bojen positioniert. Diese Bojen müssen vom AUV gefunden werden. Dabei muss über/vor jeder Boje gehalten werden, um die Erkennung zu verdeutlichen. Die Position aller Bojen muss im Log eingezeichnet werden. Zusätzliche Punkte können für die Erstellung einer 3D-Karte oder die komplette Rekonstruktion des Hafenbereiches gewährt werden.

#### 2.1.2.3 Finden eines Lecks einer Pipeline

Ähnlich wie in der vorherigen Aufgabe sind Bojen am Seeboden verteilt. Die Bojen sind mit Nummern beschriftet, die die *Stärke* eines Lecks symbolisieren sollen. Je höher die Nummern, umso näher ist das System an dem virtuellen Leck. Diese Aufgabe soll die dynamische Routenplanung und Objekterkennung von Systemen hervorheben. Nachdem das Leck gefunden wurde, muss die Leckstelle inspiziert werden. Das Leck entspricht der Rohrkonstruktion, die schon für den SAUC-E verwendet wurde. Hier soll eine mögliche genaue Karte/Rekonstruktion der Struktur bereitgestellt werden. Darüber hinaus befinden sich ein Ventil und ein herausziehbarer Stab an der Struktur. Diese beiden Objekte müssen zusätzlich im Detail erkannt werden.

#### 2.1.2.4 Manipulationsaufgabe an einer Unterwasserstruktur

Diese Aufgabe ist die einzige, die semiautonom oder auch ferngesteuert durchgeführt werden darf. Das AUV darf hierbei mit dem ASV verbunden sein, wobei das ASV als Relaisstation zur Basisstation dient. Das Ziel ist es, den Kontakt mit der Struktur für mindestens 30 Sekunden herzustellen, anschließend ein Ventil um 90°zu drehen, um dann einen Griff aus der Struktur herauszuziehen. Eine äußerst präzise Navigation sowie eine robuste zuverlässige Kommunikation mit allen Fahrzeugen ist von äußerster Bedeutung für diese Aufgabe.

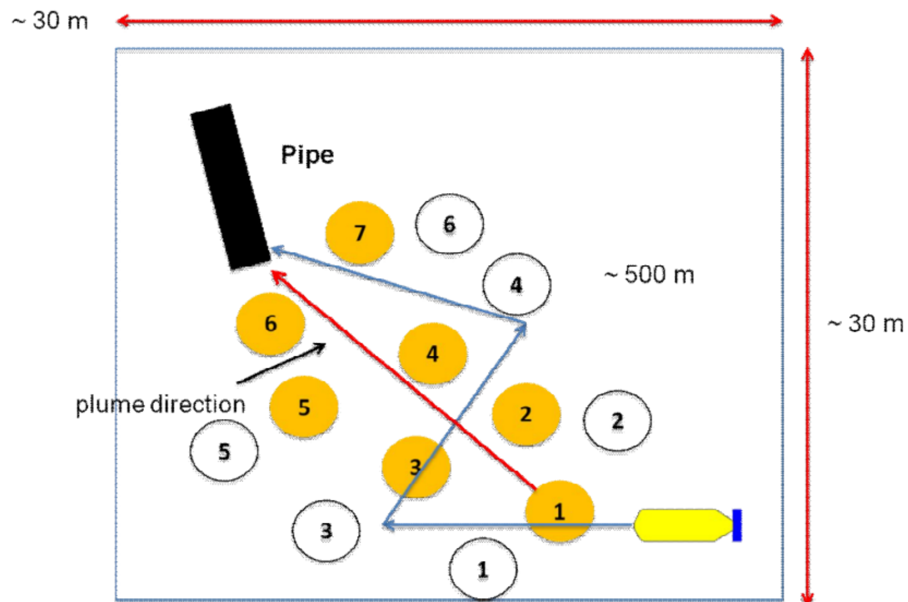



Figure 2.7: Beispiel der Navigation zur Leckerkennung beim euRathlon. (Quelle: www.eurathlon.eu)

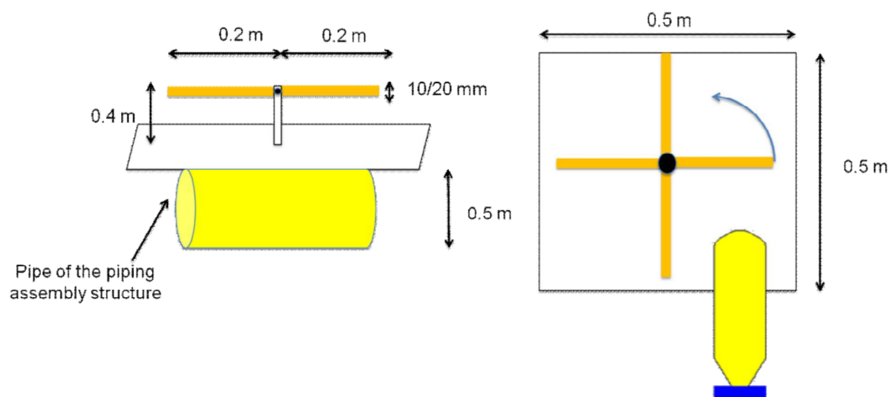
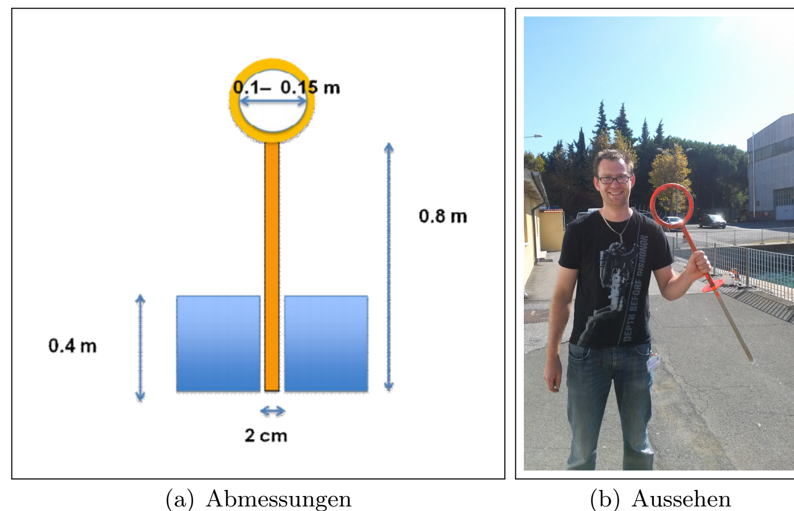



Figure 2.8: Das Ventil, das während der Aufgabe *Manipulation an einer Unterwasserstruktur* gedreht werden muss. (Quelle: www.eurathlon.eu)




(a) Abmessungen (b) Aussehen

Figure 2.9: Der Ring, der aus einer Struktur während der Aufgabe *Manipulation an einer Unterwasserstruktur* des euRathlon gezogen werden muss.
(Quelle: www.eurathlon.eu & Matthias Goldhoorn)

#### 2.1.2.5 *Grand Finale* - Kombination aller Aufgaben

Die letzte Aufgabe des euRathlon besteht aus dem Aneinanderketten sämtlicher vorher genannter Aufgaben abzüglich der Aufgabe *Finden eines Lecks einer Pipeline.* Die Aufgaben müssen hierbei autonom, ohne Auftauchen oder Interaktion von außen, aneinandergereiht werden, um die volle Punktzahl zu erhalten. Eine robuste und zuverlässige Missionsplanungssoftware ist hierfür unerlässlich. Die größte Schwierigkeit liegt darin, eine robuste Lokalisierung zu erhalten, sobald das System aus dem Außenbereich in den Innenbereich hineinfährt. Im Außenbereich ist außer mittels GPS keine globale Position bestimmbar. Im Innenbereich helfen die Wände eine globale Position zu bestimmen. Hierbei müssen die eingesetzten Filteralgorithmen mit eventuellen Sprüngen in der Lokalisierung robust umgehen können. Auch die Missionsplanung muss bei einem Fehlschlag einer Teilmission entweder die Aufgabe wiederholen oder zur nächsten Aufgabe übergehen können.

### 2.1.3 Demonstrationsmission unter Laborbedingungen

Neben den Wettbewerben SAUC-E und euRathlon soll das Verhalten der Systeme in einer Laborumgebung innerhalb der Räumlichkeiten des Deutsches Forschungszentrum für

Künstliche Intelligenz - Robotics Innovation Center (DFKI-RIC) Bremen erprobt werden. Hier steht ein Testbecken zur Verfügung, in dem Unterwassersysteme ausgiebig in einer definierten Umgebung evaluiert werden können. Das Testbecken, das in Bild 2.10 zu sehen ist, hat die Maße von 23 Metern Länge x 19 Metern Breite x 8 Metern Tiefe und somit ein Fassungsvermögen von 3496 $m^3$.

Im Gegensatz zu den Wettbewerben sollen in dieser Mission Langzeitaspekte der Autonomie evaluiert werden. In den Wettbewerben werden die Missionen so definiert, dass bei einem Fehler oder unerwarteten Zustand das System auftauchen oder die aktuelle Aufgabe überspringen soll, um maximale Kontrolle über das Fahrzeug zu behalten. Im Gegensatz dazu soll in der künstlichen Umgebung dieser Mission das System möglichst lange und oft in einer Schleife die Aufgaben abarbeiten, um die Robustheit der Detektoren sowie der gesamten Missionsplanung zu untersuchen.

Figure 2.10: Testbecken des DFKI-RIC

### Szenario der Demonstrationsmission

Um eine vergleichbare Mission für das Avalon-System zu erstellen, werden verschiedene Bestandteile des euRathlon und des SAUC-Es ausgewählt. Die Mission soll folgende Auf-

gaben beinhalten:

- Wandverfolgung inkl. Passieren von Ecken der Testumgebung
- Pipelineverfolgung
- Lokalisierung
- Anfahren globaler Koordinaten
- Ausweichverhalten im Fehlerfall einzelner Aktionen zur Erhöhung der Robustheit

Die Gesamtmission inkl. der im Becken installierten Testumgebung ist in Abbildung 2.11 skizziert.

Figure 2.11: Demonstrationsmission des Avalon-Systems zur Validierung von langfristigen Missionen sowie der Robustheit gegen Fehler.

### 2.1.4 Zusammenfassung der Anforderungen aller Szenarien

Beide Wettbewerbe, sowohl der SAUC-E als auch die *euRathlon Sea Competition*, stellen große Herausforderungen an die eingesetzten Systeme. Diese Anforderungen umfassen sowohl die Hard- als auch die Software. Die Systeme werden in echten, nicht in Laborbedingungen eingesetzt und müssen diesen Umgebungen standhalten. Darüber hinaus ist nur eine eingeschränkte Kommunikation möglich. Es ist von hoher Relevanz, stets eine Kontrollmöglichkeit über die Systeme zu besitzen. Eventuelle Softwarefehler dürfen nicht

zur Folge haben, dass die Systeme tot oder unkontrolliert herumtreiben und nicht mehr steuerbar sind. Ein physikalischer Zugriff auf die Systeme ist nur durch Hilfsschlauchboote mit Mehraufwand möglich. Unter Anbetracht des engen Zeitplans wäre dies ein großer Zeitverlust und erweckt zugleich den Eindruck eines unprofessionellen Auftritts.

Neben den Anforderungen an Detektoren sowie die Systemregelung ist es von großer Bedeutung, dass die Systeme intelligent Entscheidungen treffen können, was den Missionsablauf anbelangt. Diese Anforderungen gelten sowohl für die Wettbewerbe SAUC-E und euRathlon als insbesondere auch für das Demonstrationsszenario zur Realisierung einer robusten Architektur. Eventuelle Missionsanpassungen oder Änderungen an Abläufen müssen dynamisch vorgenommen werden können, falls Teilmissionen fehlschlagen. Dabei ist es nicht relevant, ob der Fehlschlag durch Fehler in Detektions-, Regelalgorithmen oder der Missionsplanung hervorgerufen wurde. Die Systeme müssen in der Lage sein, von einem Fehlerzustand in einen gesicherten Zustand überzugehen. Im schlimmsten Fall bedeutet das für Autonomous Underwater Vehicles (AUVs) das Auftauchen und Abbrechen der Mission, um die Hardwareplattform nicht zu gefährden.

Für den Wettbewerb SAUC-E wurde entschieden, dass nur das Avalon-System zum Einsatz kommt. Dagon ist hauptsächlich wegen seines Gewichts nicht für den Wettbewerb nutzbar, da es über dem zugelassenen Maximalgewicht liegt. Des Weiteren bietet Avalon Vorteile in der Navigation auf kleinem Raum, da das System durch eine andere Anordnung der Thruster in der Lage ist, besser Seitwärtsbewegungen auszuführen. Weitere wesentliche Entscheidungen sind, dass auf eine aktive Erkennung des Tores verzichtet wird. Anhand der gestellten Anforderungen wird es vermutlich nur mit großem Aufwand möglich sein, das Tor aktiv zu erkennen. Stattdessen wird geplant das Tor durch eine präzise Navigation zu durchfahren. Die akustische Detektion der Blackbox wird ebenfalls aus der Mission entfernt, stattdessen soll mittels eines Rasenmähermusters versucht werden, die Box optisch zu erkennen.

Innerhalb des euRathlon-Wettbewerbes wird ebenso Avalon für die meisten Aufgaben verwendet, da sich das System ähnlich wie beim SAUC-E in einer engen Umgebung bewegen soll. Lediglich für die Langstreckennavigationsaufgabe im Außenbereich soll Dagon genutzt werden. Dagon besitzt durch das *DVL* große Vorteile in der Positionsbestimmung, sofern keine Landmarken, wie das Hafenbecken selbst, vorhanden sind.

## 2.2 Vorstellung der Evaluationsplattformen

Dieses Kapitel stellt die Systeme Avalon (Abschnitt: 2.2.1) und Dagon (Abschnitt: 2.2.2) vor. Dabei wird insbesondere das System Avalon präsentiert, das im Rahmen dieser Arbeit entstanden ist. Die zugrundeliegenden relevanten Designentscheidungen und -konzepte

werden dabei in den Vordergrund gestellt. Dagon bestand als bereits abgeschlossenes System und dient als weitere Evaluationsplattform, um die Portierbarkeit und Wiederverwendbarkeit von Komponentennetzwerken zu evaluieren.

### 2.2.1 Das Avalon-System

Das System Autonomous Vehicle for Aquatic Learning, Operation and Navigation (Avalon) dient als Basis für verschiedene Aspekte:

1. Evaluationsplattform für Komponentennetzwerke,
2. Systemmodell für Plan-Manager, der die Komponentennetzwerke verwaltet,
3. Plattform zur Evaluation des Standes der Technik für autonome Systeme im Allgemeinen (Software und Hardware).

Im Folgenden wird zunächst auf die Hardware des Systems eingegangen. Die Software des Systems Avalon wird unabhängig von der Hardwareplattform im Kapitel 2.3 behandelt.

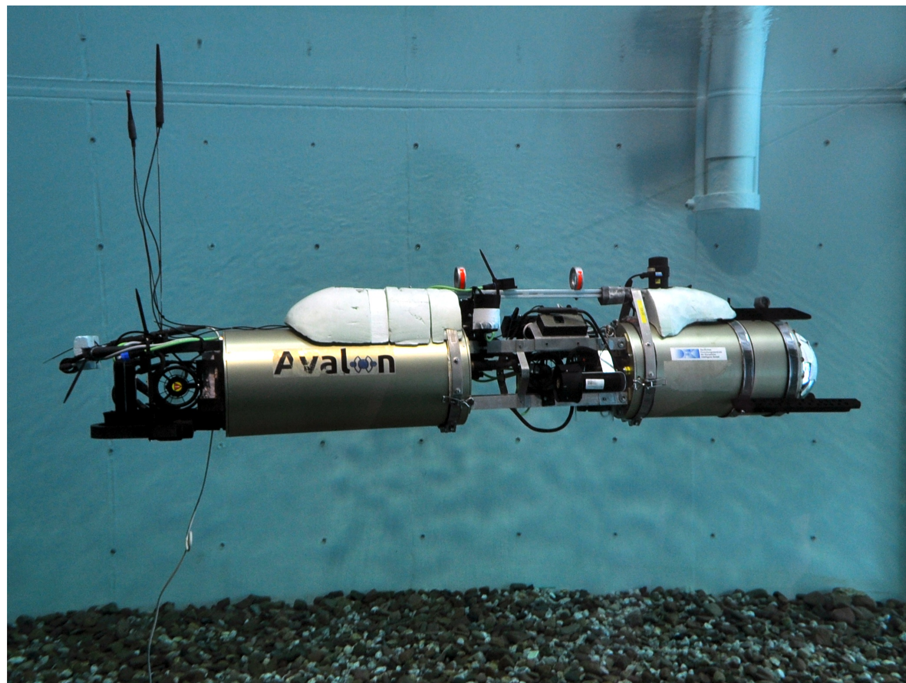



Figure 2.12: Übersicht über das Avalon-System

Wie im vorherigen Kapitel 2.1 zu den Wettbewerben beschrieben, ist es von großer Bedeutung, dass die Hardware eines AUVs möglichst robust gehalten wird. Obwohl es sich um eine Forschungsplattform im Bereich der Robotik handelt, muss der Fokus mehr auf der Zuverlässigkeit und Robustheit als auf der Erforschung neuer Hardwareeigenschaften liegen. Somit hat die Auswahl der Hardware großen Einfluss auf diese Arbeit. Die Auswahl muss dementsprechend mit großer Sorgfalt durchgeführt werden, um primär Aspekte der Software untersuchen zu können.

Die Grenze zwischen Hard- und Software bilden in der Regel die Treiber, Aktuatoren, Sensoren sowie diverse Mikrocontroller, die im Robotersystem verteilt werden. Um das Ziel der Robotik zu erreichen, *langfristig autonom operierende Systeme* zu erstellen, müssen alle diese Komponenten robust miteinander arbeiten und auch Informationen über Fehler oder unerwartete Zustände mitteilen, sodass auf höheren Ebenen eine intelligente Entscheidung für das Gesamtsystem getroffen werden kann.

#### 2.2.1.1 Die Aktuatoren und Ansteuerung des Avalons

Die Aktuatoren versorgen das System mit dem nötigen Schub für translatorische oder rotatorische Kräfte. Diese Kräfte wirken der Trägheit sowie der Dämpfung durch den Wasserwiderstand entgegen. Die Wahl geeigneter Aktuatoren hat direkten Einfluss auf die Geschwindigkeit und Agilität des Gesamtsystems. Es existieren auf dem Markt sowohl bürstenlose als auch bürstengetriebene Antriebe. Für Avalon wurde ein klassischer bürstengetriebener Antrieb gewählt. Die Wahl fiel hierbei auf die lange am Markt befindlichen Antriebe der Firma SeaBotix vom Typ BT-150 (wie auf Abbildung 2.13 zu sehen). Diese Wahl liegt darin begründet, dass keine komplizierte Kommutierung der einzelnen Motorpole erfolgen muss. Es kann ein Pulseweitenmodulation (PWM)-Signal auf die Motoren gegeben werden, das näherungsweise der ausgeübten Kraft entspricht. Die emittierte eingesetzte Leistung kann über den Stromfluss sowie die aktuell vorliegende Spannung bestimmt werden. Der Vorteil bürstenloser Motoren hingegen läge in einem geringeren Verschleiß sowie, je nach Elektronik, präziseren Informationen über die aktuelle Rotationsgeschwindigkeit. Für das Avalon-System wird jedoch angenommen, dass eine ausreichende Abbildung zwischen eingesetzter Motorkraft und ausgeübter Kraft auf die Umgebung vorhanden ist. Bürstenlose Motoren sind zudem neue Entwicklungen, über die bisher keine Erfahrungen bezüglich der Robustheit vorliegen.

Für die Ansteuerung des Systems kommen sogenannte H-Brücken, oder Vierquadrantensteller, zum Einsatz. Diese Schaltungsform ermöglicht eine Umpolung der Motoren, sodass eine Rückwärtsfahrt eingeleitet werden kann. Zusätzlich befinden sich Stromsensoren in der Schalung, die über den Hall-Effekt, also das magnetisch induzierte Feld von Strömen in einem Leiter, den Stromfluss bestimmen können. Die Motorleistung wird über

Figure 2.13: Antriebe des Systems Avalon vom Typ SeaBotix BT-150

PWM festgelegt. Je länger die *ein* Phase des Signals ist, umso größer ist die eingesetzte Leistung am Motor. Die Ansteuerung der Vierquadrantensteller erfolgt mittels eines STM32-Mikrocontrollers. Eine Darstellung der Elektronik ist in Abbildung 2.14 zu sehen. Die H-Brücke wurde im DFKI-RIC unter dem Namen *HBridge V2.0* für das *iMoby*-Projekt entwickelt.

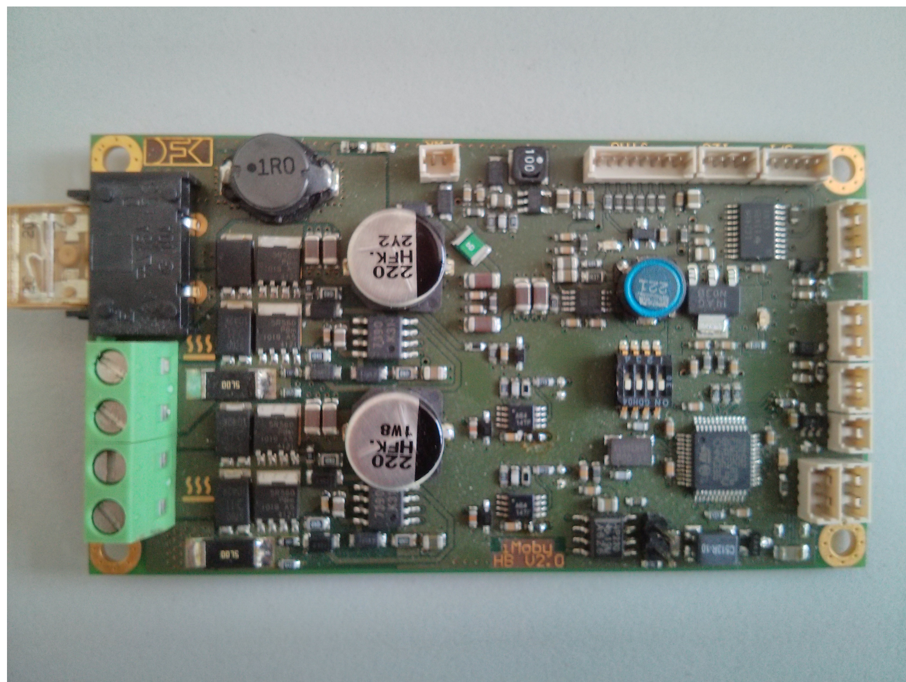



Figure 2.14: Bild der H-Brücken-Elektronik zur Ansteuerung der Antriebe. (hier V2.0)

Für die Autonomie bietet die eingesetzte Schaltung drei wesentliche Faktoren:

- Kommunikation über ein Mehrpunktbussystem (CAN)
  Die Kommunikation ermöglicht das Steuern und Überwachen der Motoren von mehreren Knoten im System aus. Das CAN-Protokoll bietet eine Priorisierung der Nachrichten. Dadurch ist es möglich, verschiedene Ebenen der Kontrolle zu definieren. Mehr dazu wird in Abschnitt 2.2.1.5 beschrieben.

- Notabschaltung bei zu großen Temperaturen oder Leistungen
  Da das Avalon-System unter Nicht-Laborbedingungen einsetzt wird, kann nicht sichergestellt werden, dass eventuelle Verstopfungen die Motoren blockieren oder Ablagerungen die Drehung erschweren. Im Fehlerfall werden die Motoren direkt, ohne Softwareinteraktion seines des Kontroll-PCs, abgeschaltet und es wird eine Meldung an die höheren Verhaltensschichten versandt.

- Rückmeldung des Status inkl. Leistung
  Diese Informationen ermöglichen die Schätzung der ausgeübten Energie. Auch werden Fehler wie Überlast oder Überhitzungen übermittelt. Die Informationen lassen somit auch Rückschlüsse zu, ob das System innerhalb der erwarteten Parameter arbeitet. All diese Zustände werden über das oben genannte CAN-Protokoll an weitere Knoten wie beispielsweise den Kontroll-PC oder die Basisstation versandt.

#### 2.2.1.2 Das Kommunikationssystem von Avalon

Ein wesentliches Merkmal sind die vorhandenen Kommunikationskanäle für AUVs. Gerade AUVs haben durch die große Dichte und die Absorbtionseigenschaften von Wasser nur eingeschränkte Kommunikationsmöglichkeiten mit Basis oder Kontrollstationen. Diese starke Einschränkung erfordert zugleich robuste interne Kommunikationskanäle zwischen einzelnen Komponenten. Somit teilt sich die Kommunikation in interne und externe Kommunikationskanäle auf. Die Vor-/Nachteile verschiedener Konzepte dabei werden im Folgenden erläutert.

**Die verwendeten Bussysteme in Avalon** Es gibt eine Vielzahl an Bussystemen, die für die Kommunikation zwischen Komponenten eingesetzt werden. Ein bedeutender Vertreter sind dabei die seriellen RS232-P2P- oder RS485-Bussysteme. Sie zeichnen sich durch eine hohe Robustheit, einfache Handhabung sowie ihre geringe Störanfälligkeit aus. Das Signal wird typischerweise als ±12-V-Signal übertragen. Dazu kommen je nach Version und Protokoll Differentialpegel und Checksummen auf Hardwareebene hinzu. In der Praxis weisen diese Kommunikationskanäle jedoch eine eingeschränkte Übertragungskapazität auf. Auch bietet der RS485-Bus keine Priorisierung von Nachrichten. RS232 ist dagegen kein echtes

Bussystem, sondern dient nur der Verbindung zweier einzelner Komponenten. Neuere Bussysteme wie SpaceWire wurden hauptsächlich für robuste Weltraumanwendungen mit starken elektromagnetischen Störfeldern entworfen. Diese spezialisierten Bussysteme verwenden dabei spezielle Hardware zum Ein- und Auslesen der Signale. Im Gegensatz dazu existiert für den weit verbreiteten CAN-Bus eine große Auswahl an Hardwarekomponenten. Der CAN-Bus bietet, mit einer Übertragungsrate von bis zu 1 Mbit, eine adäquate Lösung zwischen seriellen Schnittstellen (115.200 kBit/s) und SpaceWire (bis 200 MBit). Er besitzt eine Senderechtspriorisierung anhand der Nachrichten-IDs und ist im Automobilsektor seit Langem in Verwendung. Dies ist ein weiteres Indiz für seine Zuverlässigkeit.

Vielfältig im Einsatz sind ebenfalls Ethernetlösungen. Sie unterstützen große Kabellängen gepaart mit einer hohen und robusten Übertragungsleistung im Hochfrequenzbereich mit Bandbreiten von bis zu 1 GBit/s. Das Ethernet eignet sich somit hauptsächlich für die schnelle Kommunikation zwischen höheren Systemkomponenten. Das Ethernetprotokoll ist relativ komplex und besitzt einen großen Overhead, wodurch es nur eingeschränkt für eingebettete Systeme wie Mikrocontroller in Betracht kommt.

In Avalon kommt daher eine Kombination aus verschiedenen Schnittstellen zum Einsatz. Proprietäre Geräte wie Sonare verwenden das RS232-Protokoll. Da diese Schnittstelle durch den Entwickler der Komponenten definiert wird, ist sie eine nicht zu ändernde Eigenschaft. In Avalon kommen andererseits zwei PC-Systeme zum Einsatz, da ein hochfrequenter komplexer Datenaustausch zwischen ihnen erfolgt, wird für diesen Zweck Ethernet in der 1-GBit/s-Variante genutzt. An verschiedenen Netzwerkverteilern im System sind am Ethernetbus weiterhin die Kameras und ein faseroptischer Konverter angebunden.

Die Motorelektroniken basieren hingegen auf dem CAN-Protokoll. Dies ermöglicht eine Ansteuerung der Motoren von sämtlichen Geräten aus, die sich am CAN-Bus befinden. Diese Eigenschaft wird von einer Notfallsteuerung verwendet, die im Abschnitt 2.2.1.5 beschrieben wird.

#### Avalons externe Kommunikationskanäle

Die Kommunikation ohne physikalische Verbindung ist unter Wasser nur eingeschränkt möglich. Durch das große Absorptionsspektrum von Wasser eignen sich nur Geräte, die im Akustikbereich operieren. Diese Unterwassermodems haben eine große Reichweite, bieten jedoch nur geringe Übertragungskapazitäten von ca. 100 bit/s unter idealen Bedingungen. Somit eignen sich diese Systeme nur für rudimentäre Kommunikation, zumeist Notfallkommandos.

Um das System insbesondere für die Wettbewerbe steuerbar und konfigurierbar zu halten, wurde Avalon mit einem WiFi-Modul ausgestattet. Dieses Modul wurde mittels Epoxidharzen wasserdicht vergossen und am System befestigt. Es erlaubt zusammen mit einem

seriellen Amber-Wireless-867-MHz-Kanal die Kommunikation mit dem Fahrzeug, sobald dieses aufgetaucht ist.

Um an der Wasseroberfläche die globale Position bestimmen zu können, wurde desweiteren ein GPS-Modul wasserdicht vergossen, dieses Modul ist in Bild 2.15 zu sehen. Es ermöglicht die Positionsbestimmung des Systems an der Wasseroberfläche und kann somit für eventuelle Positionskorrekturen verwendet werden.

Figure 2.15: Wasserdicht vergossenes GPS-Modul.

Zur Kommunikation mit der Kontrollstation wird das Ethernetprotokoll genutzt. Hierbei stehen jedoch drei mögliche Kommunikationskanäle zur Verfügung: das bereits erwähnte WiFi-Funkmodul, ein Glasfaser- und ein Kupferkabel. Jeder dieser drei Kommunikationskanäle hat verschiedene Vor- und Nachteile. Das Kupferkabel ist auf ca. 50 Meter Länge begrenzt, da dies die maximale Kabellänge ist, mit der eine Übertragung stabil funktioniert. Die Funkkommunikation hat eine stärker eingeschränkte Bandbreite und arbeitet nur über Wasser. Sie funktioniert jedoch ohne physikalische Verbindung zur Kontrollstation. Ein Glasfaserkabel ist relativ empfindlich gegen auftretende physikalische Kräfte, die entstehen, falls die Faser sich verfangen sollte. Gerade im Feldeinsatz ist dieses oftmals schwer zu vermeiden.

#### 2.2.1.3 Die Spannungsversorgung von Avalon

AUVs müssen in der Lage sein, über längere Zeiträume autonom zu operieren. Eine Anforderung an das Avalon-System ist daher, dass auch längere Missionszeiten möglich sein sollten, um komplexere Aufgaben von Seiten der Hard- und Software durchführen zu können. Da es sich jedoch bei Avalon um ein Forschungssystem handelt, müssen verschiedene Sicherheitsaspekte mit in Betracht gezogen werden. Relevant ist dazu die Richtlinie zu Schutzkleinspannungen, um eventuelle elektrische Schläge zu verhindern bzw. die Spannungen so zu wählen, dass keine Schäden entstehen. Die Richtlinie [47] empfiehlt daher für den Anwendungszweck, unter den ein AUV fällt, Kleinspannungen von unter 50 V zu verwenden. Des Weiteren muss die Gefahr, die durch eine eventuelle unsachgemäße Behandlung der Akkumulatoren entstehen kann, in Betracht gezogen werden. Ein Überladen oder eine Tiefentladung kann im schlimmsten Fall je nach verwendeten Batterietyps zu einem Brand oder der Freisetzung von giftigen Gasen führen. Nach einer Untersuchung von [7] erscheinen aus chemischer Sicht die LiFePo-Batterien als die sichersten. Auch kommt es bei LiFePo-Batterien bei starker Entladung zu keiner endothermischen Reaktion. LiFePo-Batterien sind in der Praxis ein gute Alternative zu den LiIon-Batterien. Sie besitzen im Mittel mit 80-140 Wh/kg eine nur etwas geringere Energiedichte als die gefährlicheren LiIonO2-Batterien mit 120-210 Wh/kg.

Die verwendeten Akkuzellen besitzen keine Ladeelektronik. Durch die Anreihung der Zellen zu einem seriellen Verbund kann es dazu kommen, dass die einzelnen Zellen unterschiedliche Kapazitäten und Innenwiderstände aufweisen. Bei der Benutzung entladen sich so manchen Zellen schneller als andere. Dies führt zu einer unausgeglichenen Ladungsverteilung im Batteriepack und somit zu einer nicht idealen Nutzung. Es kann vorkommen, dass eine Batterie schon die Entladeschlussspannung von 2.5 V erreicht hat, während andere Zellen noch ausreichend gefüllt sind. Um diesem entgegenzutreten, müssen die Zellen untereinander ausgeglichen werden. Für Avalon wird ein externes Balancingsystem verwendet, das am DFKI entwickelt wurde. Dieses System gleicht während des Ladevorgangs die Zellspannungen aus, indem es über Ladungspumpen die Spannungen umverteilt. Das eigentliche Laden wird mittels eines Standard-Modellbauladegerätes vorgenommen, das in diesem Anwendungsfall lediglich als Konstantspannung und Konstantstromquelle dient. Dabei werden die Akkus mit bis zu 10 A bei einer Maximalspannung von 28,8 V (3,6 V je Zelle) geladen. Bei diesem Ladestrom ist das Balancingsystem in der Lage, entstehende Überspannungen zwischen den Zellen ausreichend schnell zu verteilen, sodass kein Überladen einzelner Zellen erfolgt.

Zum Schutz der Akkus und der Verkabelung wurde das Avalonsystem mit einer 20A-Sicherung abgesichert, die sich während der Benutzung trotz Unterdimensionierung als zuverlässig herausgestellt hat. Zum Schutz der einzelnen Zellen wird ein Solid-State-Relais

eingesetzt, das bei Unterspannung einer Zelle den gesamten Akku vom System trennt. Die einzelnen Zellspannungen werden mittels acht AD-Wandlern differenziell ausgelesen und ihre Werte über eine serielle Steuerleitung an den PC gesendet. Somit würde bei Unterspannungen das System automatisch abgeschaltet. Die Statusinformationen über den Zellzustand sind über den PC einsehbar. Auf eine Stromüberwachung wurde verzichtet, da die Akkupacks höhere Ströme erlauben als die, die im Avalon-System auftreten. Zwar ließe eine Strommessung genauere Rückschlüsse über die verbrauchte Energie des Systems zu, eine Schätzung über die Betriebsdauer in Kombination mit den Zellspannungen hat sich jedoch als ausreichend für ein Forschungssystem herausgestellt.

#### 2.2.1.4 Eingesetzte Sensorik im Avalon-System

Die Sensoren für das Avalon-System müssen kleine, leichte Geräte sein, um das Gesamtgewicht nicht zu überschreiten, das für den SAUC-E Wettbewerb vorgegeben wurde. Aus der Erfahrung der Arbeitsgruppe heraus wurde eine ‚MTi XSens IMU‘ als Basisgerät für die Orientierungsschätzung gewählt. Da wie in den Missionsregeln des SAUC-E- und euRathlon-Wettbewerbes vermerkt mit größeren magnetischen Störungen im Wettkampfbereich gerechnet werden muss, ist eine Unterstützung durch einen einachsigen faseroptischen Kreisel erforderlich. Die Sensorfusion übernimmt ein Uncentered Kalman Filter (UKF)-Verfahren, das in [28] beschrieben wird.

Für die Lokalisation innerhalb der Wassersäule kommen verschiedene Sensoren zum Einsatz. Dazu gehören ein Altimeter, das den Abstand zum Boden misst, und ein Drucksensor, der die Tiefe des Systems bestimmen kann. Bei dem Altimeter handelt es sich um den ‚Tritech Echosounder‘. Mit ihm ist es möglich, bis zu einer Nähe von ca. 30 cm über Grund den Abstand zu bestimmen. Für die Bestimmung der Tiefe kommt ein industrieller Drucksensor der Firma Sensortechics vom Typ CTE8016GY7 zum Einsatz. Er wird mittels einer ebenfalls über CAN angebundenen Elektronik ausgewertet. Die Elektronik bietet eine maximale Auflösung von 0.5 mm bei einer Aktualisierungrate von bis zu 50 KHz. Da die benötigte Aktualisierungsrate nur bei 10 Hz liegt, werden beide Sensoren über Tiefpassfilter zusammengefasst, um so die Tiefe des Bodens bestimmen zu können. Die Wahl der analogen industriellen Druckmesser liegt darin begründet, dass alternative Produkte speziell für den Robotik-/Forschungsbereich deutlich kostspieliger sind und langsamere Aktualisierungsraten liefern. Für das Avalon-System ist die exakte Tiefe weniger relevant, da keine Untersuchungen des Meeres durchgeführt werden sollen. Eine genaue Positions-/Lagereglung wird hier also gegenüber einer präziseren absoluten Position bevorzugt.

Um eine Positionsbestimmung und Kartographierung der näheren Umgebung zu ermöglichen, kommt das Micron-DST-Scanning-Sonar der Firma Tritech zum Einsatz. Dieses Gerät hat eine außerordentlich kleine Bauform. Das Micron liefert dabei einen 360°-

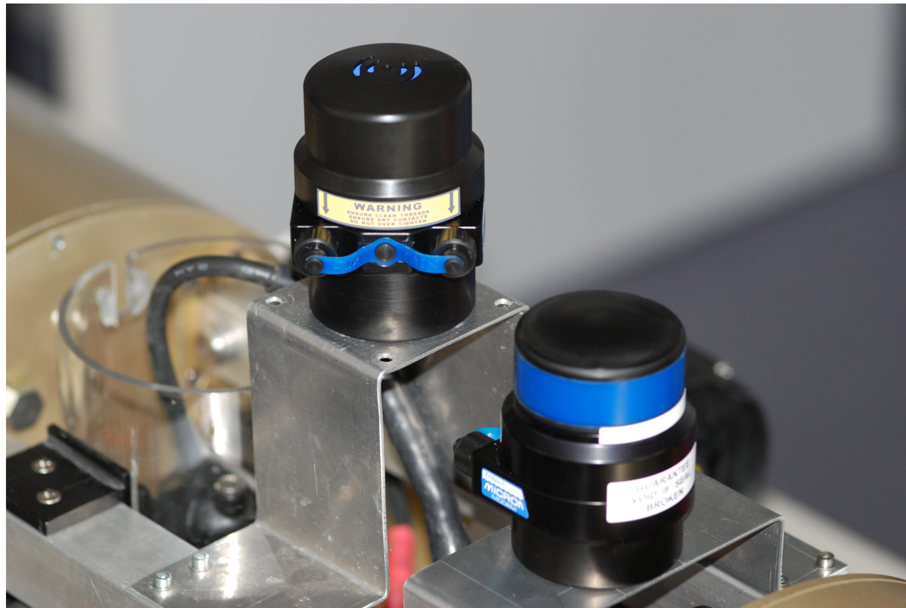



Figure 2.16: Tritech-Micron-DST-Modem und Scanning-Sonar

Rundumscan mit einer Winkelauflösung von bis zu 0.5°und einer maximalen Distanz von 30-50 Metern (je nach Bedingungen). Die horizontale Ebene besitzt jedoch einen 30°-Öffnungswinkel. Dadurch sind Signale nicht klar der Wasseroberfläche, dem Boden oder Objekten in der Wassersäule zuzuordnen. Das Bild 2.17 zeigt eine geschwungene Pipeline und links neben ihr (mittig im Bild) das Fahrzeug selbst. Da der Boden in den Ecken (speziell die linke untere Ecke) ansteigt, resultiert dies in einem stärkeren Echo (erkennbar am helleren Farbton) als beispielsweise in der oberen Bildhälfte, in der das Hafenbecken tiefer wird und ins Meer mündet. Laserscanner funktionieren im Gegensatz zu akustischen Sensoren unter Wasser nur eingeschränkt. Sie benötigen klares Wasser, das während der Wettbewerbe nicht zu erwarten ist.

Um optische Ziele identifizieren zu können, wurden für Avalon zwei Kameras der Firma Prosilica vom Typ GC650C und GC2450C ausgewählt. Die erste Kamera wurde nach unten und die zweite nach oben gerichtet. Auf diese Weise liefern die Kameras die benötigten Informationen, um die optischen Ziele zu erkennen. Wie in Abbildung 2.18 zu sehen ist, wurden die Kameras in dem Kuppeldom des Fahrzeugs verbaut, um eine möglichst gute Sicht auf die Umgebung zu erlangen.

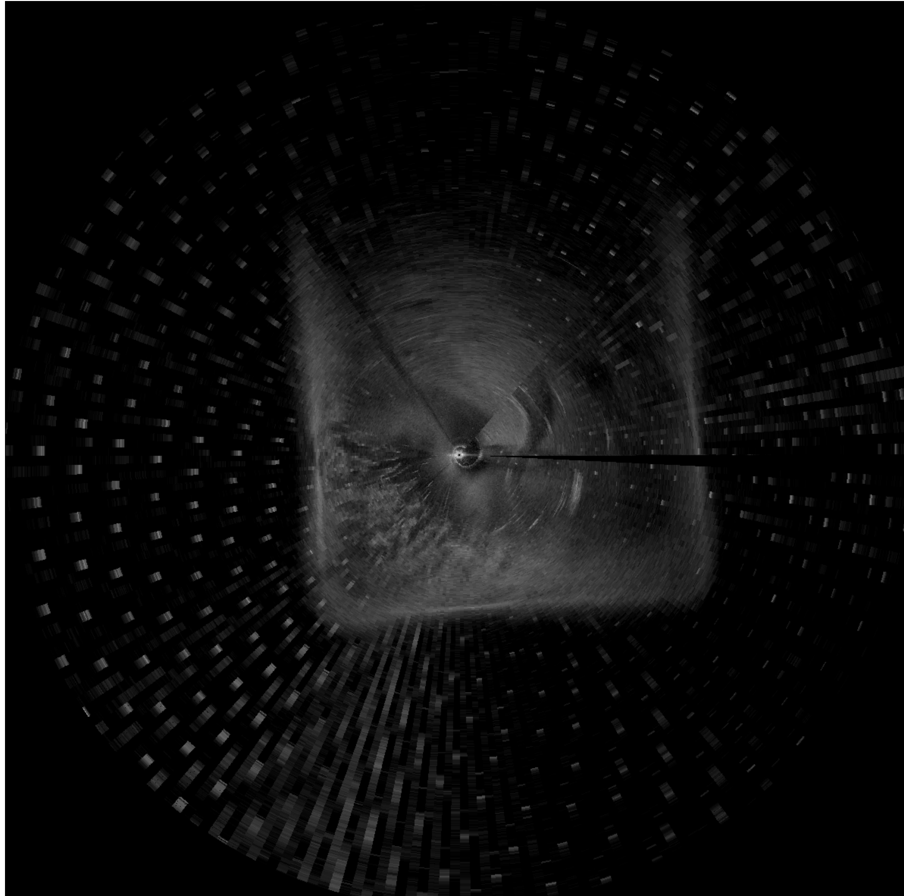


Figure 2.17: Rohdaten des Sonars. Zu erkennen sind sowohl der Boden, die Pipeline, die Wände als auch Störsignale.

#### 2.2.1.5 Das Basiskontrollsystem Avalons

Wie bereits am Anfang dieses Kapitel erwähnt, ist es von essentieller Bedeutung, stets in der Lage zu sein, Kontrolle über ein System zu erlangen. Dies gilt insbesondere, wenn das System unter echten Bedingungen eingesetzt wird. Diese Anforderung gilt speziell auch im Fall des Versagens von Softwarekomponenten. Ein AUV sollte jederzeit in einen Remotely Operated (underwater) Vehicle (ROV)-ähnlichen Operationsmodus versetzt werden können, um eventuelle Kollisionen zu vermeiden oder um das System bergen zu können. Im Gegensatz zu anderen Systemen besteht bei AUVs nur selten oder mit hohem Aufwand die Möglichkeit, direkt mit dem System zu interagieren.

Figure 2.18: Geöffnete Front des Avalon-Systems. Zu erkennen sind zwei Prosilica-Kameras für die Erfassung der Objekte unter und vor dem Fahrzeug.

Komplexe High-Level-Software, insbesondere in experimentellen Systemen, eignet sich daher nur eingeschränkt für eine solche Steuerung. Einerseits ist die Software in einem stetigen Wandel, andererseits kann es zu einer Überlastung der Systeme kommen, die zur Folge hat, dass die Systeme ggf. nicht mehr ausreichend Reaktionszeiten liefern.

Aus diesem Grund wurde für Avalon eine Notfallsteuerung auf der Basis von eingebetteten Schaltungen gewählt. Dazu wurde zusätzlich zu den Vierquadrantenstellern eine Elektronik verbaut, die über den CAN-Bus einerseits in der Lage ist, mit dem PC-System zu kommunizieren, andererseits aber auch mit den Vierquadrantenstellern selbst. Darüber hinaus besitzt diese Steuerungselektronik Schnittstellen zu allen relevanten Kommunikationskanälen.

Zu diesen Kommunikationskanälen zählt die serielle 876-MHz-Amber-Funkverbindung. Amber bietet eine robuste Kommunikation, sobald das System aufgetaucht ist. Ergänzend dazu dient ein Unterwassermodem zur Aufrechterhaltung eines Kommunikationskanals, sofern das System abgetaucht ist. Diese Kommunikationskanäle dienen, unabhängig vom High-Level-PC-System, der Basissteuerung des AUVs. Da das Unterwassermodem nur über eine eingeschränkte Bandbreite verfügt, sind über diesen Kanal nur Kommandos wie *Notauftauchen* möglich. Um während der alltäglichen Forschungsarbeit auf die zuverläs-

sige Notfallsteuerung zugreifen zu können, ist neben dem Amber-Funkmodul parallel ein Ethernet-Seriell-Wandler an den Mikrocontroller der Notfallsteuerung angebunden. Somit ist es möglich, das Notfallsteuerungsmodul über Wifi, das Glasfaserkabel sowie das Kupferkabel PC-unabhängig zu nutzen.

Wie bereits zuvor angesprochen, bietet das CAN-Protokoll durch priorisierte Pakete die Möglichkeit, niederpriorisierte Nachrichten zu überschreiben. Die Vierquadrantensteller wurden so programmiert, dass sie zwischen niedrig- und hochpriorisierten PC-Kontrollnachrichten der Notfallsteuerung unterscheiden. Somit ist durch die eingebettete Programmierung und die Eigenschaften des CAN-Protokolls sichergestellt, dass jederzeit eine Kontrolle über das AUV möglich ist. Angemerkt sei an dieser Stelle, dass theoretisch jede Komponente, also auch der PC, am CAN-Bus hochpriorisierte Nachrichten versenden könnte. Dies wird jedoch als unwahrscheinlich erachtet, da Nachrichten dem PC unbekannt sind, solange keine explizite Programmierung der relevanten Kommandos vorliegt.

Angesichts der eingeschränkten Bandbreite des Unterwassermodems wurde nur ein Notauftauchen über diesen Kanal realisiert. Sollte dies nötig sein, würde das Fahrzeug an die Wasseroberfläche zurückkehren und dann über den 867-MHz-Kanal wieder steuerbar sein. Weitere Anpassungen könnten dann über den WiFi-Kanal vorgenommen werden, sobald eine Verbindung hierüber etabliert worden ist. Um die Funktionalität des Unterwassermodems auch für andere High-Level-Anwendungen nutzen zu können, besteht eine bidirektionale Gatewayfunktionalität zwischen dem UW-Modem und dem CAN-Bus. Somit kann das High-Level-PC-System Modemnachrichten über den CAN-Bus absetzen und lesen. Die gleiche Funktionalität ist ebenfalls für das Amber-Protokoll realisiert. Eine Gesamtübersicht dieser Kommunikationswege ist in Abbildung 2.19 zu sehen.

Um die verschiedenen Geräte von der Bodenstation robust bedienen zu können, wurde eine Operators Control Unit (OCU) entworfen. Diese bietet die Möglichkeit, sämtliche Kommunikationskanäle zu verwenden, und ist somit die Basis-Steuerungseinheit des Avalon-Systems. Durch diese einheitliche Steuerungsschnittstelle ist es möglich, das Avalon-System ohne weitere Hardware zu bedienen. Die OCU wurde auf Basis eines minimalen Linuxsystems in Kombination mit einem *Gumstix Overo Fire* realisiert. Zusätzliche Eingaben werden über eine, eigens hierfür entworfene, Auswerteelektronik über den USB-Port als General Purpose Input Output (GPIO) Schnittstelle eingespeist. Die OCU, die in Abbildung 2.20 zu sehen ist, verfügt über einen eigenen Akku, WiFi, Bluetooth und das 867-MHz-Funkmodem als Gegenseite der Kommunikation. Das Oberflächen-Tritech-Modem kann extern angeschlossen werden, um hierüber Kommandos an Avalon zu senden.

Um einerseits bei einem Stromausfall, aber auch allgemein für Feldtests die OCU nutzbar zu halten, besitzt sie eine eingebaute Batterie. Neben Batteriebetrieb kann die OCU auch extern mit Strom versorgt werden. Die Schaltung ist dabei so gewählt, dass ein kontinuierlicher Betrieb auch beim Wechsel der Versorgung möglich ist. Die OCU ist wasserdicht

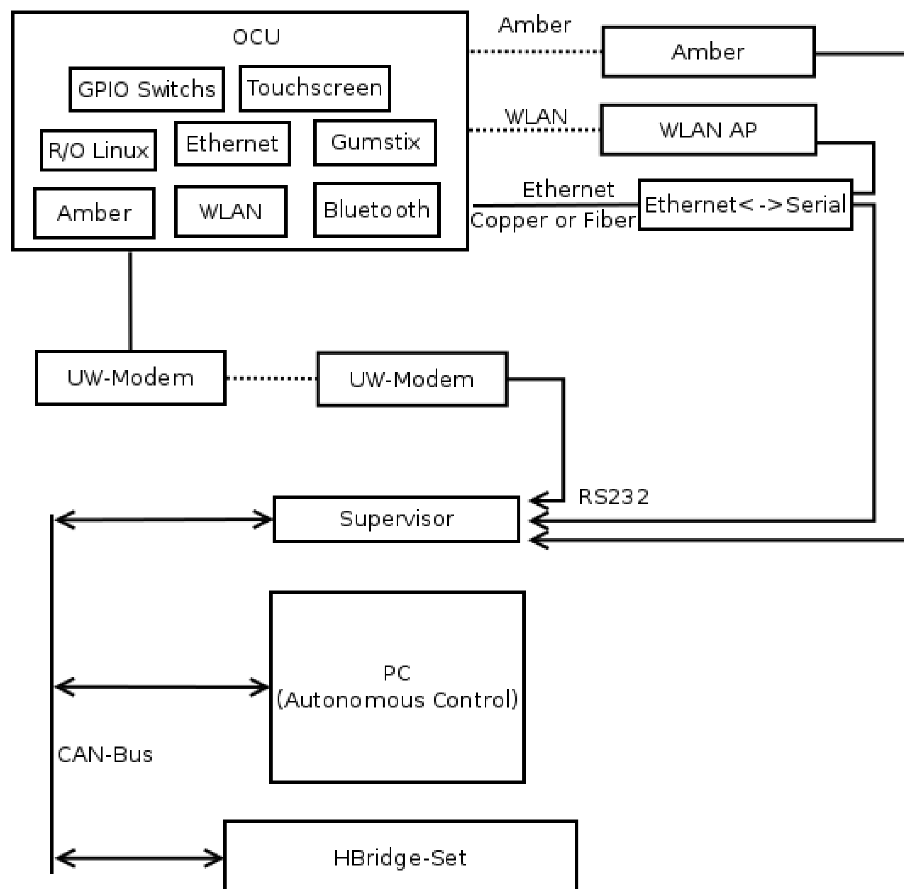



Figure 2.19: Schematische Darstellung der Steuerung über die Operators-Control-Unit (OCU) in Avalon.

konzipiert. Dadurch kann sie selbst dann benutzt werden, falls sie ins Wasser fallen sollte. Hierzu kommen Bulgin-Stecker zum Einsatz, die dem IP86-X-Standard entsprechen. Da über die OCU eine Steuerung des Systems möglich ist, wird zusätzlich eine externe Eingabequelle benötigt. Die Wahl fiel hierbei auf die Graupner-MC-20-Fernbedienung (Abbildung 2.21), die auch im Modellbausektor verwendet wird. Die Wahl dieser Steuerung liegt einerseits in ihrer Robustheit, andererseits in der Möglichkeit begründet, die Fernbedienung mittels eines SUM-D-Protokolls direkt über einen seriellen Port am Empfänger auszulesen. Die verschiedenen Schalter der MC-20 erlauben die Belegung mit verschiedenen Steuermodi, sodass das System für verschiedene Verhaltensweisen im Voraus programmiert werden kann und während der Wettbewerbe oder im Außeneinsatz von einer Person

(a) Basiskontrolleinheit für das Avalon-System während ihres Einsatzes am CMRE.

(b) Gumsix Overo Fire SoC Prozessor der OCU (Quelle: gumstix.com)

leicht steuerbar ist. Dies ... Avalon, ohne ein Laptop nutzen zu müssen, was in F...

Figure 2.21: Graupner-MC-20-Steuerung für das Avalon-System.

### 2.2.2 Vorstellung des Systems Dagon

Das Dagonsystem entstand im Rahmen der Evaluation eines optischen Simultaneous Localization and Mapping (SLAM)-Verfahrens [29]. Dabei wurde in der Arbeit Dagon als Nachfolgeplattform des Avalon-Systems dargestellt. Die zuvor aufgeführten Änderungen an Avalon erfolgten jedoch nach der Fertigstellung von Dagon selbst. Bei der Konzeption von Dagon wurde primär auf ein verbessertes System mittels GPU sowie das Stereokamerasystem Wert gelegt, das für den optischen Stereo-SLAM verwendet wurde. An dieser Stelle wird von einer kompletten Systembeschreibung von Dagon abgesehen. Der Entwurf und die Konzepte des Systems wurden bereits ausführlich in der Abhandlung [30] beschrieben. Stattdessen werden im Folgenden die wesentlichen Unterschiede zwischen den Systemen aufgeführt, um im Anschluss an dieses Kapitel Vor- und Nachteile zu erörtern.

#### 2.2.2.1 Dagons Stromversorgung

Dagon besitzt ein proprietäres Batteriesystem (siehe Abb. 2.22) mit einer Kapazität von 1.5 kW/h. Der Hersteller liefert zusätzlich zu diesem System ein Lade- und Überwachungssystem für den kompletten Batteriesatz. Dies ermöglicht während des Ladevorgangs eine vollständige Zellenüberwachung und einen schnellen Ladevorgang.

#### 2.2.2.2 Dagons Kamerasystem und optisches SLAM-Verfahren

Im Unterschied zu Avalon besitzt Dagon zwei nach unten gerichtete Kamerasysteme (siehe Bild 2.23. Die Kameras sollten genutzt werden, um ein auf Stereokamera basierendes SLAM-Verfahren zu realisieren. In den Wettbewerben wurde jedoch auf den Einsatz dieses Verfahrens verzichtet, da es nicht für die Umgebungsbedingungen, die in dem SAUC-E- und euRathlon-Szenario erwartet wurden, vorgesehen war.

#### 2.2.2.3 Die Aktuierung von Dagon - die Thruster

Die Thruster von Dagon werden mittels einer proprietären Elektronik betrieben. Bei den Motoren handelt es sich, im Gegensatz zu den in Avalon eingesetzten Motoren, um bürstenlose Antriebe. Diese Antriebe besitzen den Vorteil, dass die Umdrehungsgeschwindigkeit direkt regelbar ist. Im Gegensatz zu den bürstengetriebenen Motoren erhält der Benutzer somit stetig eine Rückmeldung über den aktuellen Status der Antriebe. Bei den Motoren selbst handelt es sich um eine eigene Konzeption, die für das Fahrzeug Dagon erstellt wurde. Einer der Nachteile der Konzeption ist jedoch die Anordnung der Thruster an Dagon. Das Dagon-System besitzt, ebenso wie Avalon, fünf ansteuerbare Achsen, jedoch

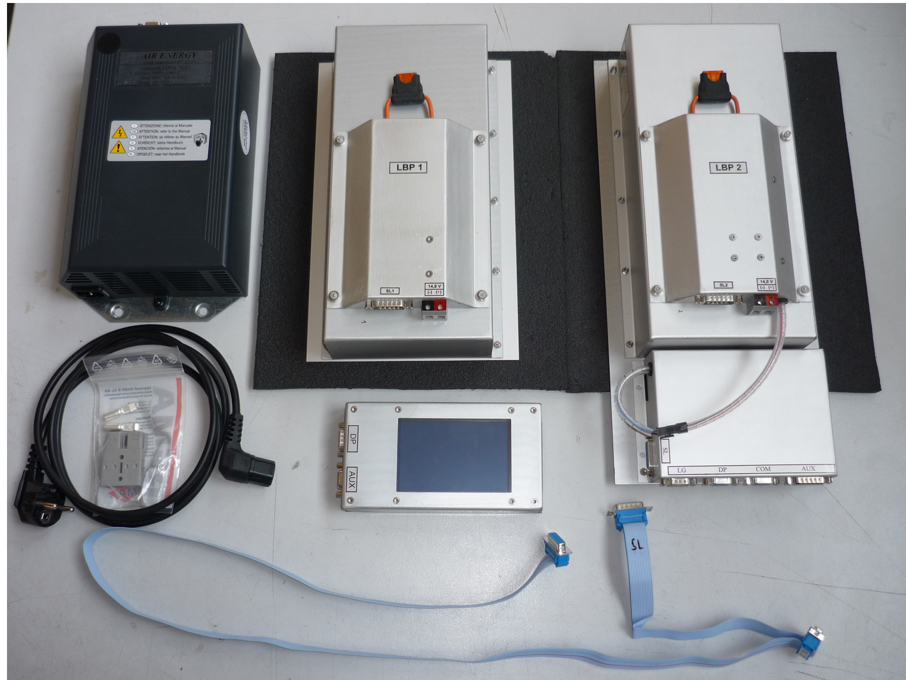



Figure 2.22: Batteriesystem des Dagon-Systems. (Quelle: DFKI)

wurde auf den mittleren vertikalen Antrieb verzichtet. Durch diesen fehlenden Thruster ist eine Seitwärtsfahrt mit dem System nur langsam möglich.

#### 2.2.2.4 Dagons Steuerungs- und Kommunikationskonzept

Im Gegensatz zu Avalon hat Dagon neben der PC-Steuerung keine Möglichkeit, die Motoren zu kontrollieren. Die Motoren sind direkt mit den zwei PC-Systemen verbunden. Dabei werden vier der fünf Motoren von einem PC gesteuert, der verbleibende Motor wird hingegen von dem zweiten PC betrieben. Somit müssen stetig beide PC-Systeme in Betrieb sein, um Dagon steuern zu können. Diese verteilte Ansteuerung der Motoren erfordert somit eine stetige Synchronisation beider PCs. Auch bietet Dagon keinen 876-MHz-Kommunikationskanal oder ein fest installiertes WiFi. Stattdessen wurde eine externe Box (Bild: 2.24) an Dagon befestigt, die bei Bedarf anstelle des Kupferkabels verwendet werden kann. Darüber hinaus besitzt Dagon kein (funktionierendes) Unterwassermodem. Das im System vorhandene Glasfaserkabel sowie die vorhandenen Steckerverbinder sind nicht feucht steckbar, wodurch das System bei jeder Umkonfiguration aus dem Wasser geholt werden muss.

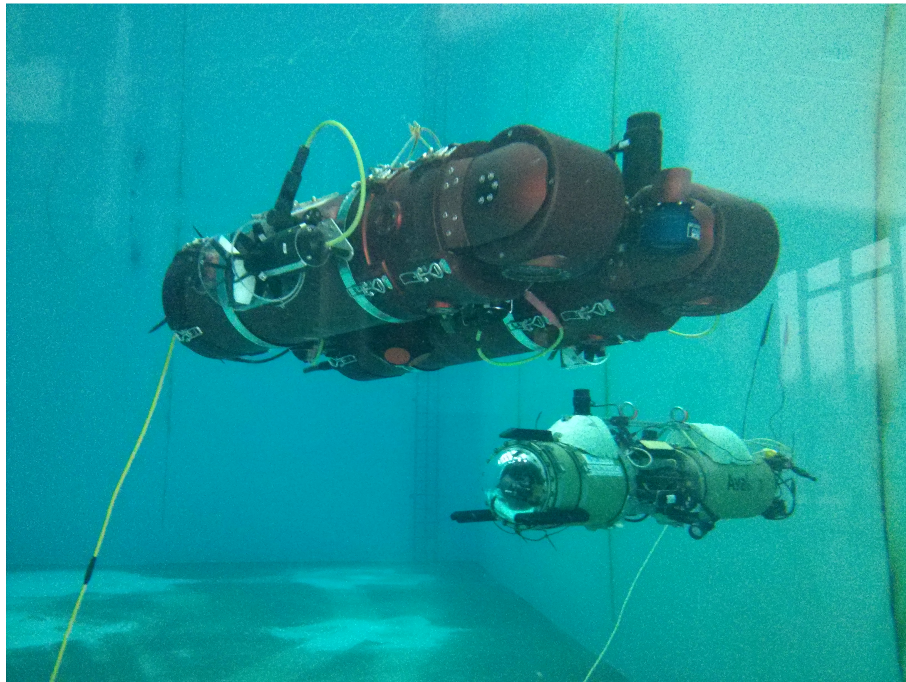


Figure 2.23: Dagon und Avalon im Testbecken des DFKI. Zu erkennen sind die zwei nach unten gerichteten Kameras.

Um die fehlende Grundlagensteuerung, wie sie die Konzepte von Avalon vorsehen (vergl. 2.2.1.5), zu kompensieren, wurde eine Softwarelösung gewählt. Eine Komponente empfängt über den TCP-Stack Steuerungskommandos für die Motoren. Dies geschieht analog zu dem bei Avalon eingesetzten *Mainboard*. Diese Softwarelösung beendet die High-Level-Software und öffnet, anstelle von ihr, alle seriellen Ports zu den Antrieben. Diese Realisierung setzt jedoch einen funktionsfähigen PC voraus. Da die Motoren an verschiedenen Rechnern innerhalb des Dagon-Systems angeschlossen sind, musste auch eine zusätzliche Lösung zur Kommunikation dieser beiden Systeme untereinander geschaffen werden, was die Fehleranfälligkeit deutlich erhöht.

#### 2.2.2.5 Navigations- und Lokalisierungsfähigkeiten von Dagon

Ein Vorteil von Dagon im Vergleich zum Avalon-System ist, dass es einen DVL-Sensor besitzt. Das DVL ermöglicht eine präzise Bestimmung der AUV-Bewegungen über dem Grund. Dabei nutzt es den Dopplereffekt, um die Bewegungen in alle Richtungen auf wenige cm/s genau bestimmen zu können. Im Gegensatz zu Avalon verwendet Dagon

Figure 2.24: Dagons WiFi-Zugangspunkt, der anstelle des Kupferkabels verwendet werden kann, in einer wasserdichten Box (gelb mittig).

einen kalibrierten digitalen Tiefensensor, der die Tiefe des Systems exakt bestimmen kann. Dieser arbeitet jedoch mit 10 Hz deutlich langsamer als der von Avalon. Zusätzlich zu dieser Sensorik besitzt Dagon einen *LBL-Responder*. Mit ihm ist Dagon in der Lage, seine Position innerhalb eines Arrays von LBL-Stationen global genau bestimmen zu können. Dieses System darf jedoch während der hier behandelten Wettbewerbe nicht eingesetzt werden und erfordert eine externe Infrastruktur.

#### 2.2.2.6 Auf Dagon eingesetzte Sensorik

Neben den angesprochenen Sensoren verfügt Dagon über ein ähnliches Sensorkonzept wie Avalon. Dagon besitzt eine XSens IMU, einen faseroptischen einachsigen Kreisel sowie eine zusätzliche, nach vorne gerichtete Kamera. Als Hauptsensor steht Dagon ebenfalls ein Micron-Scanning-Sonar zur Verfügung, womit eine globale Lokalisierung in strukturierten Umgebungen möglich ist. Statt eines speziellen Altimeters kann bei Dagon das DVL zur Bestimmung der Distanz über Grund genutzt werden.

## 2.3 Softwarekomponenten der beiden AUV-Plattformen

Die Software ist für autonome Systeme von essentieller Bedeutung. In diesem Kapitel werden die Kernkomponenten vorgestellt, die in den Systemen eingesetzt werden. Ihre Vor- und Nachteile werden detailliert betrachtet und analysiert. Es existieren teilweise redundante Lösungen für das gleiche Problem, wie beispielsweise zur Lageschätzung 2.3.2,2.3.2. Die Generierung der Komponentennetzwerke wird mittels eines Plan-Managers (siehe Abschnitt 2.4) durchgeführt.

### 2.3.1 Die Lageschätzung für AUVs

Die Lageschätzung ist ein essentieller Bestandteil moderner AUV-Systeme. Es ist von enormer Bedeutung, dass insbesondere bei stromlinienoptimierten Fahrzeugen die Orientierung des Systems genau bekannt ist, da das System ansonsten durch Strömungseffekte abdriften würde und aktiv gegen diesen Fehler gegenregeln müsste. Unterwassersysteme unterliegen, bedingt durch ferromagnetische Ablagerungen an den Hafenwänden sowie durch natürliche Mineralvorkommen, starken Magnetfeldverzerrungen. Aus diesem Grund sind Magnetfeldinformationen nur eingeschränkt nützlich, um die Nord-Süd-Achse (Yaw-Achse, oder auch Drehung um die Z-Achse) bestimmen zu können. Deswegen wurde für Avalon ein System gewählt, das die Orientierung mittels klassischer elektronischer Gyroskope sowie Beschleunigungsmesser und eines einachsigen Glasfaserkreisels schätzt. Faseroptische Kreiselkompasse unterliegen einem geringen Drift und weisen eine hohe Auflösung auf. Diese Sensoren wurden mittels eines in [27] vorgestellten Filters kombiniert. Dieser Kalmanfilter [66] eignet sich besonders für normal verteilte Messwerte ohne Messausreißer. Die initiale Nordausrichtung wird beim Systemstart auf die aktuelle Systemausrichtung festgelegt. Diese muss einmalig händisch bei jedem Experiment determiniert werden.

### 2.3.2 Lageregelungstreiber

**Minimales Modell** Es wurden während der Projektzeit zwei verschiedene Lageregelungen für die Systeme entworfen. Der erste Regelungsalgorithmus basiert auf PID-Reglern, die die Eingaben auf die Antriebe abbilden.

Die Abbildung der Kräfte für das Avalon-System findet wie folgt statt:

$$t_l = \dot{x}\lambda_x \tag{2.1}$$
$$t_r = \dot{x}\lambda_x \tag{2.2}$$
$$t_{r_h} = \dot{y}\lambda_y + \alpha_h\dot{\psi} \tag{2.3}$$
$$t_{r_v} = \dot{z}\lambda_z + \alpha_v\dot{\theta} \tag{2.4}$$
$$t_v = \dot{z}\lambda_z \tag{2.5}$$
$$t_h = \dot{y}\lambda_y \tag{2.6}$$

Hierbei entspricht $t_{l,r,r_h,r_v,v,h}$ den Motoren links, rechts, hinten horizontal, hinten vertikal, mittig vertikal und mittig horizontal. Die Geschwindigkeiten bzw. Positionen entlang der drei Achsen werden als $x, y, z$ und die Rotationen um die entsprechenden Achsen als $\phi, \theta, \psi$ dargestellt. $\lambda$ beschreibt die Skalierungsfaktoren für die verschiedenen Antriebe und Achsen und $\alpha$ den Zusammenhang zwischen mittleren und hinteren Motoren.

Das Modell vereinfacht viele Attribute, beispielsweise wird durch das symmetrische Ansteuern des linken und rechten Thrusters $t_l, t_v$ angenommen, dass beide Thruster sich identisch verhalten und das AUV perfekt symmetrisch aufgebaut wurde. Ebenso wird die Trägheit des Systems hierbei vernachlässigt. Trotz dieser starken Vereinfachungen ist das System grundlegend steuerbar.

Der erste Teil dieser minimalen Reglerkette abstrahiert die Regelung eines AUV-Systems annähernd auf ein 2D-Problem, da die Tiefe bedingt durch die Tiefenmessung (siehe Abschnitt 2.2.1.4) gut bekannt ist. Außerdem muss der Pitch (Drehung um die Y-Achse) nicht beachtet werden. Das AUV richtet sich selbstständig gerade im Wasser aus, daher wird der Pitch als 0 angenommen.

Daraus ergibt sich die folgende Kontrollmatrix:

$$\dot{x} = \dot{x} \tag{2.7}$$
$$\dot{y} = \dot{y} \tag{2.8}$$
$$\dot{z} = K_p(z - PV_z) + K_i \int_0^t (z - PV_z)(r)\,dr + K_d \tfrac{\delta(z-PV_z)(t)}{dt} \tag{2.9}$$
$$\dot{\psi} = K_p(\psi - PV_\psi)(t) + K_i \int_0^t (\psi - PV_\psi)(r)\,dr + K_d \tfrac{d(\psi-PV_\psi)(t)}{dt} \tag{2.10}$$
$$\dot{\theta} = K_p(\theta - PV_\theta)(t) + K_i \int_0^t (\theta - PV_\theta)(r)\,dr + K_d \tfrac{d(\theta-PV_\theta)(t)}{dt} \tag{2.11}$$

Hierbei ist $PV$ die aktuelle Position der entsprechenden Reglereingabe und $K_p, K_i, K_d$ die Regler-Parametrisierung des PID-Regleranteils.

Somit geschieht die Steuerung auf Basis von $\dot{x}, \dot{y}, z, \theta, \psi$, wobei $\phi$ unaktuiert bleibt, da das Avalon-System durch die Massenverteilung in der Regel aufrecht entlang der Rollachse

liegt. Eventuelle Abweichungen werden vernachlässigt, da diese Achse durch die Aktuatorenanordnung nicht aktuierbar ist.

Um Regelungsaufgaben im Roboterkoordinatensystem (bspw. auf Basis von Kameradaten) vornehmen zu können, wurde ein sogenannter *relative-position-controller* eingeführt. Dieser ermöglicht die Regelung in verschiedenen Kombinationen von Koordinatensystemen. Diese Separierung wurde getroffen, damit einzelne Detektoren nicht auf Weltkoordinaten, wie die Ausrichtung $\psi$ oder die Tiefe $z$, angewiesen sind. Es ist somit nicht nötig, dass der Bojenerkenner die relative Ausrichtung der Boje in Weltausrichtung sowie die Tiefe bestimmt. Durch den eingeführten Zwischenschritt ist es möglich, dass die Regelung auf der Basis von relativen Bewegungsinformationen vorgenommen wird. Diese Separierung reduziert den Entwicklungsaufwand einzelner Komponenten und definiert eine dedizierte Komponente, die die Regelung vornimmt.

$$\dot{x} = K_p(x - PV_x) + K_i \int_0^t (x - PV_x)(r)\, dr + K_d \frac{\delta(x-PV_x)(t)}{dt} \tag{2.12}$$

$$\dot{y} = K_p(y - PV_y) + K_i \int_0^t (y - PV_y)(r)\, dr + K_d \frac{\delta(y-PV_y)(t)}{dt} \tag{2.13}$$

$$z = \begin{cases} PV_z + \Delta z, & \text{if rel}_z \\ z, & \text{else} \end{cases} \tag{2.14}$$

$$\psi = \begin{cases} PV_\psi + \Delta \psi, & \text{if rel}_\psi \\ \psi, & \text{else} \end{cases} \tag{2.15}$$

$$\phi = PV_\phi + \Delta \phi \tag{2.16}$$

Je nach Anwendungsfall ist es möglich, eine relative anstatt globale Regelung auf Basis der Tiefe $z$ oder des Headings $\psi$ vorzunehmen. Zusätzlich müssen verschiedene Informationen für den $PV$-*Ist*-Zustand des Systems dem Regler übergeben werden. Beispielsweise würde eine Regelung nach Weltkoordinaten die aktuelle Position aus der Lokalisation in dem $PV$-Eingabevektor benötigen. Eine Pipelineverfolgung hingegen wäre nur auf die Position der Pipeline unter dem AUV angewiesen. Die Regler-*Soll*-Eingabe wäre in diesem Fall konstant null, um das System mittig über der Pipeline zu halten. Des Weiteren sind je nach Anwendungsszenario verschiedene Parametrisierungen für die $K$-Reglermatrix nötig. Zudem muss auch festgelegt werden, ob auf einer absoluten oder relativen Ausrichtung geregelt werden soll. Der Unterschied zwischen diesen Regelungen ist, dass ein Anfahren von Weltkoordinaten anhand globaler Koordinaten geschieht. Im Gegensatz dazu liefern Detektoren oftmals Informationen im Roboterkoordinatensystem, somit muss das System in der Lage sein, auch auf der Basis von relativen Informationen regeln zu können. Auch diese Parametrisierung erfolgt anwendungsspezifisch und ist Bestandteil des Bestimmung

des Komponentennetzwerkes, wie später in Abschnitt 2.4 beschrieben.

**Erweitertes Modell zur Lage und Positionsregelung** Das zweite Modell berücksichtigt, im Gegensatz zu dem ersten, weitere physikalische und konzeptuelle Eigenschaften des Systems. Das erste, minimale Modell vernachlässigte diverse Parameter. Somit stellt das zweite Modell eine Erweiterung der ersten Version dar. Dabei wurde speziell auf diese Attribute Wert gelegt:

- Korrektere Umsetzung physikalischer Eigenschaften
- Verifizierbarkeit der Korrektheit der Konfiguration
- Portabilität zwischen verschiedenen Trägersystemen
- Modulare Funktionsweise für zukünftige Anwendungen und Szenarien
- Mathematisch striktere Trennung zwischen verschiedenen Einheiten und Koordinatensystemen

Aus den oben genannten Gründen wurde eine dreischichtige Architektur der Reglerkette gewählt. Im ersten Schritt wird, ähnlich wie bei der Avalon-spezifischen Reglerkette, die Abstraktion zwischen Kraftvektoren und dem System vorgenommen. Dieser Regler wird *Acceleration-Controller* genannt. Die Abstraktion wird mittels einer Thrustermatrix $T^{6\times m}$ vorgenommen, wobei $m$ die Anzahl der zu aktuierenden Motoren darstellt. Jede Zeile stellt dabei den Einfluss eines Thrusters auf die Welt dar, also die aufgetretenen Kräfte in die Richtungen $\ddot{p} = \ddot{x}, \ddot{y}, \ddot{z}, \ddot{\phi}, \ddot{\theta}, \ddot{\psi}$. Mittels einer Matrixmultiplikation zwischen $\ddot{p} \cdot T$ ergeben sich die nötigen Steuerkommandos für die Thrustereingaben für einen gegebenen Kraftvektor in Roboterkoordinaten $\ddot{p}$.

Für das Konzept wurde angenommen, dass AUVs in der Regel in einer Ebene operieren und die Rotationen um die Roll- oder Pitch-Achse im Normalfall durch Regler- oder Messungenauigkeiten entstehen. Aus diesem Grund wurde ein sogenannter *Body-aligned-Frame* eingeführt. Dieses Koordinatensystem hat den Ursprung im Roboterkoordinatensystem. Die xy-Ebene ist jedoch identisch mit dem Weltkoordinatensystem. Dies bedeutet, dass der Pitch/Roll (sofern aktuierbar) ignoriert wird. Dies hat insbesondere den Vorteil, dass eventuelle Fehler im Pitch nicht zum Ab-/Auftauchen[1] des Fahrzeuges führen würden. Der nächste Controller mit dem Namen *AlignedToBody* setzt das kraftbasierte System des *Acceleration-Controllers* in das Roboterkoordinatensystem um.

[1] einen idealen Controller vorausgesetzt

$$\ddot{p}_{x,y,z} = (AngleAxis(-\psi, Z)PV_{\phi,\theta,\psi})\ddot{p}_{x,y,z} \tag{2.17}$$
$$\ddot{p}_{\phi,\theta,\psi} = \ddot{p}_{\phi,\theta,\psi} \tag{2.18}$$

Zu beachten ist, dass keine Anpassung der Rotationskräfte nötig ist, da diese orientierungsunabhängig sind. Die nächste Ebene der Kontrollkette stellt der *AlignedVelocity*-Controller dar. Wie der Name Andeutet, abstrahiert der Kontroller die Kräftematrix der Thruster zu Geschwindigkeiten. Als Basis für die Regelung wurde ein PID-Regler verwendet. Zur vereinfachten Schreibweise wird im Folgenden die Form $PID(Soll, Ist)$ benutzt. Zu beachten ist, dass der integrale Anteil der Regler nach oben begrenzt wird, um ein zu starkes Anwachsen durch ein beispielsweise festhängendes System zu verhindern.

$$\ddot{p} = PID(\dot{p}, \dot{P}V) \tag{2.19}$$

Analog zum *AlignedVelocity*-Controller existiert ein *AlignedPosition*-Controller, der die Umsetzung von Geschwindigkeiten auf Positionen vornimmt.

$$\dot{p} = PID(p, PV) \tag{2.20}$$

Ein weiterer Controller übernimmt die Umsetzung von Weltkoordinaten in das *Aligned-Body*-Frame. Hierzu werden die Weltpositionen auf das körpereigene Koordinatensystem abgebildet.

$$p. = p * PV_{\phi,\theta,\psi} \tag{2.21}$$

**Flexibilität** Die Szenarien, für die Avalon ausgestattet sein soll, unterscheiden sich signifikant in den Regelungsaufgaben. Es ist bis auf ein Szenario nie der Fall, dass der gesamte Kontrollvektor $p$ von einer Komponente erzeugt wird. In diesem Zusammenhang ist es nötig, neben der oben genannten formalen Definition des Regelverhaltens diverse Aspekte in die Software-Architektur miteinzubeziehen. Die Möglichkeit, flexibel verschiedene Eingaben mischen zu können, ist dabei eine wesentliche Fähigkeit, die im Folgenden erläutert wird.

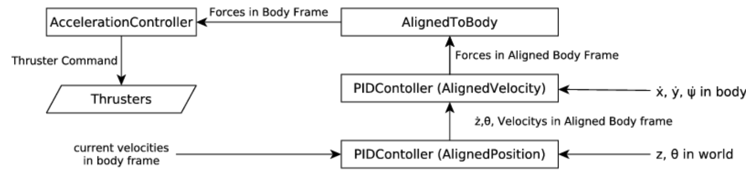



Figure 2.25: Reglerkette für ein spiralförmiges Abtauchen.

Die Pipelineverfolgung benötigt beispielsweise eine Regelung der y-Position im Body-aligned-Frame bei einer konstanten Fahrgeschwindigkeit, während die Regelung des Headings relativ geschehen muss. Die Tiefe kann entweder relativ über den Pipelinedetektor erfolgen, als fester Wert in Relation zum Boden vorgegeben oder als fester Tiefenwert festgelegt werden (Abbildung: 2.26). Die Komponente zur Wandverfolgung verhält sich ähnlich. Sie ermittelt in Roboterkoordinaten einen festen Abstand und eine relative Ausrichtung zur Wand. Für die Bojeninspektion ist eine konstante Drehung um die Boje vorgesehen. Dies soll erreicht werden, indem die Boje mittels einer Drehung des AUVs mittig im Bild gehalten wird, während sich das Fahrzeug konstant seitwärts bewegt. Die Höhe im Bild wird als relative z-Information benötigt, die y-Achse als konstante Geschwindigkeit und die Drehung $\psi$ als aktuelle Reglereingabe. Durch die Regelung auf die Bildmitte dreht sich das Fahrzeug automatisch um das Zielobjekt. Die x-Position kann als relative Position des Bojendetektors bestimmt werden, um die Boje in einer festen Größe im Bild zu halten. Ein Beispiel, das die Flexibilität der Reglerkette verdeutlicht, ist in der Abbildung 2.25 zu sehen: Ein spiralförmiges Abtauchen eines AUVs bei angestellter Drehung um die y-Achse ($\theta$, Pitch) sowie einer Zieltiefe $z$ führt bei einer konstanten Fahrtgeschwindigkeit $\dot{x}$ zum effizienten und schnellen Abtauchen eines stromlinienförmigen AUVs. In der Abbildung ist erkennbar, dass die finale Reglereingabe sich aus dem Zusammenführen der verschiedenen Kontrollinformationen bildet.

Um die Fehlersicherheit zu erhöhen, werden verschiedene zusätzliche Maßnahmen getroffen. Einerseits wird über eine Konfiguration vermerkt, welche Achsen jede Ebene der Kontrollkette steuern soll. Andererseits wird überwacht, ob diese geforderten informationen während der Regelung vorliegen. Zusätzlich wird sichergestellt, dass für jeden Eingabedatenstrom die Daten nicht älter als die höchste zu erwartende Latenz sind und dass sie auch nur exakt jene Informationen beinhalten, die über diesen Kanal eingeschleust werden sollen. Diese komplexe Konfiguration erhöht die Sicherheit der Controllerkette. Im Fall einer inkorrekten Verwendung der Controller kann somit sichergestellt werden, dass das System in einen definierten Fehlerzustand übergeht. Ohne diese Konfiguration würde das System widersprüchliche Daten zusammenzufassen, was zu einer undefinierten Bewegung des Systems führen würde.

Unabhängig von der Reglerkette wurde eine Sicherheitskomponente eingeführt, die die Zieltiefenwerte gegen die aktuelle Wassertiefe evaluiert. Bei einem Unterschreiten eines Sicherheitsabstandes zum Boden wird die Zieltiefe mit der erlaubten Maximaltiefe überschreiben. In diesem Fall wird eine Warnung an die höheren Ebenen übergeben. Diese zusätzliche Instanz, die in Abbildung 2.27 zu sehen ist, dient dazu, unvorhergesehene Untiefen reaktiv zu umgehen, um das System nicht auf Grund auflaufen zu lassen.

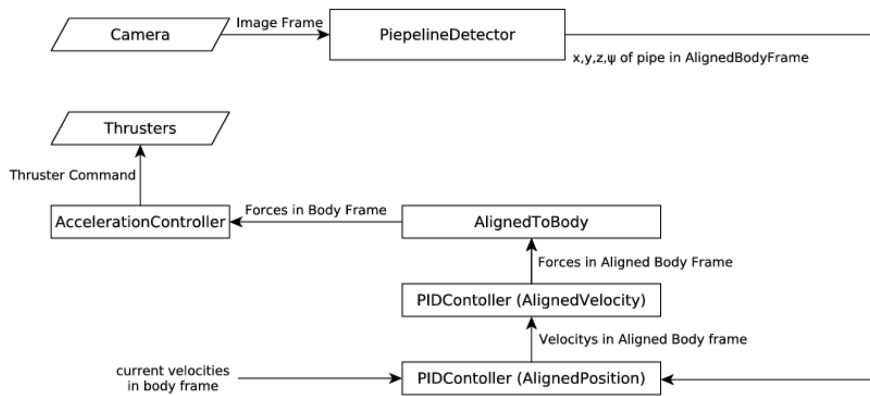



Figure 2.26: Regelkreis der Pipelineverfolgung.

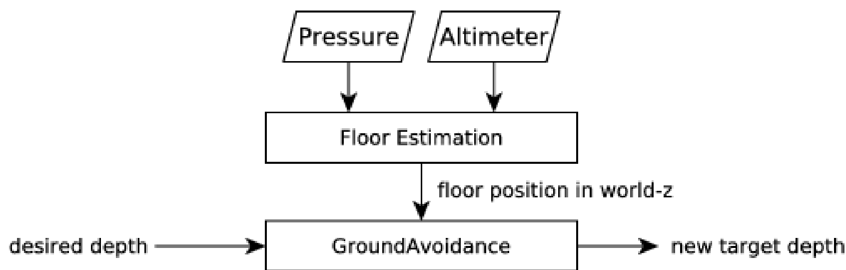



Figure 2.27: Eingefügtes Zwischenmodul zur Vermeidung von Bodenkontakten.

## 2.4 Vorstellung der Komponentennetzwerke

Dieser Abschnitt der Arbeit beschäftigt sich mit den Komponentennetzwerken, also der Behandlung einzelner Softwaremodule, um ein gewünschtes Systemverhalten herbeizuführen. Ähnlich wie Softwarebibliotheken bieten Komponentennetzwerke eine Möglichkeit, einzelne

Komponenten einfacher wiederzuverwenden. Um die Komponentennetzwerke zu erstellen, werden verschiedene Frameworks wie Robot Operating System (ROS) oder The Robot Construction Kit (Rock) benutzt. Diese definieren Schnittstellen, die eine Komponente erfüllen muss. Des Weiteren ist es je nach eingesetztem Framework möglich, eine Modellbeschreibung über zusätzliche Schnittstellen oder Eigenschaften der Komponente zu definieren. Das Framework Rock bietet zudem zusätzlich die Option, Komponenten extern zu steuern. Somit ist es möglich, von einer übergeordneten Schicht aus zu entscheiden, ob eine Komponente aktiv sein soll oder nicht.

Die Selektion solcher Komponenten geschieht mittels eines sogenannten *Plan-Managers*. Der *Plan-Manager* ermittelt die benötigten Komponenten für gegebene Anforderungen an das System. Anforderungen könnten beispielweise sein: *benötige Orientierung* oder *verfolge Pipeline*. Das Verfolgen einer Pipeline erfordert verschiedene Teilkomponenten wie eine Kamera, den Pipeline-Detektor, Regelung und weitere. Die Abbildung bzw. Identifikation muss systemspezifisch erfolgen. Gerade in Anbetracht dessen, dass im Rahmen dieser Arbeit verschiedene Systeme gesteuert werden sollen, bietet das automatisierte Behandeln von Komponentennetzwerken große Vorteile im Vergleich zu der imperativen Behandlung von Algorithmen durch manuelle Ansätze. Anstatt manuell eine Abbildung dieser Anforderung auf ein System vorzunehmen, können Komponenten automatisch abgebildet werden. Manuelle, zumeist scriptbasierte Ansätze erfordern die Definition einer systemspezifischen Verschaltung aller vorhandenen Softwaremodule, was die Komplexität unnötig steigert. Auch müssen Veränderungen oder Fehlerbehebungen auf allen Zielplattformen manuell erfolgen. Das gefährdet die Konsistenz. Es besteht gerade bei einer, wie in der Forschung üblich, iterativen Arbeitsweise das Risiko, dass Änderungen in einem System eingepflegt, aber nicht in ein anderes System übernommen werden. Im Gegensatz dazu haben modellbasierte, automatische Ansätze den Vorteil, definieren zu Können, wie Modelle aussehen sollten bzw. was die Anforderungen sind. Die Abbildung dieser Modelle geschieht dann automatisch auf vorhandene Ressourcen der Plattformen.

### 2.4.1 Einführung in die Planung von Komponentennetzwerken

Um zwischen unterschiedlichen Plan-Managern unterscheiden zu können, muss an dieser Stelle zunächst erörtert werden, aus welchen Teilen ein typischer *Plan-Manager* besteht. Ein Plan-Manager muss, um die nötigen Komponenten zu identifizieren, die innerhalb eines Komponentennetzwerkes zu einem Zeitpunkt aktiv sein müssen, eine Suche innerhalb einer Sammlung aller Softwarekomponenten ausführen. Diese Suche wird, da sie die Interaktion zwischen verschiedenen Modulen voraussetzt, oftmals als *Planung* bezeichnet. Das Erstellen eines *Plans* kann als das Bestimmen eines Komponentennetzwerkes für einen oder mehrere Zeitpunkte gesehen werden. Sofern *Planer Pläne* erstellen können, die über

mehrere Zeitpunkte verlaufen, wird zusätzlich eine zeitliche Sequenz von Komponentennetzwerken vorgegeben. In diesem Fall sind *Planer* in der Lage, die nötigen Schritte zu bestimmen, um einen Zielzustand zu erreichen. An diesem Punkt verschmelzen *Plan-Manager* mit klassischen *Planning*-Verfahren aus dem Bereich der Künstlichen Intelligenz zur Planung von Verhalten. Eine Kombination, die bei fortgeschrittenen Systemen zum Einsatz kommt, ist die Modellierung von Problemen. Es werden *Modelle* der Algorithmen erstellt und auf eine abstrakte Weise ihr Verhalten bzw. ihr Einfluss auf die Welt beschrieben. Diese *Modelle* repräsentieren einerseits vorhandene Softwarealgorithmen, andererseits erlauben sie die Definition von abstraktem Verhalten bzw. Verhaltensprozessen. Diese Beschreibung fällt in den Bereich der *Ontologien*, da Informationen miteinander verknüpft werden, um abstrakte Beschreibungen von Verhalten, Aktionen oder Effekten vorzunehmen. Lose, abstrakte Verbunde sind ein weiteres Merkmal komplexer Repräsentationen. Durch diese Abstraktion wird die Wiederverwendbarkeit sonst spezieller Implementierungen oder Modellierungen erhöht. Ein Beispiel hierfür ist, dass eine *XSens::IMU* ein Gerät ist, das eine *Orientierung* bereitstellen kann. Wird zu einem Zeitpunkt eine *Orientierung* benötigt, ist die *XSens::IMU* eine mögliche Quelle hierfür und kann genutzt werden.

Die Modelle unterscheiden sich je nach Lösung jedoch stark voneinander. Dies hängt mit den unterschiedlichen Problemen zusammen, die mit Hilfe der Modelle dargestellt werden. Im obigen Beispiel der IMU wird eine Abstraktion konkreter Komponenten vorgenommen, da eine konkrete IMU zu einer abstrahierten Orientierung umgeformt wird. Es können aber auch abstraktere Anforderungen wie Zeit [45], Anforderungen an die Zielkonfiguration im Raum [44] oder ein Verbund aus verschiedenen Domänen [63] vorliegen. Im Rock-Framework stand das System aus Roby/Syskit bereits als integrierte und nutzbare Lösung bereit. Roby/Syskit ging aus der Arbeit von Joyeux [33] hervor und wurde stetig weiterentwickelt [34] [35], [2], [36] , [24]. Dieses System wurde entworfen, um maximale Sicherheit für die Systementwicklung zu gewährleisten. Dabei liegt sein Fokus auf der Ressourcenselektion und der Abstraktion von konkreten Algorithmen. Die Planung des Verhaltens, um einen Zielzustand zu erreichen, wird dabei nur rudimentär betrachtet. Zudem besitzt das System keine Möglichkeit der temporalen, spatialen oder symbolbasierten Planung. Die modellierbaren Missionsszenarien, die repräsentierbar sind, gehören zu den einfacheren Szenarien im Vergleich zu den oben genannten Ansätzen und bieten eine überwiegend lineare Sequenzierung von Verhalten.

Da Syskit die primären Aspekte dieser Arbeit unterstützt, wird er für die weitere Evaluation als Plan-Manager genutzt. Die Verhaltensmodellierung ist zwar weniger komplex als mit Hilfe von anderen Ansätzen gemäß dem Stand der Technik, jedoch bietet die Modellierung von Syskit eine robuste und eindeutige Modellierungssprache. Da der Fokus dieser Arbeit die Evaluation der Robustheit der aktuellen Unterwasserrobotik ist, bietet Syskit die besten Voraussetzungen für eine detaillierte Analyse der Herausforderungen.

### 2.4.2 Einführung von Syskit

Wie in der Einleitung beschrieben, liegt der Fokus der Syskit/Roby-Lösung auf der Abstraktion von systemspezifischen Softwaremodulen und der Robustheit sowie einer möglichst wenig fehleranfälligen und doch flexiblen Modellierung. Syskit besteht dabei aus verschiedenen Elementen:

- Roby
  Roby ist der Unterbau von Syskit. Das Roby-Prinzip folgt dabei dem *Event-Action*-Muster, wie es unter anderem aus [37] bekannt ist. Ein Event hat eine Aktion zur Folge. Events können dabei ignoriert oder behandelt werden. Über das Roby-Event/Action-System wird die zeitliche Sequenzierung von Verhalten vorgenommen. Roby unterscheidet dabei explizit zwischen Fehlern, also *uncontrollable*, und geplanten/erwarteten, also *controllable* Events. Auf *controllable* Events kann auf höheren Ebenen reagiert werden. Fehler (*uncontrollable*) müssen behandelt werden, da sie sonst zu einem Versagen des Netzwerkes führen würden.

- Syskit-Modellsprache
  Die Syskit-Modelle basieren auf einer Roby-DSL. Diese Modelle werden durch Syskit interpretiert und in Ruby-Objekte übersetzt. Die Modelle erlauben das Einbetten von Ruby-Code, der zur Laufzeit interpretiert wird, um eine möglichst hohe Flexibilität zu gewährleisten.

- Syskit-Modelle
  Die Syskit-Modelle sind eine Erweiterung der Roby-Modelle, um die Anwendungsspezifika eines Rock-Komponentennetzwerkes besser zu erfassen. Dabei spielen insbesondere Datenflüsse, Ausführungszeiten und funktionale Komponentenverbunde eine besondere Rolle.

- Syskit-*Planer*
  Das eigentliche Bestimmen eines Komponentennetzwerkes für eine oder mehrere zeitliche Anforderungen wird in Syskit von der *Network Resolution Engine* übernommen. Diese zu Syskit gehörige Komponente errechnet das Komponentennetzwerk für gegebene Anforderungen auf Basis der Modelle. Die Anforderungen können extern vorgegeben sein oder durch Roby-Events explizit gefordert werden.

### 2.4.3 Vorstellung der Syskit-Modelle

Um die folgenden Schritte der Arbeit verstehen zu können, erfolgt an dieser Stelle eine Übersicht über die Grundstruktur der Modellsprache, in der die Komponenten für Syskit beschrieben werden.

Ein Ziel von Syskit ist, die Abhängigkeit von konkreten Algorithmen oder auch Hardwarekomponenten in der Beschreibungssprache zu reduzieren. Aus diesem Grund werden *DataServices* genutzt. Diese DataServices ermöglichen es, Datenflüssen eine semantische Bedeutung zuzuordnen. Wie eingangs beschrieben, liefert beispielsweise eine *XSens::IMU* den DataService *Orientation*. Durch diese Abstraktion ist es möglich, funktionale Verbunde wie eine Sensorfusion zu erstellen, wie in Abbildung 2.28 gezeigt. Diese funktionalen Verbunde werden als *Kompositionen* bezeichnet.

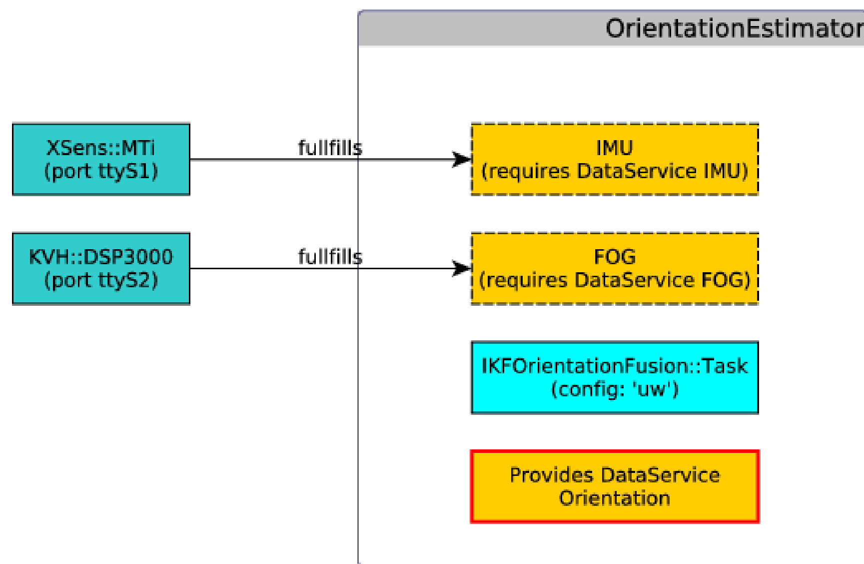



Figure 2.28: Schematische Darstellung einer Komposition zur Sensorfusion.

Wie in Abbildung 2.28 zu sehen ist, können *Compositions* oder Kompositionen wieder selbst *DataServices* bereitstellen. Durch diese Abstraktion erlaubt es die Syskit-Modellsprache, eine starke Hierarchisierung und Schachtelung von Algorithmen in der Modellsprache vorzunehmen. Diese in Abbildung 2.29 skizzierte Funktionalität gestattet die Wiederverwendung bestehender Modulverbunde in *StateMachines*, die zur Verhaltensmodellierung genutzt werden können.

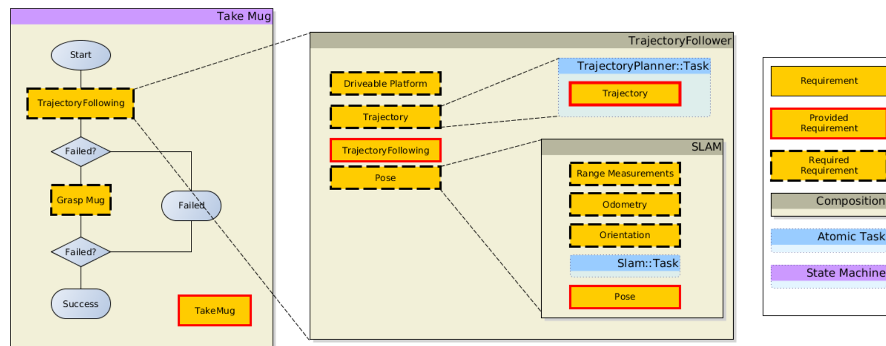



Figure 2.29: Schematische Darstellung der hierarchischen Modellierung mittels Syskit.

### 2.4.4 Vorstellung der eingesetzten Modelle

In dem folgenden Abschnitt werden die systemunabhängigen Modelle dargelegt. In den Wettkampfszenarien werden sowohl Avalon als auch Dagon als Systeme eingesetzt. Beide Systeme ähneln sich in ihrer Hardwarekonfiguration, sodass diese Systeme sich ideal eignen, um die Übertragbarkeit der Softwarealgorithmen aufzuzeigen. Die systemunabhängige Modellierung umfasst dabei Modelle, die sowohl von Avalon als auch von Dagon genutzt werden können. Hervorzuheben sind jedoch die Unterschiede zwischen den Modellen. Im Laufe dieses Abschnittes wird gezeigt, an welchen Stellen die vorhandene Modellierung von Syskit an ihre Grenzen stößt, aber auch wo die Vorteile in der Abstraktion von Komponentenverbunden liegen. In dieser Arbeit werden nur die Modelle betrachtet, die relevant sind, um die grundlegende Funktionsweise sowie Probleme der Syskitmodellierung aufzuzeigen.

#### Lagebestimmung

Im Folgenden wird die Modellbeschreibung zur Orientierungsschätzung aufgezeigt.

```
1         add_main OrientationEstimator::BaseEstimator.prefer_deployed_tasks("orientation_estimator
              ↪ "), :as => 'estimator'
```

Zunächst wird definiert, dass der Verbund vom Typ *IKFOrientationEstimatorCmp* ist. Des Weiteren wird die primäre Einheit innerhalb dieser Komposition mit dem Namen *estimator* vom Typ *OrientationEstimator::BaseEstimator* hinzugefügt. Die Besonderheit *prefer_deployed_tasks* gibt an, dass diese Komponente, sofern sie gestartet wird, den Namen *orientation_estimator* tragen soll. Diese Benennung ist für die Behandlung des Komponentennetzwerkes nicht direkt nötig, erleichtert jedoch das Identifizieren einzelner Komponenten zur Laufzeit, um eventuelle Debuggingaufgaben zu vereinfachen.

Listing 2.1: IKF Estimator

```
1         add WallOrientationCorrection::OrientationInMap, :as => 'ori_in_map'
2         add XsensImu::Task, :as => 'imu'
3         add FogKvh::Dsp3000Task, :as => 'fog'
```

In den Zeilen 3-5 werden weitere Elemente wie die IMU und ein faseroptischer Kreisel dem Verbund hinzugefügt. Diese Elemente werden von der Sensorfusion benötigt, damit der Drift der IMU im Wasser kompensiert werden kann. An dieser Stelle sei darauf hingewiesen, dass Geräte explizit verwendet werden. Hierbei wäre eine Abstraktion mittels DataServices möglich. Dies ist jedoch für die Anwendung im gegebenen Szenario nicht nötig, da beide Systeme über das gleiche Sensorsetup verfügen.

```
4         if ::CONFIG_HACK == 'default'
5             ori_in_map_child.with_conf("default", 'halle')
6             estimator_child.with_conf("default", "avalon", "halle", "Bremen")
7         elsif ::CONFIG_HACK == 'simulation'
8             ori_in_map_child.with_conf("default", 'sauce')
9             estimator_child.with_conf("default", "sauce", "Bremen")
10        elsif ::CONFIG_HACK == 'dagon'
11            ori_in_map_child.with_conf("default", 'halle')
12            estimator_child.with_conf("default", "dagon", "halle", "Bremen")
13        end
```

In den Zeilen 6-15 sind bereits Einschränkungen in der Mächtigkeit der Modelle von Syskit zu sehen. Es ist auf Modellebene nicht möglich zu definieren, dass die Konfiguration von Kindelementen abhängig vom aktuellen System ist. Zwar erlaubt es die Definition Syskit, innerhalb der systemspezifischen Konfigurationseigenschaften solche Attribute zu überschreiben bzw. zu spezifizieren, wie im folgenden Auszug gezeigt:

```
1             define 'orientation_estimator', IKFOrientationEstimatorCmp.use(
2                     'ori_in_map' => WallOrientationCorrection::OrientationInMap.with_conf("
                          ↪ default", 'sauce')
3                     'estimator' => OrientationEstimator::BaseEstimator..with_conf("default", "
                          ↪ sauce", "Bremen")
4             )
```

Jedoch gilt diese Spezialisierung nur für das neue Objekt *'orientation_estimator_def'*. Sollte die Komposition *IKFOrientationEstimatorCmp* automatisch als Bereitsteller einer

*Orientation* gewählt werden, gilt diese Spezialisierung nicht. Durch dieses *define* wird intern ein neues Modellobjekt erzeugt, das eine spezialisierte Version des Ursprungsmodells ist. Das ursprüngliche Modell bleibt neben der Spezialisierung im Pool aller Modelle unverändert bestehen.

```
14
15        imu_child.connect_to estimator_child.imu_orientation_port
16        fog_child.connect_to estimator_child.fog_samples_port
17        estimator_child.connect_to ori_in_map_child.orientation_in_world_port
18
19        export ori_in_map_child.orientation_in_map_port, :as => 'orientation_samples'
20        provides Base::OrientationSrv, :as => "orientation"
21        export estimator_child.heading_correction_port, :as => "heading_offset"
22        provides Base::OrientationToCorrectSrv, :as => "orientation_to_correct"
23
24        event :INITIAL_NORTH_SEEKING
25        event :INITIAL_ALIGNMENT
26        event :MISSING_TRANSFORMATION
27        event :NAN_ERROR
28        event :ALIGNMENT_ERROR
29        event :CONFIGURATION_ERROR
30    end
```

Der letzte Codeblock mit den Zeilen 16-32 definiert nötige Verbindungen, die erstellt werden müssen, damit die Komposition sinnvoll arbeiten kann. Zusätzlich wird definiert, dass dieser funktionale Verbund die Fähigkeiten *Base::OrientationSrv* und *Base::OrientationToCorrectSrv* bereitstellt. Um eine Sequenzierung zu ermöglichen, werden Zustände definiert, die dieser funktionale Verbund einnehmen kann. Diese Zustände werden durch das primäre Kind, das durch *add_main* in Zeile 2 hinzugefügt wurde, emittiert. Bei einer Zustandsänderung des Kindes findet somit die gleiche Zustandsänderung innerhalb dieses Verbundes statt, die im Folgenden zum Elternknoten propagiert wird.

**Anfahren von Weltkoordinaten** Das Anfahren von globalen Weltkoordinaten ist sowohl für den SAUC-E als auch für den euRathlon eine fundamentale Aufgabe. Die Systeme fahren dabei verschiedene Positionen an, um von dort aus nach Objekten zu suchen oder weitere Zielpunkte anzufahren.

```
1     class SimplePosMove < ::Base::ControlLoop
2         overload 'controller', AvalonControl::RelFakeWriter
3
4         argument :heading, :default => 0, :type => :double
5         argument :depth, :default => -6, :type => :double
6         argument :x, :default => 0, :type => :double
7         argument :y, :default => 0, :type => :double
8         argument :timeout, :default => nil, :type => :double
9         argument :finish_when_reached, :default => nil, :type => :bool  #true when it should
              ↪ success, if nil then this composition never stops based on the position
10        argument :event_on_timeout, :default => :success, :type => :string
11        argument :delta_xy, :default => DELTA_XY, :type => :double
12        argument :delta_z, :default => DELTA_Z, :type => :double
13        argument :delta_yaw, :default => DELTA_YAW, :type => :double
14        argument :delta_timeout, :default => DELTA_TIMEOUT, :type => :double
```

Obwohl die *Kompositionen* keine Algorithmen sind, die auf einem System laufen, so besitzen sie doch Attribute, die an ihre Kinder weitergegeben werden können. Des Weiteren

können in ihnen spezielle Codeblocks definiert werden, die zu bestimmten Zeitpunkten ausgeführt werden.

```
15          attr_reader :start_time
16          add ::Base::PoseSrv, :as => 'pose'
17
18          on :start do ev
19                  reader_port = nil
20                  if pose_child.has_port?('pose_samples')
21                      reader_port = pose_child.pose_samples_port
22                  else
23                      pose_child.each_child do c
24                          if c.has_port?('pose_samples')
25                              reader_port = c.pose_samples_port
26                              break
27                          end
28                      end
29                  end
30                  @reader = reader_port.reader
31                  @start_time = Time.now
32                  Robot.info "Starting Position moving #{self}"
33                  controller_child.update_config(:x => x, :heading => heading, :depth=> depth, :y
                        ↪ => y)
34                  @last_invalid_post = Time.new
35          end
```

Der Codeblock *on: start (event)* definiert Aktionen, die bei der Emittierung von Signalen ausgeführt werden sollen. Genauer handelt es sich um Funktionen, die zur Laufzeit bei Auftreten eines Events ausgeführt werden. Innerhalb des *start*-Blockes werden verschiedene Readerobjekte angelegt, die in der Lage sind, Daten auszulesen, die zwischen einzelnen Komponenten fließen. Die Zeile 33 propagiert manuell Attribute an die Kind-Einheit weiter. Eine automatische Propagation auf Modellebene ist in Syskit nicht vorgesehen, somit ist es leider nötig diesen manuellen Weg zu nutzen. Diese Codeblöcke ermöglichen somit das händische Propagieren, jedoch werden dabei keine modellbasierten Repräsentationen genutzt.

```
36          poll do
37              @last_invalid_pose = Time.new if @last_invalid_pose.nil?
38              @start_time = Time.now if @start_time.nil?
39              if @start_time.my_timeout?(self.timeout)
40                  Robot.info  "Finished Pos Mover because time is over! #{@start_time} #{
                        ↪ @start_time + self.timeout}"
41                  emit event_on_timeout
42              end
```

Der *poll*-Block wird innerhalb des Syskitslebenszyklus mit einem Intervall von ca. 0.1 Hz aufgerufen und ermöglicht die Einbettung von Code, der regelmäßig ausgeführt wird, wenn diese *Komposition* aktiv ist. Im ersten Schritt erfolgen diverse Checks, ob benötigte Objekte existieren. Falls ein Sicherheitstimeout seit dem Start abgelaufen ist, wird ein Event emittiert, damit das System auf höheren Ebenen auf diesen Fehler (konkret das Nichterreichen einer Zielposition) reagieren kann. Die Umsetzung dieses Sicherheitsverhaltens wird dabei nicht durch das Modell, sondern durch eine nutzerspezifische Implementierung durchgeführt.

```
43
44              if finish_when_reached
45                  if @reader
```

```
46                      if pos = @reader.read
47                          if
48                              pos.position[0].x_in_range(x,delta_xy) and
49                              pos.position[1].y_in_range(y,delta_xy) and
50                              pos.position[2].depth_in_range(depth,delta_z) and
51                              pos.orientation.yaw.angle_in_range(heading,delta_yaw)
52                                  @reached_position = true
53                                  if @last_invalid_pose.delta_timeout?(delta_timeout)
54                                      Robot.info "Hold Position, recalculating"
55                                      emit :success
56                                  end
```

Im weiteren Verlauf des *poll*-Blocks finden Überprüfungen statt, ob die Zielposition eines Systems erreicht wurde. Falls ja, wird das *success*-Event emittiert, um die höhere Ebene darüber zu informieren, dass die Zielposition erreicht wurde und mit dem nächsten Schritt einer Mission fortgefahren werden kann.

```
57                          else
58                              if @reached_position
59                                  Robot.info "################## Bad Pose! ################"
60                              end
61                              @last_invalid_pose = Time.new
62                              @reached_position = false
63                          end
64                      end
65                  end
66              end
67          end
68      end
```

**Sonderbehandlung des Sonars** Eines der Ziele von Syskit ist es, eine Infrastruktur bereitzustellen, um möglichst wenig fehleranfällige Systeme modellieren zu können. Deswegen sind die Syskitmodelle strikt hierarchisch angeordnet. Eine höhere Ebene kann dabei, unter Verwendung spezieller Konfigurationen, von sich selbst oder auch ihren Kindelementen Konfigurationen anfordern. Dies führt dazu, dass ein Abhängigkeitsbaum entsteht. Jeder Teilbaum ist dabei unabhängig von anderen Teilbäumen. Anforderungen in einem Teilbaum haben somit keinen Einfluss auf Anforderungen des Nachbarzweigs. Daraus ergibt sich, dass jede Konfiguration eindeutig und nicht widersprüchlich zu den Anforderungen eines anderen Astes sein muss. Benötigt ein Teilbaum ein Modell eines speziellen Gerätes, so muss der andere Teilbaum, der dieses Gerätemodell auch nutzt, die gleichen Anforderungen an das Modell stellen. Falls die Anforderungen widersprüchlich sein sollten, führt dies zu einem nicht auflösbaren Ressourcenkonflikt. Während der Erstellung der Netzwerke für die Missionen traten jedoch genau diese, auf den ersten Blick widersprüchlichen, Anforderungen an ein Gerätemodell auf.

Die folgenden Modelle 2.2,2.3 skizzieren dabei den Spezialfall, in dem widersprüchliche Anforderungen an das Gerät bestehen. Anhand der Modelle wird erläutert, warum diese zunächst widersprüchlich erscheinenden Anforderungen dennoch korrekt sind.

Listing 2.2: Localization Model

```
1  module Localization
2      class ParticleDetector < Syskit::Composition
3          add UwParticleLocalization::Task, :as => 'main'
4          add Base::SonarScanProviderSrv, :as => 'sonar'
5
           ...
16         if ::CONFIG_HACK == 'default'
17             main_child.with_conf("nurc", "slam_testhalle")
18             sonar_conf = ['default']
19         elsif ::CONFIG_HACK == 'simulation'
20             main_child.with_conf("sim_nurc", 'slam_testhalle')
21             sonar_conf = ['default']
22         elsif ::CONFIG_HACK == 'dagon'
23             main_child.with_conf("nurc_dagon", 'slam_testhalle')
24             sonar_conf = ['default']
25         end
```

Die Lokalisierung benötigt ein Sonar in der Konfiguration mit einem 360°Scanradius. Mit dieser Einstellung kann die Lokalisierung optimal die Position des Fahrzeugs in einer Umgebung bestimmen. Diese Konfigurationsanforderung wird an die jeweilige Sonarinstanz weitergegeben.

Listing 2.3: Wall Dector Model

```
1  module Wall
2      class Detector < Syskit::Composition
3
4          add_main WallServoing::SingleSonarServoing, :as => 'detector'
5          if ::CONFIG_HACK == 'simulation'
6              add Base::SonarScanProviderSrv, :as => 'sonar'
7          else
8              add SonarTritech::Micron, :as => 'sonar'
9          end
10
           ...
24
25
26         conf 'wall_front_left',
27              sonar_child => ['default', 'wall_front'],
28              detector_child => ['default', 'wall_front_left']
29         conf 'wall_front_right',
30              sonar_child => ['default', 'wall_front'],
31              detector_child => ['default', 'wall_front_right']
32         conf 'wall_right',
33              sonar_child => ['default', 'wall_right'],
34              detector_child => ['default', 'wall_right']
35         conf 'wall_left',
36              sonar_child => ['default', 'wall_left'],
37              detector_child => ['default', 'wall_left']
38         conf 'hold_wall_right',
39              sonar_child => ['default', 'wall_right'],
40              detector_child => ['default', 'hold_wall_right']
```

Die Wandverfolgung hat als Zielsetzung, möglichst dicht an einer Wand entlangzufahren. Hierbei ist weniger die globale Position im Becken als die Distanz zu einer Wand relevant. Da die Wand in einem geringen Abstand abgefahren werden soll, ist eine hohe Aktualisierungsgeschwindigkeit der Distanz zur Wand somit von hoher Bedeutung, um bei Wellen oder Regelungsfehlern nicht mit der Wand zu kollidieren. Da es sich bei dem eingesetzten Sonar um ein Scanning Sonar handelt, ist die Aktualisierungsgeschwindigkeit direkt von der Größe des Scanwinkels abhängig, die aktuell konfiguriert ist. Die Anforderungen der Lokalisierung und der Wandverfolgung stehen somit konkurrierend zueinander.

Während jedoch die Lokalisierung zeitweise auch mit einem eingeschränkten Scanwinkel, wie er von der Wandverfolgung gewünscht ist, arbeiten kann, stellt dieser eingeschränkte Scanwinkel eine schwierige Anforderung für die Wandverfolgung dar. Die Syskit-Modelle bieten keine Möglichkeit der direkten Modellierung für das Problem, sie erlauben jedoch die Registrierung von Codeblöcken. Mit diesen Codeblöcken ist es möglich, das Verhalten von Syskit direkt zu beeinflussen. Die Sonarkonfiguration wird aus diesem Grund nicht auf Modellebene, sondern in einem speziell eingerichteten Handler durchgeführt.

```
Roby.every(1, :on_error => :disable) do
    wall = tryGetTask("wall_servoing")
    localization = tryGetTask("uw_particle_localization")
    sonar = tryGetTask("sonar")
    buoy_on_wall = tryGetTask("buoy_on_wall")

    if(wall.running?)
        if buoy_on_wall.running?
            sonar_conf = ['default']
        else
            sonar_conf = ['default','wall_right']
        end
    else
        sonar_conf = ['default']
    end
```

In diesem mit 0.1 Hz aufgerufenen Handler wird versucht, die aktuellen Algorithmeninstanzen zur Laufzeit zu akquirieren. Sofern dies erfolgreich war, wird die aktuell benötigte Konfiguration des Sonars bestimmt.

```

    begin
        if(sonar.running?)
            if sonar_conf != State.current_sonar_conf
                ::Robot.info "Reconfiguring sonar to: #{sonar_conf}"
                sonar.apply_conf(sonar_conf,true)
                State.current_sonar_conf = sonar_conf
            end
```

Falls das Sonar aktiv ist, wird überprüft, ob die Konfiguration der aktuell benötigten entspricht. Sofern dies nicht der Fall sein sollte, wird die Konfiguration zum Gerät gesendet.

```
            if CONFIG_HACK != 'simulation'
                #Sainity check
                #if sonar_conf.include?('wall_left')
                if sonar_conf.include?('wall_right')
                    if sonar.config.continous == true
                        ::Robot.warn "Sonar seems to be configured invalid even hack is active, reforcing"
                        State.current_sonar_conf = nil #Try to configure again
                    end
                else
                    if sonar.config.continous == false
                        ::Robot.warn "Sonar seems to be configured invalid even hack is active, reforcing"
                        State.current_sonar_conf = nil #Try to configure again
                    end
                end
            end
        end
    rescue Exception => e
        ::Robot.warn "Somethig happening during application of our sonar hack"
        ::Robot.warn e
    end
end
```

Da das Setzen der Konfiguration sich in den ersten Versuchen als nicht immer zuverlässig herausgestellt hat, wurde eine Sicherheitsüberprüfung eingeführt. Diese kontrolliert, ob die aktuell geschriebene Konfiguration der zum Sonar gesendeten entspricht. Falls hierbei eine Abweichung festgestellt wird, wird die Konfiguration erneut gesendet.

**Verhaltenssequenzierung** Die bisher eingeführten Modelle beschreiben funktionale Teilverbunde, die das Verhalten zu einem Zeitpunkt des Systems determinieren. Um jedoch einen Verhaltenswechsel zu ermöglichen, muss eine Sequenzierung verschiedener Verbunde durchgeführt werden. Hierfür wird in Syskit ebenfalls das Event/Action-Modell genutzt. Ein Event, beispielsweise *end_ of_ pipe* wird durch einen Piplinedetektor emittiert, Syskit empfängt dieses und ändert daraufhin die Anforderungen an das Netzwerk. Ein Folgezustand könnte beispielsweise sein, eine neue Weltposition anzufahren. Um das Systemverhalten so steuern zu können, wurden endliche Automaten (auch StateMachines, FSMs) in Syskit eingeführt. Diese erweitern und abstrahieren das Event/Action-basierte Interface um eine formale, modellbasierte Syntax.

```
1  describe("Do the minimal demo once")
2  state_machine "minimal_demo_once" do
3      init = state simple_move_def(:finish_when_reached => true, :heading => 0, :depth => -4, :
           ↪ delta_timeout => 5, :timeout => 15)
4
5      s1 = state find_pipe_with_localization
6      pipeline1 = state pipeline_def(:depth => -5.5, :heading => 0, :speed_x => 0.5, :turn_dir
           ↪ => 1, :timeout => 180)
7      align_to_wall = state simple_move_def(:finish_when_reached => true, :heading => 3.14, :
           ↪ depth => -5, :delta_timeout => 5, :timeout => 15)
8      rescue_move = state target_move_def(:finish_when_reached => true, :heading => Math::PI, :
           ↪ depth => -5, :delta_timeout => 5, :x => 0.5, :y => 5.5)
9      wall1 = state wall_right_def(:max_corners => 1)
10     wall2 = state wall_right_def(:timeout => 20)
11
12     surface = state simple_move_def(:finish_when_reached => true, :heading => 0, :depth => 1,
           ↪  :speed_x => 0.1, :delta_timeout => 5, :timeout => 30)
13
14     start(init)
15     transition(init.success_event, s1)
16     transition(s1.success_event, pipeline1)
17     transition(s1.failed_event, rescue_move)
18     transition(pipeline1.end_of_pipe_event, align_to_wall)
19     transition(pipeline1.success_event, align_to_wall)
20     transition(pipeline1.lost_pipe_event, rescue_move)
21     transition(rescue_move.success_event, wall1)
22     transition(align_to_wall.success_event, wall1)
23
24     transition(wall1.success_event, wall2)
25     transition(wall2.success_event, surface)
26     forward surface.success_event, success_event
27 end
```

Wie in den Zeilen 3-12 zu sehen ist, bestehen die Zustände der StateMachine aus Anforderungen an das Netzwerk, die parametrisiert werden können. Es ist zu beachten, dass sämtliche Parameter zur Laufzeit an die Kompositionen übergeben werden. Die Transitionen definierten Konditionen zu Zustandsübergängen. Die letzte Zeile 26 des Codeblocks definiert das Ausgangsverhalten der StateMachine selbst. In diesem Fall führt nur das *surface.success_ event* zum Erfolg der StateMachine *minimal_ demo_ once*. Jede andere

Zustandsänderung würde zu einem Fehler der StateMachine führen und müsste von einer höheren Schicht behandelt werden. Findet diese Behandlung nicht statt, resultiert dies in einem Abbruch der Gesamtmission.

```
1      describe("Find_pipe_with_localization").
2          optional_arg("check_pipe_angle",:bool ,false)
3      action_state_machine "find_pipe_with_localization" do
4          find_pipe_back = state target_move_def(:finish_when_reached => false ,
5          :heading => 1 + POOL_ALIGNMENT, :depth => -6, :x => -6.5, :y => --0.5, :timeout => 180)
6          pipe_detector = state pipeline_detector_def
7          pipe_detector.depends_on find_pipe_back, :role => "detector"
8          start(pipe_detector)
9
10  #         pipe_detector.monitor(
11  #             'angle_checker', #the Name
12  #             pipe_detector.find_port('pipeline'), #the port for the reader
13  #             :check_pipe_angle => check_pipe_angle). #arguments
14  #             trigger_on do pipeline
15  #                 angle_in_range = true
16  #                 if check_pipe_angle
17  #                     angle_in_range = pipeline.angle < 0.1 && pipeline.angle > -0.1
18  #                 end
19  #                 state_valid = pipeline.inspection_state == :ALIGN_AUV  pipeline.inspection_state
       ↪  == :FOLLOW_PIPE
20  #                 state_valid && angle_in_range #last condition
21  #             end. emit pipe_detector.success_event
22          forward pipe_detector.align_auv_event, success_event
23          forward pipe_detector.follow_pipe_event, success_event
24
25          forward pipe_detector.success_event, success_event
26          forward pipe_detector,find_pipe_back.success_event,failed_event #timeout here on moving
27      end
```

Die Besonderheit dieser StateMachine ist die Abhängigkeit zwischen dem Pipelinedetektor und der Lokalisierung. Zwar steuert die Lokalisierung aktiv das System an eine Zielposition, jedoch wird durch Zeile 7 explizit definiert, dass der Detektor das eventemittierende Modul des Zustands ist. Somit kann ein aktiver Zustand aus zwei Untermodulen bestehen, wobei nur ein Modul als eventemittierende Instanz dienen kann. Dies führt im mathematischen Sinn zu parallelen StateMachines, wobei die Eigenschaften der Parallelität konzeptuell von Syskit/Roby vernachlässigt werden.

Darüber hinaus können weitere Bedingungen an die StateMachine-Transitionen geknüpft werden. Somit ist es, wie in dem Beispiel gezeigt, möglich, nur dann die Transition durchzuführen, wenn der Pipelinewinkel in einem definierten Bereich liegt und die Pipeline zuverlässig erkannt wurde. Diese Überprüfung sollte sicherstellen, dass keine falschen Objekte als Pipeline erkannt werden. Dieser Sonderfall wird in Abschnitt 2.5.3 detaillierter erläutert.

## 2.5 Erkenntnisse während der Evaluationen

In diesem Abschnitt werden sowohl die Erfahrungen mit den Hardwaresystemen als auch mit den eingesetzten Softwarekomponenten sowie dem Plan-Manager beschrieben. Avalon und Dagon haben jeweils verschiedene konzeptuelle Vor- und Nachteile. Die Software

sowie das Management des Komponentennetzwerkes wird durch die konzeptuell gleichen Lösungen und Modelle realisiert. Im Folgenden werden die Probleme sowie Vor- und Nachteile des gewählten Lösungsweges erläutert. Im Anschluss folgt ein Ausblick auf den Hauptteil 3 dieser Abhandlung. Dieser widmet sich der Verbesserung der Behandlung der Komponentennetzwerke.

### 2.5.1 Erkenntnisse über die Hardware

Der folgende Abschnitt widmet sich den Erkenntnissen, die während des Umgangs mit den Systemen Avalon und Dagon aus den verschiedenen Anwendungsszenarien gewonnen wurden.

#### Spannungsversorgung

Die Konzepte von Avalon und Dagon unterscheiden sich, wie im Abschnitt 2.2.2.1 und 2.2.1.3 dargestellt, stark voneinander. Bedingt durch den proprietären Aufbau des Dagon-Batteriesystems war es innerhalb von einer Woche dreimal nicht möglich, das System zu starten. Es ließ sich weder einschalten noch reagierte es auf das extern angeschlossene Diagnosesystem des Herstellers. Aus nicht ersichtlichen Gründen funktionierte das System am Abend wieder. Das Avalon-System erwies sich hingegen als zuverlässig. Vorteile hatte das Dagon-System jedoch bei den Ladezyklen. Dagon ließ sich schneller laden als Avalon. Dagon benötigte ca. vier Stunden für eine volle Ladung, während die Ladedauer von Avalon ca. acht Stunden betrug. Dies ist der Tatsache geschuldet, dass das ausgleichen der Akkuladungen von Avalon ein stark limitierender Faktor ist. Sowohl das Ladegerät als auch die Balancingeinheit reizen die Kapazitäten des Batteriepacks nicht aus. Dagon hatte weiterhin den Nachteil, dass das Batterie Management System nicht über den PC ausgelesen werden konnte. Es ist unbekannt, ob dies an mangelnder Verkabelung/Integration lag oder ob dies entgegen den vorliegenden Informationen grundsätzlich nicht möglich ist. Dadurch musste der Batteriezustand während der Operation im Wasser vollkommen geschätzt werden. Dies führte zwar zu keinen Nachteilen während der Wettbewerbe, ist jedoch ein Hindernis, um langfristige vollautonome Missionen durchführen zu können.

#### Kritikalität der Konnektoren

Avalon sowie Dagon verwenden Konnektoren der Firma Teledyne-Impulse. Jedoch nutzt Dagon eine andere Serie als Avalon, die bei Dagon verwendeten Verbinder sind nicht feucht oder unter Wasser steckbar. Dadurch muss Dagon für jeden Steckvorgang aus dem Wasser gehoben werden, um eine Rekonfiguration vorzunehmen oder ein Kabel anzuschließen. Dies

wiederum führt zu einem entscheidenden Zeitverlust während des Wettbewerbes. Zwar besitzt Dagon ebenso wie Avalon einen WLAN-Accesspoint zur externen Kommunikation, dieser wird jedoch über den gleichen Anschluss betrieben wie ein alternatives Kupferkabel. Somit muss vor einem Missionsbeginn entschieden werden, welcher Kommunikationskanal genutzt wird. Da es jedoch selbst bei kabelgebundenem Betrieb für die Wettbewerbe nötig ist, die Kabelverbindung nach dem Start einer Mission zu entfernen, wurde ein Adapterkabel gefertigt , das den Dagonstecker auf eine Avalon-kompatible Buchse umsetzt. Damit wurde zumindest das Problem der feucht steckbaren Kupfer-Netzwerkverbindung umgangen.

### Dagons WLAN

Zum Planungsbeginn des Wettbewerbes sollte Dagon, nach Aussage des Systemverantwortlichen, zwar über WLAN und GPS verfügen, jedoch stellte sich eine Woche vor Abfahrt heraus, dass beides nicht funktionsfähig ist. Das System wurde daraufhin repariert, wie in Abbildung 2.24 zu sehen ist. Dazu musste jedoch der gleiche Netzwerkport an Dagon verendet werden, der sonst für das Kupferkabel in Verwendung ist. Dieser nicht feucht steckbare Verbinder verhinderte somit einen schnellen Austausch bzw. Wechsel von WiFi auf Kupfer. Die Box verfügte bereits über einen Konnektor der *Bucaneer 400 series*. Bei diesem Steckertyp handelt es sich um wasserdichte Steckverbinder, die jedoch lediglich IP68-zertifiziert sind. Es sind somit keine dedizierten Unterwasser-Verbinder. Diese Steckverbinder führten zusammen mit der Konzeption der Box dazu, dass einen Tag, bevor die Langstreckennavigationsaufgabe mit Dagon absolviert werden sollte, die komplette WLAN- und GPS-Box voll Wasser lief und beides nicht mehr nutzbar war. Speziell war die Verschraubung der Steckverbinder an der Box nicht gesichert, die Vibrationen des Fahrzeugs lösten die Verschraubung, wodurch Wasser in die Box eintreten konnte, das somit sowohl das GPS- als auch das WLAN-Modul völlig zerstörte.

### Avalons WLAN und GPS

Im Gegensatz zu Dagon wurden bei Avalon die externen Komponenten komplett druckneutral vergossen. Abbildung 2.15 zeigt das eingegossene GPS-Modul. Aufgrund des Defektes des Dagon-Moduls wurde in der Nacht vor der Langstreckennavigation noch ein Adapter Gefertigt, mit dem das Avalon-Modul an Dagon verwendet werden konnte. Dies war jedoch nur für das GPS möglich. Es standen für das WLAN kurzfristig nicht genug Reservekonnektoren zur Verfügung. Die für Avalon druckneutral vergossenen Komponenten erwiesen sich während aller Missionen als zuverlässig. Dies hatte verschiedene Gründe. Einerseits wurden die Hohlräume der für Avalon verwendeten Bucaneer-400-Steckverbinder mit

Vergussmasse aufgefüllt, wodurch kein Wasser bei Defekten der Kabelisolierung über die Steckverbinder in die Module eindringen kann. Des Weiteren wurden auch die Module des GPS und WLAN selbst komplett druckneutral vergossen. Dieses Vorgehen führte zu einer hohen Robustheit. Gerade bei Systemen, die häufig genutzt werden, können sich keine Verbinder mehr lösen und somit kann anders als bei Dagon kein Wasser eindringen.

Die WLAN-Verbindung selbst hat sich, wie erwartet, als relativ unzuverlässig herausgestellt. Trotz hoher Antennen von Avalon war es nicht immer möglich, eine stabile WLAN-Verbindung über längere Strecken aufzubauen. Es war oftmals nötig, Avalon näher an die Basisstation zu fahren, um robust zu kommunizieren. Es ist unklar, ob dies durch Reflexionen der Wasseroberfläche, eventuell vorhandene Störsender auf dem Gelände oder mangelhafte Antennen verursacht wurde. Trockentests mit dem identischen Setup ergaben stets eine WLAN-typische Reichweite von ca. 50 Metern.

#### Steuerung der Systeme

Avalon verfügt über eine fehlersichere Notfallsteuerung. Im Gegensatz dazu besitzt Dagon, wie in 2.2.2.4 beschrieben, eine Steuerung, die nur über den PC erfolgen kann. Das ausfallsichere Mehrkanal-Steuerungskonzept von Avalon hat sich auf den Wettbewerben als immens robust und nützlich herausgestellt. Durch das Modem konnte jederzeit eine Nachricht zum Auftauchen an Avalon gesendet werden. Sobald das System an der Oberfläche war, konnte es mittels des 867-MHz-Protokolls an die Küste gesteuert werden. Hier konnte die nützliche WLAN-Verbindung etabliert werden. Aus diesen Gründen hat sich die Dreikanalkommunikation als von enormer Bedeutung herausgestellt. Durch den Zeitgewinn, der für die Rückgewinnung der Kontrolle über das Fahrzeug realisiert wurde, war es möglich, schnell Missionsadaptionen vorzunehmen.

Dagon zu steuern war nach dem Defekt der WLAN-Verbindung nur mit großen Mühen möglich. Eine Kommunikation über einen Funkkanal stand nicht mehr zur Verfügung. Da die Mission ohne Kabel nicht erlaubt war, musste über ein durch die Organisatoren ins Wasser gebrachtes Schlauchboot mittels des gefertigten Adaptersteckers Dagon an ein Laptop angebunden werden, um hierüber die Startkommandos zu erteilen oder Rekonfigurationen vorzunehmen. Eine eventuelle Fehlersuche war vom Schlauchboot aus nicht praktikabel. Es ist für eine Person nicht möglich, auf einem schwankenden Schlauchboot das System festzuhalten und gleichzeitig Debuggingarbeiten am Laptop auszuführen.

#### Wartungsaufwand der Plattformen

Bedingt durch den Ausfall des WLAN- und GPS-Moduls musste Dagon während des Wettbewerbes zwecks Neuverkabelungen leider geöffnet werden. Das Öffnen des Systems gestaltet sich im Gegensatz zum Avalon-System als leichter, da die Verschlussschnallen am System fest verschweißt sind. Dies ermöglicht ein Aufhebeln der Druckhüllen. Das Verschließen des Systems ist hingegen nur mit großer Vorsicht möglich. Beim Einführen besteht die Gefahr, Kabel abzuscheren, die zwischen der linken und rechten Hälfte verlaufen. Auch besitzt Dagon feste Verbinder zwischen der Schublade und dem Innenleben, die vorsichtig manuell gezogen/gesteckt werden müssen, was die Fehlerwahrscheinlichkeit erhöht, falls Stecker übersehen werden.

Bei Avalon hingegen befinden sich eine elektrische Verbindung zwischen Schublade und Hülle lediglich in der hinteren Hälfte. Diese Verbindung wird mittels eines Steckers automatisch hergestellt, es existieren keine überhängenden Kabel. Ein Abscheren von Kabeln ist somit nicht möglich. An dieser Stelle ist jedoch der Fehler aufgetretenen, dass sich durch den Transport die Kabel aus dem Steckverbinder selbst gelöst haben. Dies wurde dadurch begünstigt, dass die falschen Querschnitte verwendet wurden.

### 2.5.2 Erkenntnisse über die Software

Die Softwarekomponenten arbeiteten überwiegend wie erwartet. Es traten innerhalb der einzelnen Module nur vernachlässigbare Probleme auf, die für diese Arbeit von keiner Bedeutung sind. Daher beschäftigt sich dieser Teil der Arbeit lediglich mit den Modulen, die Probleme verursachten. Im Speziellen waren das die Reglerkette bzw. ihre Integration in die Systeme sowie die Behandlung der Komponentennetzwerke zur Verhaltensmodellierung.

#### Reglerkette

Während der Benutzung der Reglerkette traten zwei unterschiedliche Probleme auf. Das erste Problem war, dass die Reglerkette auf PID-Reglern basiert, um das System zu steuern. Für die Regelung von Beschleunigungen, Geschwindigkeiten und Positionen wird bei Avalon hingegen ein Bewegungsmodell (gestützt durch das Sonar) genutzt, um diese Informationen zu schätzen. Es stellte sich während der Wettbewerbsvorbereitungen heraus, dass sich diese beiden Regler gegeneinander aufschwingen können. Durch die langsame Aktualisierungsrate des Sonars in Verbindung mit dem Bewegungsmodell, das immer nur eine Annäherung der Bewegung approximieren kann, kam es vor, dass Avalon sich vorwärts bewegte, obwohl das Modell noch eine Rückwärtsfahrt errechnet hatte. Die PID-Regler

regelten dementsprechend dagegen an und verstärkten die rückwärtige Bewegung. Die Regelung auf geschätzten Geschwindigkeiten, basierend nur auf einem Bewegungsmodell, stellte sich somit als zu ungenau heraus, als dass sie in einem realen System eingesetzt werden könnte. Daher wurde auf Avalon während der Wettbewerbe die alte, vereinfachte Reglerkette eingesetzt.

Auf Dagaon hingegen kam die neue Reglerkette zum Einsatz. Da Dagon zusätzlich über ein DVL verfügt, war das erste Problem nicht gegeben, da eine ausreichend schnelle und genaue Aktualisierung der Geschwindigkeiten und Beschleunigung mittels des Gerätes erfolgte. Durch den Ausfall des GPS-Moduls wurde jedoch die Konfiguration der Eingabedatenströme in die Filterkomponente geändert. Da Rock in einem komplexen System zur Zeitbestimmung den sogenannten StreamAligner [11], basierend auf der Arbeit von Willenbrock [69], nutzt, ergaben sich unvorhergesehene Probleme. Die Integration im Framework sammelt die Datensamples bis zu einem maximalen Zeitpunkt. Durch die Änderung der Eingabeströme ergab sich aber eine leere Eingabe, weshalb alle Daten der verarbeiteten Module zu lange gesammelt wurden, da keine GPS-Daten vorlagen [2].

Während des Wettbewerbes führten die Fehler jedoch dazu, dass Dagon im Finallauf wild zu schwingen begann und das System keinen stabilen Zustand erreichen konnte, da es auf veralteten Daten regelte. Die komplette Aufgabe wurde somit während des Wertungslaufs erfolglos abgebrochen.

Die Information über vorhandene Quellen der Filterkomponente war nicht Bestandteil der Softwaremodelle, sondern wurde explizit als Konfiguration enkodiert. Auch bestand innerhalb des gesamten Frameworks keine Introspektionsmöglichkeit in Bezug auf das Gesamtverhalten. Hier lagen somit mehrere Schwachstellen in der Umsetzung der modellgetriebenen Softwareentwicklung und des Systemdesigns vor.

Es hätte auch erkannt werden müssen, dass die Samples, die am Ende der Verarbeitung auf die Motoren geschrieben wurden, veraltet waren. Dieses hätte zu einem Fehler führen müssen. Das Fehlverhalten lag jedoch hauptsächlich an der fehlerhaften Umsetzung der Dagon-Treiber, auch wenn es durch das Kommunikationsframework hätte erkannt werden können.

Andererseits hätte die Verifikation der Modelle erkennen müssen, dass ein Eingabedatenstrom (hier das GPS) nicht vorhanden ist. In der Folge hätte die Konfiguration des StreamAligners angepasst werden müssen, um den nicht vorhandenen Datenstrom zu ignorieren. Dieser Fehler hätte auf einer konsistent umgesetzten Modellebene erkannt werden können, noch bevor die eigentliche Mission gestartet worden wäre.

[2] Der Fall, dass kein GPS-Lock vorliegt, wurde jedoch berücksichtigt.

### 2.5.3 Wartbarkeit der Komponentennetzwerke

Die Erstellung der Komponentennetzwerke für beide Systeme hat sich als schwieriger herausgestellt als erwartet. Es traten verschiedene Probleme auf. Die Probleme waren teilweise konzeptioneller Natur, andere waren durch Implementierungsfehler hervorgerufene Bugs.

Hervorzuheben sind dabei beispielsweise Probleme mit Konfigurationsanforderungen. Wie in dem Codebeispiel 2.4.4 gezeigt, ist es nicht Möglich, über die vorhandene Modellierungsebene globale Konfigurationswünsche für spezialisierte Kompositionen zu definieren. Dieses führte dazu, dass die Modellebene umgangen wurde. In Abschnitt 2.4.4 wird erklärt, wie dazu eine globale Ruby-Variable *CONFIG_HACK* genutzt wird. Es ist offensichtlich, dass durch die explizite Einführung von Zielsystemen diese Modelle nicht systemunabhängig modelliert werden können. Es wäre hierfür nötig, die Verwendung in Abhängigkeit von einer Instanziierung explizit einzuschränken. Da zunächst der im Codebeispiel 2.4.4 beschriebene Weg befolgt wurde, führte dies in manchen Netzwerkinstanziierungen dazu, dass die Konfigurationen nicht korrekt gesetzt wurden, wenn kein virtuelles Objekt *orientation_estimator_def* genutzt wurde. Dies resultierte in der Folge in einem Fehlverhalten der kompletten Systeme, da die Filter mit ungültigen Einstellungen betrieben wurden.

Ein anderes Problem war die fehlende Möglichkeit der Simulation bzw. statischen Analyse des Verhaltens der Komponentennetzwerke sowie des Gesamtsystems. Zwar existiert ein (physikalisches) Simulationssystem der kompletten Softwaremodule, jedoch ermöglicht es nur ein Abspielen der Mission in Echtzeit und ist primär für die Überprüfung der Detektoren und Regelung ausgelegt. Eine statische Analyse der Modelle hingegen ist nicht vorhanden. Da Roby ein eventgetriebenes System ist, werden sämtliche Zyklen erst bei Auftreten konkreter Events durchgeführt. Es besteht bisher keine Möglichkeit, durch eine Simulation das Verhalten der Module zu analysieren. Aus diesem Grund kann zu Analysezwecken das Netzwerk immer nur für einen definierten Zustand betrachtet werden. Es ist beispielsweise möglich, das Netzwerk für *orientation_estimator_def* zu analysieren. Im Gegensatz dazu erlaubt es Roby aber nicht, zeitliche Sequenzen zu analysieren, wie sie für die StateMachines aus Abschnitt 2.4.4 nötig wären.

Der Codeauszug 2.4.4 hat während der Anwendung zwei weitere Probleme aufgezeigt. Wenn in StateMachines die *A.depends_on(B)*-Direktive benutzt wird, führt dies dazu, dass die abhängige Direktive *B* nicht mehr beendet wird, solange die übergeordnete StateMachine selbst aktiv ist. Dabei handelt es sich zwar um einen Implementierungsfehler von Roby/Syskit, jedoch ist er schwer nachvollziehbar, da durch fehlende Simulationsmöglichkeiten dieses Verhalten erst zur Laufzeit auftritt. Das andere Problem, das in diesem Codeauszug auftrat, ist der eingeführte *Monitor*. Dieser funktionierte leider nur nichtdeterministisch. In

der Folge führte dies dazu, dass die Mission trotz dieser neu eingeführten Sicherheitsprüfung während der Ausführung manchmal hängenblieb oder fälschlicherweise die falsche Pipeline erkannte. Neben diesem immer wieder auftretenden Fehler gab es einzelne Fehlerevents, die jedoch während der Wettbewerbe aus Zeitgründen nicht protokolliert wurden. Insgesamt erwies sich der eingesetzte Plan-Manager Roby/Syskit als nicht immer deterministisch.

### 2.5.4 Erfahrungen während der Systemeinsätze

#### SAUC-E

Während des SAUC-E traten die bereits angeführten Probleme der widersprüchlichen Sonarkonfiguration auf. Dieses führte zu dem Umbau der modellbasierten Definition der Sonarbehandlung. Die Modellebene wurde, wie in 2.4.4 beschrieben, ignoriert und durch ein codebasiertes Verfahren ersetzt. Des Weiteren erwies sich der Roby-Layer als relativ langsam. Roby berechnet den Folgezustand reaktiv, das bedeutet, bei Auftreten des korrespondierenden Events. Dies führt, je nach Komplexität des Folgezustandes, zu einer deutlich spürbaren Verzögerung. In der Teilmission der Bojenerkennung fährt das System so lange vorwärts, bis eine Boje erkannt wird, und ändert daraufhin das Netzwerk. Diese Neuberechnung dauerte jedoch in der Praxis ein bis drei Sekunden, was in der Folge dazu führte, dass das System unter Umständen schon an der Boje vorbeigefahren war. Da eine Beschleunigung von Roby nicht möglich war, wurde das Problem umgangen, indem ein Zwischenzustand eingeführt wurde. Dieser Zwischenzustand lässt das AUV für wenige Sekunden rückwärtsfahren, um in etwa an der Position zu sein, an der die Boje erkannt wurde.

#### euRathlon

Die Portierung der Komponenten auf das Dagon-Fahrzeug erforderte, trotz modellbasierter Entwicklung, mehr Zeit als ursprünglich geplant. Dies lag in der Tatsache begründet, dass selbst die Treiberkomponenten von Dagon in einem desolaten Zustand waren. Manche Komponenten waren derart nichtdeterministisch programmiert, dass sie regelmäßig abstürzten oder zehnmal gestartet werden mussten, bevor sie ihre Operation aufnahmen. Hinzu kamen die Probleme mit dem WLAN- und GPS-Modul, die zuvor angesprochen wurden. Die Portierung der Softwaremodule brachte die Probleme der Konfigurationen zutage, die in Abschnitt 2.4.4 erläutert wurden. Der bestehende Softwarestack von Dagon nutzte darüber hinaus keinerlei korrekte Synchronisation der eigentlichen Sensordatenflüsse. Durch die Portierung der Reglerkette, Lokalisierung und Bewegungsschätzung entstand das Problem, dass die Reglerkette zu schwingen begann, wie in 2.5.2 beschrieben.

Zusammen mit den Problemen des nicht funktionalen GPS/WLAN-Moduls, das in 2.5.1 beschrieben wurde, führte dies dazu, dass die Langstreckennavigationsaufgabe nicht erfolgreich absolviert werden konnte. Das System begann sich aufzuschwingen und war nicht operierbar. Es bestand durch die Rekonfiguration ohne GPS keine Zeit mehr, die Reglerkette ausführlich zu testen.

Der Softwarestack auf dem Avalon-System führte jedoch auch zu Problemen. Die eingeführten Code-Poll-Blöcke 2.4.4 (siehe Zeilen 36-67) führten zu einer Schwierigkeit. Diese Code-Blöcke werden zur Laufzeit ausgewertet, sind also wie ein reguläres Programm anzusehen. Während der Wettbewerbe wurde an diesen Stellen eine ungültige Anweisung wie *Robot.puts #{invalid_variable}* genutzt. Diese durch einen Flüchtigkeitsfehler eingeführte ungültige Variable führte zu einem Fehler innerhalb der Ausführung während einer Mission. Durch eine hier nicht relevante Verkettung von Programmabläufen führte dies dazu, dass Roby für vier Minuten blockierte und daraufhin die Mission mit einem Fehler abbrach. In dieser Zeit verblieb Avalon an der Zielposition, da die einzelnen unterlegenen Tasks weiterhin ausgeführt wurden. Das letzte Komponentennetzwerk behielt seine Funktion bei. Lediglich der Kontrollzyklus des Plan-Managers war blockiert. In der Folge erkannte Roby nach vier Minuten die ungültige Variable (die zu Diagnosezwecken genutzt wurde) und brach die Gesamtmission ab.

#### Demonstrationsmission

Die Demonstrationsmission hatte, im Gegensatz zu den Wettkampfszenarien, das Ziel, das System möglichst lange in einer unendlichen Schleife von Zyklen zu halten. Dabei galt es eventuelle Detektor- oder Regelungsfehler zu erkennen, um ein Ausweichverhalten (in der Regel das Anfahren einer globalen Zielposition) zu realisieren. Ein großes Problem stellte dabei, neben den schon genannten Problemen, der Plan-Manager selbst dar. Die bereits erwähnten konzeptionellen Probleme in Syskit erlaubten keine Überprüfung der Missionsmodellierung auf sämtliche Fehlerzustände hin. Aus diesem Grund war es Nötig, durch iteratives Starten und Analysieren der auftretenden Fehler das Verhalten zu analysieren und zu verbessern. Ein weiteres Problem war, dass die Software augenscheinlich mit jeder Netzwerkiteration mehr Zeit benötigte, um den Folgezustand eines aktuellen Netzwerkes planen zu können. Eine Rekonfiguration dauerte im Extremfall mehrere Minuten. Dennoch wurde die Demonstrationsmission erfolgreich durchgeführt. Das System lief über mehrere Stunden in dem im Missionszenario skizzierten Ablauf. Das einzige Problem war, dass durch die Fehler der zu langsamen Umschaltzeit zwischen *Zielkoordinaten anfahren* und *Pipelineverfolgungsverhalten starten* die Pipeline öfter verloren wurde, sodass das Sicherheitsausweichverhalten *zur Wand zu fahren* ausgelöst wurde. Erschwerend kam hinzu, dass das System weiterhin durch die fehlerhafte Monitorimplementierung aus 2.4.4 des Öfteren

fälschlicherweise quer, wie in Bild 2.30 zu sehen, entlang der Attrappen die Detektion aktivierte und kurz darauf das Ausweichverhalten startete.

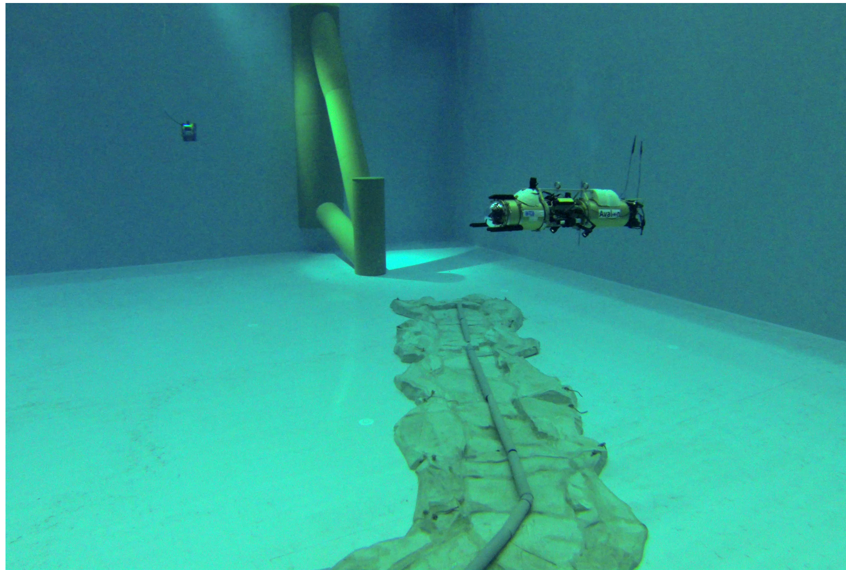


Figure 2.30: Avalon während der Pipelineverfolgung für die Demonstrationsmission.

## 2.6 Zusammenfassung und Herleitung der Kernprobleme

Wie in dem vorherigen Teil dieser Abhandlung herausgearbeitet, liegen die Probleme der modernen Robotik weniger an der Hardware als in den Softwaresystemen, im Speziellen der Verhaltensmodellierung.

Zwar bestehen immer wieder Probleme mit der Hardware, besonders wie bei Dagon mit dem GPS/WLAN-Modul, diese sind jedoch darin begründet, dass die Systeme zu wenig gewartet oder genutzt werden, was in der Forschung mutmaßlich ein häufiges Problem ist. Selbst mit einfachen Mitteln lassen sich robuste Komponenten durch beispielsweise das Vergießen realisieren. Einzelne Softwarekomponenten wie Detektoren arbeiten in den meisten Situationen zuverlässig genug, um verschiedene Aufgaben zu absolvieren. Die Analyse oder Erstellung dieser Detektoren im Detail soll nicht Bestandteil dieser Arbeit sein. Ein Beispiel ist aber die Pipelineverfolgung. Sie arbeitet in den meisten Fällen hinreichend zuverlässig, Fehlerfälle konnten jedoch über höhere Eben, gute modellbasierte Ansätze vorausgesetzt, leicht behoben oder abgefangen werden. Auch ist eine komplexe Hardwarebasis nicht immer nötig, um die aktuellen Probleme aufzuzeigen oder den Stand der Technik,

speziell in den Bereichen Robotik und KI, voranzutreiben. Wenngleich Avalon die technisch einfachere Basis ist, ließen sich viele Probleme und komplexe Missionen mit diesem Fahrzeug sehr gut lösen. Die Systeme autonomer zu gestalten erfordert keine besseren oder noch raffinierteren Hardwareplattformen mit einer noch größeren Anzahl anfallender Daten. Das Verhalten der Systeme komplexer und fehlersicherer zu gestalten, kann selbst mit einfachen Systemen ermöglicht werden. Sicherlich benötigen Systeme, die über Monate oder Jahre autonom operieren, weitere Verbesserungen der Hardware. Diese sind jedoch aus den Erfahrungen, die im Rahmen dieser Arbeit gewonnen wurden, einfacher zu realisieren als ein Softwaresystem zu entwickeln, das mit gängigen Entwicklungsmethoden robust, nachvollziehbar und intelligent Entscheidungen treffen kann. Eventuell auftretende Fehler in der Hardware oder innerhalb von Softwarealgorithmen zu erkennen und mittels intelligenter Methoden abzufangen oder mit ihnen umgehen zu können, sollte der Fokus neuer Entwicklungen sein. Es ist unnötig und aus der Erfahrung heraus nicht relevant, dass eine Hardwarekomponente oder Softwarekomponenten in allen Situationen fehlerfrei funktionieren. Hingegen sind das Zusammenspiel aller Komponenten und das Abfangen möglicher Fehler die Herausforderung in der Entwicklung künftiger intelligenter Systeme.

Aus diesen Gründen wird der Fokus des nächsten Kapitels dieser Arbeit auf der Entwicklung eines Systems liegen, das diesen Anforderungen gerecht werden kann. Die aufgetretenen Probleme des Plan-Managers Roby/Syskit werden detailliert analysiert und betrachtet. Es wird im Verlauf dieser Arbeit gezeigt, dass viele dieser Probleme durch eine nicht durchdachte Modellierung, unnötig hohe Komplexität sowie eine zu schwachen Formalisierung entstanden sind. Des Weiteren wird gezeigt, dass durch striktere modellgetriebene Ansätze solche Probleme gelöst werden können oder gar nicht auftreten würden.

# 3 Constraintbasierte Planung von Komponentennetzwerken

Wie zuvor aufgezeigt, ist ein großes Problem der Robotik das Management von Komponentennetzwerken. Hier bestehen sowohl Probleme in der Portierbarkeit von modellgetriebenen Lösungen als auch in der Unflexibilität oder Fehleranfälligkeit bestehender Verfahren.

Die Probleme lassen sich dabei auf verschiedene Grundprobleme zurückführen und beruhen dabei immer auf fehlerhaften oder nicht vorhandenen Konzepten. Die relevanten Probleme, die bereits im vorherigen Kapitel aufgezeigt wurden, sind in diesem Zusammenhang:

1. Fehlende Flexibilität der Konfigurationsmöglichkeiten von weichen oder teilweise widersprüchlichen Anforderungen (wie beispielsweise im Fall des Sonars, Abschnitt: 2.4.4).

2. Fehlende Flexibilität in der Übertragbarkeit der Algorithmen. Bedingt durch die fehlende Möglichkeit, systemweite Spezialisierungen oder Constraints zu definieren, ist das Verhalten bei einer automatischen Selektion nicht immer spezifiziert. Dieses Problem wurde anhand von systemspezifischen Konfigurationen aufgezeigt, wie in Abschnitt 2.4.4 beschrieben.

3. Probleme, die durch fehlende Modelltrennung von Algorithmen entstehen. Durch die Einbindung von Laufzeit-Code in die Modellschicht, wie in Beispiel 2.4.4 gezeigt, werden sämtliche Konzepte und Grundlagen der modellgetriebenen Entwicklung gebrochen. Hierdurch ist es nicht mehr möglich, die Modelle auf Konsistenz oder Gültigkeit zu analysieren.

In dieser Arbeit soll untersucht werden, wie sich solche Probleme mit Hilfe von constraintbasierten Verfahren lösen lassen. Dazu werden die bestehenden Modelle auf Schwachstellen hin untersucht und ein neuer Ansatz präsentiert, der durch minimale Änderungen innerhalb der Modelle eine robustere und portablere Lösung bieten soll. Im Laufe der Arbeit wird gezeigt, dass der erste und zweite Punkt der oben beschriebenen Problemklassen durch die gleiche Änderung bestehender Konzepte behoben werden kann. Der dritte Punkt kann

durch eine strikte Reduktion des Plan-Managers auf eine Modellebene gelöst werden, wobei dies keinen Verlust an Flexibilität oder Funktionalität darstellt.

Das Ziel der Arbeit ist es nunmehr also, ein neues Konzept und System zu erarbeiten, das die Vorteile bestehender Verfahren aufgreift und diese erweitert. Dadurch wird das neu entstehende System einerseits robuster und eindeutiger, aber auch mächtiger in der Ausdrucksstärke sein. Gleichzeitig werden formale Bedingungen erarbeitet und definiert, die die Behandlung von Komponentennetzwerken formalisieren. Somit wird es möglich sein, eine formale Verifikation des Systemverhaltens durchzuführen. Eine Formalisierung der Bestimmung eines Komponentennetzwerkes wurde von bestehenden Verfahren bisher nicht realisiert und ist, neben dem Entwurf des Gesamtsystems, ein wesentlicher Bestandteil dieser Abhandlung. Die Formalisierung ermöglicht zudem die Verhaltensanalyse und -bestimmung eines Systems über die Zeit aus einer vollständig neuen Perspektive.

## 3.1 Erläuterung des gewählten Lösungswegs

Im Folgenden werden die bestehenden Modelle des Plan-Managers Syskit/Roby abgeändert. Zunächst werden die Konzepte, die bisher verfolgt wurden, kurz dargestellt sowie die Vor- und Nachteile dieser Modelle, speziell im Vergleich zum Stand der Technik, diskutiert, um dann Lösungsvorschläge darzulegen.

### 3.1.1 Detaillierung der Syskit-Modelle

Syskit ist eine Kombinationslösung aus Modellrepräsentation und Anbindung dieser Modelle an das Rock-Framework. Die konkrete Einbindung wurde bereits in Abschnitt 2.4.3 beschrieben. Die Modelle sind dabei hierarchisch organisiert und durch Interfaces, sogenannte DataServices, abstrahiert. Diese lose hierarchische Kopplung stellt ein gutes Konzept zur Wiederverwendung und Integration von Algorithmen in Systemen dar. Dabei ist Roby ein event-/aktionsgetriebenes System, das in der Lage ist, Roboterverhalten (sogenannte Pläne) zu erstellen und zu behandeln. Diese beiden Komponenten interagieren, um das System in den gewünschten Zustand zu bringen bzw. eine Mission zu vollziehen.

Dabei setzt Syskit den Fokus auf Konzepte der modellgetriebenen Softwareentwicklung. Syskit ist so konzipiert, dass es dem Entwickler Erlaubt, normalen, nicht modellbasierten Programmcode einzuschleusen, der von dem Roby-Event-System behandelt bzw. ausgeführt wird. Diese Integrationsfähigkeit erscheint auf den ersten Blick als hilfreich, jedoch durchbricht sie sämtliche Konzepte und Regeln der modellgetriebenen Softwareentwicklung, da eine Verifikation und Analyse nicht mehr möglich ist. Somit lassen sich keine Modellverifikationen mehr vornehmen und das System ist nicht mehr evaluierbar. Des

Weiteren werden sämtliche Modelle erst zur Laufzeit ausgeführt, was zu der Problemklasse drei wie eingangs in diesem Kapitel beschrieben führt.

Das Sonarbeispiel aus Abschnitt 2.4.4 und die daraus resultierende Problemklasse eins entsteht, da es in Roby nicht möglich ist, weiche Anforderungen an Komponenten zu stellen. Wäre es möglich zu definieren, dass die Lokalisierung das Sonar präferiert im 360°-Modus betreibt, aber andere Lösungen (hier 90°) erlaubt wären, wäre eine konsistente und gültige Modellbeschreibung möglich. In Syskit hingegen führen unterschiedliche Anforderungen immer zu einer Nichterfüllbarkeit des Modells. In umgekehrter Betrachtung führen nicht definierte und mehrdeutige Lösungen ebenfalls zu einer Nichterfüllbarkeit des Modells, was zu Problemklasse zwei führt.

### 3.1.2 Vorschlag zur Änderung der Modelle

Die Idee hinter dieser Arbeit ist, die aufgeführten Probleme mit minimalen Änderungen an den Modellen zu lösen. Die Modelle, die Syskit verwendet, haben sich grundsätzlich als geeignete Repräsentation für robotische Systeme und deren Aufgaben herausgestellt. Diese Arbeit trennt daher die Syskit-Modelle von Syskit und betrachtet die Konzepte dieser Modelle unabhängig von dem gewählten Lösungsweg, den Roby und Syskit verfolgen, um die Modelle zu lösen.

Um das Sonarproblem zu lösen, ist es vonnöten, bevorzugte Konfigurationen definieren zu können. Dies resultiert aus dem Fakt, dass die Lokalisierungskomponente in der Lage ist, ihre Arbeit zu verrichten, auch wenn das Sonar nur ein 90°Fenster betrachtet. Die Wandverfolgung benötigt jedoch eine höhere Aktualisierungsfrequenz, da in dem Zustand das AUV näher an der Wand operiert. Diese Anforderungen ließen sich über Güteklassen wie folgt ausdrücken:

- Lokalisierung:
  - Güte 200 (wenn Sonar 360°)
  - Güte 50 (wenn Sonar 90°)
- Wandverfolgung: benötige Sonar 90°

Es ließen sich ebenfalls andere Szenarien hieraus ableiten. Als Beispiel soll ein System mit einem Schwenk/Neige-Laserscanner dienen. Das System kann nicht schneller fahren, als der Schwenk/Neige-Scanner den Bereich vor dem System erfassen kann. Die maximale Fahrgeschwindigkeit ergibt sich somit aus der Erfassungsrate und der Verarbeitungsgeschwindigkeit der Algorithmen. Die modellgetriebene Softwareentwicklung könnte daher auch komplexere Formeln für Zusammenhänge erlauben.

Es könnte beispielsweise definiert werden, dass die Systemgeschwindigkeit von verschiedenen Faktoren, wie der Wahrnehmungsgeschwindigkeit, abhängt. Somit wäre sichergestellt, dass in Abhängigkeit von der Systemkonfiguration (egal ob durch falsche Sensorauswahl oder anderweitige Konfigurationen von Sensoren) ein System nicht schneller fahren dürfte, als die Sensoren die Umgebung vor ihm wahrnehmen. Interessant ist dabei, dass sich dadurch explizit das Systemverhalten einschränken lässt und ungültige Anforderungen an das Systemverhalten bereits auf Modellebene identifiziert werden können.

Diese Überlegungen führen zu der Erkenntnis, dass ein constraintbasiertes Verfahren ermöglichen sollte, zu bestimmen, welche Konfiguration ein System zu welcher Zeit braucht, um eine entsprechende Aufgabe lösen zu können.

Wäre beispielsweise auf einem Robotersystem kein Scanner verfügbar, könnte keine Navigation durchgeführt werden, da die Umgebung nicht wahrgenommen werden kann. Sollte in einem anderen Szenario beispielsweise die Decke gescannt werden, wäre ein Fahren zu diesem Zeitpunkt nicht möglich, da die Wahrnehmung der vorderen Umgebung eingeschränkt ist. Durch Constraints ließe sich so eine Systemkonsistenz gewährleisten, ohne dass der Entwickler des Systems alle Rahmenbedingungen selbst prüfen müsste. Diese Nutzungsweise löst zugleich den zweiten Punkt der vorgestellten Problemklassen. Bei einem solchen System wäre es nicht nötig, mehrdeutige Lösungen zu verbieten. Hingegen wären solche Lösungen immer valide, solange alle (globalen) Constraints erfüllt sind. Dürfte eine spezifische Möglichkeit nicht selektiert werden, ist es an dem Designer der Modelle, diese Constraints möglichst allgemeingültig zu erfassen.

Dieses Vorgehen erhöht die Portabilität von Modellen enorm, da im Gegensatz zu bestehenden Verfahren die Modelle selbst analysiert werden könnten, um valide Lösungen zu finden. Somit wäre ein Problem auf eine Abbildung von Konfigurationen und Komponenten auf Interfaces und Algorithmen(-instanzen) reduzierbar. Die Auswahl ist dabei abhängig von der zur Verfügung stehenden Hardware, Software sowie dem aktuellen Szenario.

Wie zuvor angeführt, widerspricht Syskit/Roby sich selbst in der Intention der klaren modellgetriebenen Softwareentwicklung, da es erlaubt, Laufzeitcode in die Modelle einzubinden, was die Evaluierbarkeit mit heute zur Verfügung stehenden Mitteln unmöglich macht. Dies liegt an der Nichtbeweisbarkeit (Stichwort, Halteproblem [60]) von komplexen Programmiersprachen. Um das Halteproblem zu umgehen, ist die Reduktion auf strikt evaluierbare Modelle nötig. Dies widerspricht der Flexibilität, die ursprünglich durch das Konzept von Syskit angestrebt wurde. Jedoch wird im Folgenden gezeigt, wieso dies keine technische Einschränkung für die komponentenbasierte Softwareentwicklung und modellgetriebene Verhaltensbeschreibung ist. Vielmehr ist das Gegenteil der Fall, da eine klare formale Trennung der Ebenen ein klareres Verständnis und eine klarere Trennung von funktionalen Ebenen ermöglicht, die hingegen durch bestehende Konzepte aufgeweicht wurden. Ein typisches Beispiel ist in Abbildung 3.1 zu sehen.

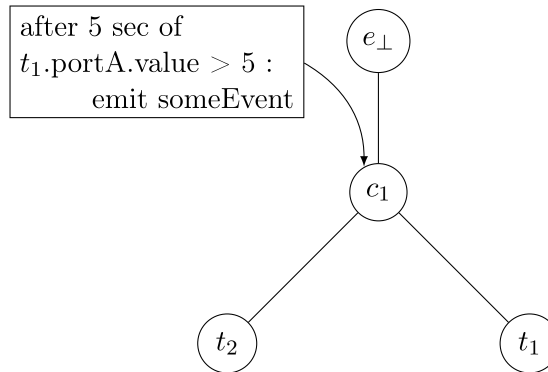



Figure 3.1: Bisheriges Konzept: Beispiel von Laufzeitcode innerhalb von Kompositionen, $e_\perp$ als Wurzel im Abhängigkeitsgraphen sowie $t_1, t_2$ als Kinder der Komposition $c_1$, die Laufzeitcode beinhaltet

Fälle wie diese sind bei genauerer Betrachtung invalide Formulierungen. Kompositionen sollten selbst keine Entscheidungen über das Verhalten durch, wenngleich einfache, algorithmische Analysen treffen. Die Intelligenz, die eine Komponente beinhaltet, kann und sollte in die Ebene der Komponenten selbst verschoben werden. Diese Umformung ist beispielhaft in Abbildung 3.2 dargestellt.

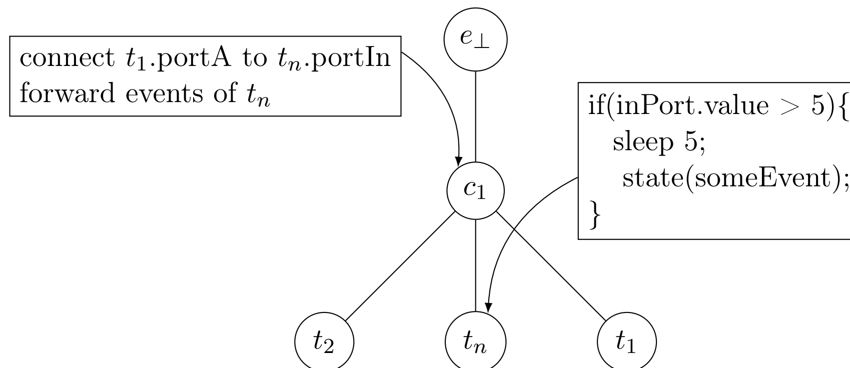



Figure 3.2: Neues Konzept: Laufzeitcode nur als Algorithmen innerhalb der Tasks. Kompositionen enthalten nur Modellwissen.

Bei Befolgen on dieses neuen Konzepts wird deutlich, dass die eigentlichen datenverarbeitenden Instanzen innerhalb des Netzwerkes selbst angesiedelt sind. Eine Supervisionsschicht oder ein Plan-Manager hat nicht die Aufgabe, Daten selbst zu analysieren oder auszuwerten. Er soll lediglich auf relevante Änderungen des Zustandes von Komponenten

reagieren. Es ist nicht nötig, die Evaluierbarkeit einer Mission oder eines Komponentennetzwerkes durch das Erlauben von touringvollständiger Behandlung von Daten innerhalb des Plan-Managers zu erschweren. Eine Begrenzung des Funktionsumfanges ist ein gängiger Weg, um Verifikationen moderner Softwareentwicklungen zu ermöglichen. Durch Eingrenzen des Fokuses verschiedener Lösungen wird es möglich, formale Verifikationen der Sub-Komponenten durchzuführen. Zusätzlich erlauben es formale Modelle wie [4], Verifikationen anhand zeitlicher Bedingungen durchzuführen. Somit könnten die Modellkonzepte selbst komplexere Verhaltensdefinitionen, wie im vorherigen Beispiel 3.1, wieder auf Modellebene erlauben. Durch eine Modellabstraktion einer solchen Erweiterung wären weitere Verifikationen auf Modellebene möglich. Ansonsten wären solche Verifikationen nur durch statische Code-Analysen umzusetzen. Anhand des neu vorgestellten Ansatzes und der strukturierten Trennung der Ebenen ist die Verhaltensverifikation impliziter Bestandteil der Untersuchung eines Komponentennetzwerkes bzw. der Untersuchung der Abfolge mehrerer Netzwerke über die Zeit. Durch diese Separation zwischen Modellebene und Laufzeitcode können diverse Probleme umgangen werden oder in dieser Form nicht mehr auftreten.

- Die Modellebene ist ohne Simulation formell verifizierbar. Demzufolge:
  - Lassen sich Sequenzierungsfehler erkennen,
  - werden Fehlerhafte Einstellungen analysierbar und
  - Tippfehler in den Modellen führen nicht erst zur Laufzeit zu Fehlern.

Diese Umformung widerspricht jedoch verschiedenen Grundannahmen, die Roby und Syskit verfolgen. Roby wurde entwickelt, um ein möglichst flexibles System bereitzustellen. Dieses event-/actionbasierte System kann reaktiv neue Zustände bestimmen. Intern werden Komponentennetzwerke berechnet, die auf Graphprobleme reduziert werden. Syskit sollte ein modellgetriebenes Interface zu Roby sein, das zugleich die Anbindung an das Orocos-Framework [58] sowie eine Modellsprache bereitstellt. Dabei folgt Syskit zwei für diese Arbeit relevanten Prinzipien.

Einerseits gehen die Syskit-Konzepte implizit davon aus, dass zwei Teilbäume im Abhängigkeitsgraphen der entstehenden Komponentennetzwerke sich gegenseitig nicht beeinflussen dürfen. Andererseits ist der Plan-Manager Roby/Syskit ein event-/actiongetriebenes System, das zur Laufzeit dynamisch neue Zustände plant. Hierbei können die Modelle durch Laufzeitcode ergänzt werden, um ein möglichst flexibles System bereitzustellen.

Diese Arbeit orientiert sich größtenteils an den Konzepten von Roby/Syskit, jedoch wird die Flexibilität in den Syskit/Roby-Modellen an manchen Stellen bewusst deutlich reduziert, um die angesprochene formale Verifikation zu erlauben. Gleichzeitig wird gezeigt, dass durch diese Reduktion keine Funktionalität verloren gehen wird, stattdessen wird ein

deutlich klareres Interface geschaffen. Auch das Konzept der Betrachtung von Komponentennetzwerken als Graphproblem wird als essentielle Basis fortgesetzt. Im Gegensatz dazu wird anstelle einer iterativen programmiertechnischen Lösung ein formales constraintbasiertes Konzept erarbeitet und präsentiert, das die Komponentennetzwerke bestimmt. Dieses Konzept ermöglicht es, anstelle des event-/actiongetriebenen Systems ein formales modellbasiertes System zu nutzen, das auf Modellbasis analysiert und verifiziert werden kann.

Die Konzepte, die bereits Syskit/Roby verfolgen, besitzen nur einen endliche Menge an Zuständen. Aus diesem Grund ist es nicht nötig, dass der Plan-Manager eine reaktive, online ausgeführte Realiation ist. Aufgrund der Endlichkeit der erreichbaren Zustände ist es möglich, eine Verifikation des Gesamtverhaltens durchzuführen, ohne zur Laufzeit auf einen reaktiven Planer oder Suchalgorithmen angewiesen zu sein.

### 3.1.3 Umformung der Modelle in eine formalisierte Problemstellung

Wie in dem vorherigen Absatz bereits angeführt, resultiert die Einführung von Constraints für Kinder in einem Graphproblem. Es ist vonnöten, einen Graph zu finden, der alle Anforderungen (möglichst optimal) erfüllt. Diese Suchprobleme erfordern eine globale Suche über den gesamten Zustand aller Module.

Eine Analogie zu Planungsproblemen wird hierbei schnell deutlich. Gängige Beschreibungssprachen für diese Art von Planungsproblemen in der Robotik sind beispielsweise HTN [32] oder PDDL [43]. Beide Sprachen erlauben es, *Probleme* in ihren eigenen Sprachmodellen auszudrücken. Diese Probleme können dann von Planern, also speziellen Suchalgorithmen, gelöst werden. Für PDDL existieren verschiedene *Planer*, mit denen die in PDDL beschriebenen *Probleme* gelöst werden können.

#### 3.1.3.1 Überführung der Modelle in PDDL

PDDL wurde in [43] eingeführt und in vielen Arbeiten wie [18], [13], [22], [21], [26] fortgeführt. PDDL steht hierbei für ,"Planning Domain Definition Language‘. Sie basiert auf den frühen Konzepten von STRIPS (Stanford Research Institute Problem Solver) [16] und ADL (Action Description Language) [20]. PDDL ist eine Planungssprache, die für typische Weltzustands-Planungsprobleme wie beispielsweise die ‚Türme von Hanoi‘ genutzt wird. PDDL und seine Erweiterungen finden in der Robotik breite Anwendung, speziell in den Bereichen der Wissensrepräsentation für Planungsprobleme [46], [68], [64]. In PDDL ist die Lösung jeweils eine Abfolge von Aktionen, um zu einem gewünschten Zielzustand zu gelangen.

**Einleitung** Zunächst wurden einfache Testmodelle erzeugt, um die Funktionsweise zu analysieren und eine geeignete Repräsentation der Syskit-Modelle in PDDL zu finden. Dabei müssen verschiedene Fälle abgebildet werden. Hierzu zählen die Abstraktionsfähigkeit der Syskit-Modelle sowie die Abhängigkeiten, die von Kompositionen entstehen können.

Die PDDL kennt dabei nur Vorbedingungen, die zum Zeitpunkt der Ausführung gültig sein müssen, sowie eine Veränderungsfunktion, die auf den Weltzustand angewendet wird, sofern die gegebene Aktion ausgeführt wird. Um die Syskit-Modelle darauf abbilden zu können, werden Abhängigkeiten von Kompositionen als Vorbedingungen modelliert. Ein DataService ist in dieser Darstellung eine Vorbedingung, die erfüllt sein muss, falls dieser DataService genutzt wird. Bei der Verwendung dieses DataServices wird ein korrespondierendes Symbol im Weltzustand abgelegt. Dies löst zugleich das Problem der Reihenfolge bei Abhängigkeiten, da zunächst alle abhängigen DataServices erfüllt sein müssen, bevor eine Komponente gestartet werden kann.

**Modelle** In PDDL werden verschiedene Einheiten beschrieben. Dazu zählen:

- Objekte
- Prädikate
- Startzustand
- Zielzustand
- Aktionen und Operatoren

All diese Einheiten müssen in der Art und Weise befüllt werden, dass das Syskit-Modell eine Abbildung erhält, die vom PDDL-Solver lösbar ist. Die Prädikate sind dabei die Eigenschaften, die erfüllt werden können, und bilden je nach Belegung den Weltzustand. Aktionen verändern den aktuellen Weltzustand. Welche Prädikate initial gelten, ist über den Startzustand definiert.

Die Syskit-Modelle werden wie folgt auf die PDDL-Modelle abgebildet:

Listing 3.1: Abbildung der Syskit-Modelle in PDDL

```
1 (define (domain network)
2   (:requirements :strips :equality :typing :conditional-
      ↪ effects)
3   (:types input_port output_port instance_req)
4   (:predicates
5          (has-output ?x - instance_req ?y - output_port)
```

```
6           (has-input ?x - instance_req ?y - input_port)
7           (is-connected ?x - output_port ?y - input_port)
8           (should-connected ?x - instance_req ?y - input_port ?
               ↪ z - output_port)
9           (requests ?x - instance_req ?y - instance_req)
10          (fullfills ?x - instance_req ?y - instance_req)
11          (depends ?x - instance_req ?y - instance_req)
12          (is-running ?x - instance_req)
13          (data-service ?x - instance_req)
14          (task ?x - instance_req)
15          (composition ?x - instance_req)
16          (is-root ?x - instance_req)
17 )
```

Der Codeblock 3.1 definiert verschiedene Prädikate (Eigenschaften), die für Objekte gelten können. Dies ist dabei der Basissatz an Prädikaten. Verschiedene Modelle ändern die Menge der Prädikate nicht.

```
1 (:action startt :parameters (?r - instance_req ?t -
     ↪ instance_req)
2  :precondition (and
3     (task ?t)
4     (not (is-running ?t)) ;;Unsure ob das rein soll
5     (or (depends ?r ?t) (fullfills ?t ?r))
6   )
7  :effect (and  (is-running ?t) (requests ?r ?t) )
8 )
```

Die Aktion *startt* (Start-Task) definiert die Änderung des Weltzustandes beim Ausführen der Aktion. Die Parameter sind dabei der Elternknoten sowie der Task selbst, der gestartet werden soll. Vorbedingungen sind, dass es sich bei $t$ um einen Task handelt, der Task bisher nicht läuft und dass $r$ von $t$ abhängt, bzw. $t$ von $r$ benötigt wird. Der Weltzustand ändert sich daraufhin wie folgt: Der Task $t$ ist aktiv (is-running) und er hängt von $r$ ab (requests ?r ?t).

```
1 (:action startds :parameters (?r - instance_req ?ds -
     ↪ instance_req)
2 :precondition (and
3     (not (is-running ?ds)) ;;Unsure ob das rein soll
4     (data-service ?ds)
```

```
5       (depends ?r ?ds)
6       (exists (?t - instance_req) (and (fullfills ?t ?ds) (is-
          ↪ running ?t) (requests ?ds ?t) ) )
7   )
8   :effect (and (is-running ?ds) (depends ?r ?ds) (requests ?r ?
      ↪ ds))
9   )
```

Das Starten eines DataServices mittels *startds* ist größtenteils analog umgesetzt wie der Start eines Tasks. Der Unterschied dabei ist, dass ein Task $t$ existieren muss, der den DataService $ds$ erfüllt, und dass der $ds$ selbst benötigt wird. Der Effekt ist, dass der DataService aktiv ist und von dem Elternobjekt abhängt.

```
1   (:action stop :parameters (?t - instance_req)
2    :precondition (and
3       (or (data-service ?t) (task ?t) (composition ?t) )
4       (is-running ?t)
5       (not (exists (?r - instance_req) (requests ?r ?t)))
6       )
7    :effect (and
8       (not (is-running ?t))
9       (forall (?dep - instance_req)
10          ;(when (depends ?t ?dep) (not (depends ?t ?dep) ) )
11          (not (requests ?dep ?t) )
12      )
13    )
14  )
```

Das Stoppen von Objekten ist nötig, wenn ein Verhaltenswechsel erfolgen soll. Gestoppt werden müssen dabei sämtliche Elemente, also Tasks, DataServices oder Kompositionen. Das Stoppen von diesen Elementen erfordert, dass es keinen Elternknoten im Netzwerk gibt, der von diesem Objekt $t$ abhängt. Der Effekt ist, dass das Objekt $t$ nicht mehr aktiv ist und alle Kind-Entitäten nicht mehr von diesem Objekt abhängen. Durch diese letzte Bedingung können alle Kind-Entitäten gestoppt werden, solange sie keinen anderen Elternknoten besitzen.

```
1   (:action start_root
2    :precondition (and
3      (not (is-running root))
```

```
4      ;(forall (?dep - instance_req) (imply (depends root ?dep) (
          ↪ or (and  (is-running ?dep) (requests root ?dep)) (
          ↪ depends root ?dep) ) ) )
5      (forall (?dep - instance_req) (imply (depends root ?dep)
6           (or
7                   (and (is-running ?dep) (requests root ?dep) )
8                   (depends ?dep root)
9           ) )
10     )
11    )
12   :effect (and (is-running root) (depends root root) (requests
        ↪ root root ) )
13  )
```

Die Aktion des Startens des Wurzelknotens ist explizit gegeben. Der Wurzelknoten hat keine weiteren Abhängigkeiten und kann nur gestartet werden, wenn alle Kinder, die von ihm abhängen, bereits gestartet wurden. Alle globalen Anforderungen an ein Netzwerk werden als Kinder des Wurzelknotens eingetragen. Dies bewirkt ein Starten und Auflösen aller Kind-Objekte. Vorbedingung für das Starten des Wurzelknotens ist, dass er nicht bereits aktiv ist. Des Weiteren muss, falls eine Komponente vom Wurzelknoten abhängt, die Implikation aus Zeilen 5 bis 8 gelten, dass sie bereits aktiv sein muss und von ihm entweder angefordert wurde oder direkt von ihm abhängt.

Hierbei ist eine bedeutende Unterscheidung hervorzuheben: Die Bedingung *requests :a :b* besagt, dass die Komponente *b* von der Komponente zum aktuellen Zeitpunkt abhängig ist, also dass beide im aktuellen Abhängigkeitsgraph verbunden sind.

Der Zustand *depends :a :b* sagt hingegen aus, dass es aus der Modellsicht eine Abhängigkeit von *a* zu *b* gibt. Diese Abhängigkeit kann entweder direkt oder durch einen DataService bestehen.

```
1  (:action startc :parameters (?r - instance_req ?c -
      ↪ instance_req)
2   :precondition (and
3     (not (is-running ?c))
4     (not (is-root ?c) )
5     (or (depends ?r ?c) (fullfills ?c ?r))
6     (composition ?c)
7     (forall (?dep - instance_req)
8      (imply (depends?c ?dep) (and (is-running ?dep) (requests ?
          ↪ c ?dep) ) ) )
```

```
9    )
10   :effect (and (is-running ?c) (requests ?r ?c) ) ;(depends ?r
       ↪ ?c) (forall (?dep - instance_req) (when (depends ?c ?dep
       ↪ ) (requests ?c ?dep ))))
11 )
12 )
```

Die letzte Aktion 3.1.3.1, die definiert wird, ist das Starten von Kompositionen. Neben den gleichen Bedingungen, die schon für Tasks und DataServices gelten, existieren weitere Bedingungen. Kompositionen bilden hierbei einen Sonderfall: Damit eine Komposition gestartet werden kann, müssen alle Abhängigkeiten von ihr gestartet sein. Solle eine Komponente von der aktuellen Komposition abhängig sein *depends ?c ?dep*, dann muss diese Komponenten aktiv und im Abhängigkeitsbaum eingetragen sein *requests ?c ?dep*. Diese Bedingung erfordert, dass alle Kinder einer Komposition zum Zeitpunkt des Startens der Komposition aktiv sind. Dadurch wird zugleich eine zeitliche Synchronisation und ein definiertes Startverhalten erreicht. Die Selektion einer geeigneten Kind-Komponente ist dabei Aufgabe des Planers selbst.

Der Effekt der Funktion *startc* ist, analog zu den Funktionen für die DataServices und die Tasks, dass diese Komposition vom Elternknoten *r* des Anfordernden (request-modul) abhängt und aktiv ist.

All diese Definitionen bilden die sogenannten *Domäne* in der PDDL-Sprache. Die Domäne entspricht dem Modell, sie definiert, welche Eigenschaften, Funktionen und Attribute es gibt. Neben der *Domäne* muss noch das *Problem* definiert werden, das vom Planer gelöst werden soll. Das Problem definiert alle Objekte, den Start- sowie Zielzustand. Aufgabe des Planers ist es dann, mittels der genannten Aktionen eine Sequenz zu finden, mit der der Start- zum Zielzustand transformiert werden kann.

Da die Syskit-Modelle dynamische Ruby-Sprachobjekte sind, wurde eine automatische Extraktion mittels eines Ruby-Interfaces erstellt, die die Syskit-Modelle in die PDDL-Sprache exportieren kann.

Dazu werden zunächst alle Syskit-Objekte in eine neue Datenstruktur und mittels einer Codegenerierung in die PDDL-Sprache überführt. Die Codegenerierung wurde mit Hilfe von der ERB-Template-Engine durchgeführt, die Bestandteil der Ruby-Interpretersprache ist.

```
1 (define (problem network001)
2   (:domain network)
3   (:objects
4 <% objects.each do o %>
```

```
5           <%= o.name %> - <%= o.metatype %>
6   <% end %>
7     )
```

Zunächst werden alle Objekte exportiert. Die Objekte sind dabei alle TaskContexts, Kompositionen, Ports sowie die DataServices.

```
1     (:init
2
3   <% objects.each do o %>
4       <% if o.type %>
5           (<%= o.type %>  <%= o.name %>)
6       <% end %>
7       <%if o.is_running? %>
8           (is-running <%= o.name %>)
9       <% end %>
10  <% end %>
11
12
13  <% objects.each do o %>
14      <% o.depends.each do d %>
15          <%if !d.disabled? %>
16              (depends <%= o.name %>  <%= d.name %>)
17          <% end %>
18      <% end %>
19  <% end %>
20
21  ; Begin fullfillments
22  <% objects.each do o %>
23      <% o.each_fullfillment do f %>
24              (fullfills <%= o.name %>  <%= f %>)
25      <% end %>
26  <% end %>
27
28  ; Begin inputs
29  <% objects.each do o %>
30      <% o.input_ports.each do p %>
31              (has_input <%= o.name %> <%=o.name %>.<%= p.name
                    ↪ %>)
32      <% end %>
```

```
33 <% end %>
34
35 ; Begin outputs
36 <% objects.each do o %>
37     <% o.output_ports.each do p %>
38             (has_output <%= o.name %> <%=o.name %>.<%= p.name
                   ↪ %>)
39     <% end %>
40 <% end %>
41             (is-root root)
42
43 ; Begin requirements
44 <% objects.each do o %>
45     <% o.requires.each do d %>
46             (depends <%= o.name %>  <%= d.name %>)
47     <% end %>
48 <% end %>
49
50   )
```

Der Startzustand umfasst dabei sämtliche Prädikate, die für die Objekte gelten. Diese Prädikate sind somit:

- Der Typ und Name eines Objektes,
  - ob ein Objekt einen oder mehrere DataServices erfüllt,
  - ob er Input- oder Output-Ports besitzt und wie die jeweiligen Ports heißen.
- Als Besonderheit Informationen über den Wurzelknoten.

```
1    (:goal (and
2
3
4      (forall (?t - instance_req)
5      (exists (?r - instance_req)
6              (imply
7              (is-running ?t)
8              (requests ?r ?t)
9              ;(depends ?r ?t)
10             )
11             )
```

```
12              )
13
14      (is-running root)
15    ))
16  )
```

Der PDDL-Zielzustand beruht auf der Annahme, dass der Wurzelknoten aktiv ist und alle Tasks, die aktiv sind, im Abhängigkeitsgraph einen Elternknoten besitzen müssen. Dadurch dass der Wurzelknoten aktiv sein muss, müssen alle Kinder ebenfalls aktiv sein. Somit baut sich ein Abhängigkeitsgraph auf, bis alle benötigten Komponenten selektiert wurden. Inaktive Komponenten werden hingegen gestoppt, da sie keine gültige Elternkomponente besitzen und somit im Graph nicht vorkommen dürfen.

Eine gekürzte Version des Avalon-Modells aus Syskit hat somit folgende Struktur:

```
1  (define (problem network001)
2    (:domain network)
3    (:objects
4          root - instance_req
5          WallServoing::WallServoing.motion_command -
             ↪ output_port
6          Buoy::Survey - instance_req
7    )
8    (:init
9          (composition  root)
10         (task  AuvControl::WorldToAligned)
11         (depends PoseAuv::PoseEstimatorCmp  Base::PositionSrv)
12         (fullfills Gps::GPSDTask  Base::PositionSrv)
13         (depends root  Pipeline::Follower)
14         (has-output mars-Task mars-Task.simulated-time)
15         (output-port mars-Task.simulated-time)
16   )
17   ...
18 )
```

**Ergebnisse** Die aus Syskit extrahierten Modelle wurden exportiert und mit verschiedenen PDDL-Lösern evaluiert. Die eingesetzten Löser sind in den Veröffentlichungen [53] sowie zusammenfassend in [19] beschrieben.

Die Wahl der Löser basiert auf einer bestehenden Infrastruktur, die die Evaluation verschiedener PDDL-Löser ermöglicht. Da der Fokus dieser Arbeit auf der Anwendung und weniger der Entwicklung von Planern liegt, wurden bestehende Planer auf das gegebene Problem angewendet.

**Minimales Start-Modell** Das minimale Modell stellt einen Test dar, der die grundlegende Funktionsweise der Modellierung exemplarisch untersuchen soll. Das zugrundeliegende Problem wurde wie folgt modelliert:

```
1  (define (problem network001)
2    (:domain network)
3    (:objects
4        root - instance_req
5        lights - instance_req
6    )
7    (:init
8          (composition root)
9          (depends root root)
10         (task lights)
11         (depends root lights)
12   )
13   (:goal
14   (and
15     (is-running root)
16     (forall (?t - instance_req)
17         (exists (?r - instance_req)
18             (imply (is-running ?t) (requests ?r ?t) )
19         )
20     )
21
22   )
23   )
24 )
```

Bereits bei diesem Problem lieferten nicht alle Planer Ergebnisse. In der Tabelle 3.1.3.1 sind die Ergebnisse der Auswertung dargestellt.

Alle Planer, die Ergebnisse erzeugten, hatten als Ergebnis die Sequenz von:

1. startt root lights

Table 3.1: Laufzeiten der PDDL Planer für das `Minimales Start-Modell`

| Name | Laufzeit (s) |
|---|---|
| BFSF | - |
| CEDALION | 0.24 |
| FDAUTOTUNE1 | 0.06 |
| FDAUTOTUNE2 | 0.06 |
| FDSS1 | 0.1 |
| FDSS2 | 0.04 |
| LAMA | 0.02 |
| LAMA2011 | 0.06 |
| RANDWARD | 0.02 |
| UNIFORM | 0.25 |

2. start_root

**Minimales Stop-Modell** Das folgende Modell soll die Funktionsweise aufzeigen, nicht benötigte Komponenten zu stoppen. Dieses Verhalten ist nötig, wenn sich Anforderungen an das Verhaltensnetzwerk ändern. Im oben genannten Beispiel 3.1.3.1 wurde das *lights*-Modul gestartet, das jedoch im folgenden Modell nicht mehr benötigt wird. Bei korrekter Funktionsweise müsste dieser Task gestoppt werden, um keine unnötigen Ressourcenkonflikte hervorzurufen.

```
1  (define (problem network001)
2    (:domain network)
3    (:objects
4        root - instance_req
5        lights - instance_req
6    )
7    (:init
8           (composition root)
9           (is-running lights)
10          (depends root root)
11          (task lights)
12   )
13   (:goal
14   (and
15     (is-running root)
```

```
16      (forall (?t - instance_req)
17          (exists (?r - instance_req)
18              (imply (is-running ?t) (requests ?r ?t) )
19          )
20      )
21    )
22    )
23  )
```

Auch hier sind nicht alle Planer in der Lage, Ergebnisse zu erzeugen. Die Tabelle 3.1.3.1 stellt die Ergebnisse dar, die der jeweiligen Planer erzeugt hat:

Table 3.2: Laufzeiten der PDDL Planer für das `Minimales Stop-Modell`

| Name | Laufzeit (s) |
|---|---|
| BFSF | - |
| CEDALION | 0.22 |
| FDAUTOTUNE1 | 0.04 |
| FDAUTOTUNE2 | 0.06 |
| FDSS1 | 0.09 |
| FDSS2 | 0.05 |
| LAMA | 0.06 |
| LAMA2011 | 0.03 |
| RANDWARD | 0.02 |
| UNIFORM | 0.21 |

Im Gegensatz zum ersten Test 3.1.3.1 variieren die Ergebnisse der Planer voneinander. Alle Planer ermitteln, dass die folgenden zwei Aktionen ausgeführt werden müssen:

1. start_root
2. stop lights

Die undefinierte Reihenfolge liegt darin begründet, dass das Stoppen des nicht verwendeten lights-Tasks keine Abhängigkeit vom Starten des Wurzelknotens aufweist. Es ist nicht definiert, ob zunächst alle nicht benötigten Tasks gestoppt werden müssen, bevor ein neues Netzwerk gestartet werden kann. Dieses Verhalten kann unter Umständen sogar gewünscht sein, beispielsweise falls Tasks wiederverwendet werden oder die Umschaltzeit minimiert werden soll.

### Vollständiges Avalon-Modell

Im Folgenden wird das vollständige Avalon-Modell evaluiert, das aus der Extraktion des Syskit-Modells hervorgegangen ist.

Als Beispiel wurde eine Sensorfusion aus Orientierung und Tiefenmesswerten gewählt. Diese Anforderung ist variabel definiert, da verschiedene abstrahierende DataServices genutzt werden. Eine graphische Darstellung des Modells ist in Abbildung 3.3 zu sehen.

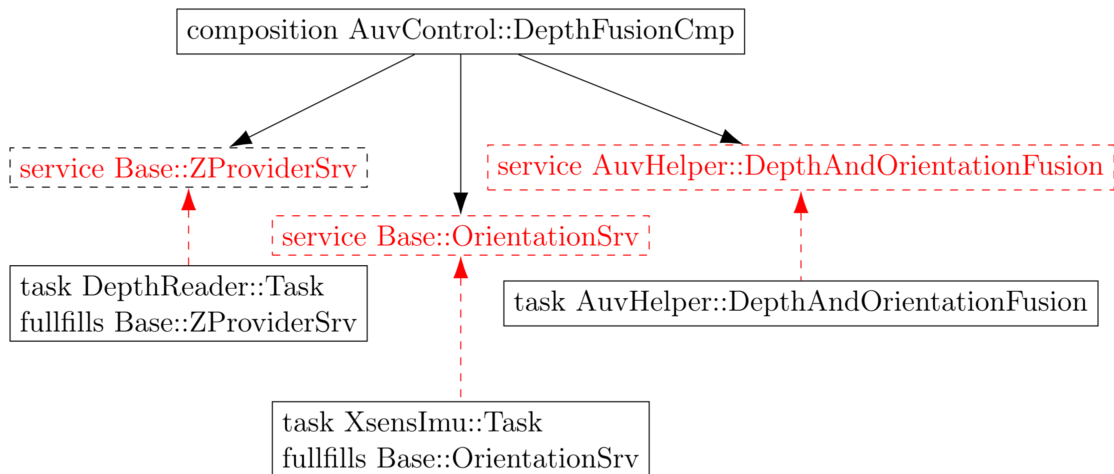



Figure 3.3: Graphische Darstellung eines Teils des Avalon-Modells.
Zu erkennen ist eine Sensorfusion *DepthFusionCm* sowie ihre Abhängigkeiten, verschiedene Komponenten erfüllen die abstrakten Anforderungen der *Services*

1. startt auvcontrol::depthfusioncmp auvhelper::depthandorientationfusion
2. startt base::orientationsrv xsensimu::task
3. startds auvcontrol::depthfusioncmp base::orientationsrv
4. startt base::zprovidersrv depthreader::task
5. startds auvcontrol::depthfusioncmp base::zprovidersrv
6. startc root auvcontrol::depthfusioncmp
7. start_root

Erneut liefern nicht alle eingesetzten Planer Ergebnisse 3.1.3.1. Auch die Reihenfolge ist unterschiedlich, was jedoch für die gegebene Problemstellung valide ist, da es irrelevant ist, in welcher Reihenfolge die Tasks gestartet werden, solange die Kompositionen, die einen

Table 3.3: Laufzeiten und Ergebnisse der PDDL Planer für das `Vollständiges Avalon-Modell`

| Planer | | |
|---|---|---|
| BFSF | Runtime<br>Solution | - |
| CEDALION | Runtime<br>Solution | 32:02.11<br>1,2,3,4,5,6,7 |
| FDAUTOTUNE1 | Runtime<br>Solution | 33:20.11<br>1,4,5,2,3,6,7 |
| FDAUTOTUNE2 | Runtime<br>Solution | 33:20.08<br>1,2,4,3,5,6,7 |
| FDSS1 | Runtime<br>Solution | 21:10.22<br>1,4,5,2,3,6,7 |
| FDSS2 | Runtime<br>Solution | 21:27.79<br>2,1,3,4,5,6,7 |
| LAMA | Runtime<br>Solution | - |
| LAMA2011 | Runtime<br>Solution | 33:20.13<br>1,4,5,2,3,6,7 |
| RANDWARD | Runtime<br>Solution | - |
| UNIFORM | Runtime<br>Solution | 32:16.11<br>1,2,4,3,5,6,7 |

Service erfüllen, nach den Tasks gestartet werden. Gleiches gilt für die DataServices. Ein DataService kann erst erfüllt werden, wenn der dazugehörige Task gestartet wurde. In den Ergebnissen gilt dies speziell für die Orientierung sowie die Tiefenmesswerte und ihre korrespondierenden Tasks: Xsens::IMU und DepthReader::Task. Um weitere Lösungen identifizieren zu können, wird testweise die XSens IMU abgeändert. Es wird die Information entfernt, dass sie eine Orientierung liefern kann. Dies wird durch das Entfernen der Zeile *(fullfills XsensImu::Task Base::OrientationSrv)* aus dem Modell erreicht. Dadurch ist der Planer gezwungen, einen alternativen Provider für den DataService *Base::OrientationSrv* zu finden. Die Ergebnisse sind in Tabelle 3.1.3.1 dargestellt.

1. startt auvcontrol::depthfusioncmp auvhelper::depthandorientationfusion
2. startt poseauv::ikforientationestimatorcmp xsensimu::task
3. startt poseauv::ikforientationestimatorcmp

wallorientationcorrection::orientationinmap

4. startt poseauv::ikforientationestimatorcmp orientationestimator::baseestimator
5. startt poseauv::ikforientationestimatorcmp fogkvh::dsp3000task
6. startc base::orientationsrv poseauv::ikforientationestimatorcmp
7. startds auvcontrol::depthfusioncmp base::orientationsrv
8. startt base::zprovidersrv depthreader::task
9. startds auvcontrol::depthfusioncmp base::zprovidersrv
10. startc root auvcontrol::depthfusioncmp
11. start_root

Um die hohen Laufzeiten der Modelle zu untersuchen, wurden die Modelle angepasst. Im Experiment aus Tabelle 3.1.3.1 wurden sämtliche (bisher für die Modelle nicht verwendeten) Ports entnommen. Wie zu sehen ist, unterscheiden sich die Laufzeiten jedoch nicht signifikant von denen der unveränderten Modellen. Ein minimalisiertes Beispiel auf den originalen Modellen wird in Tabelle 3.1.3.1 dargestellt. In diesem Beispiel wird lediglich eine IMU als direkter Task benötigt. Die Laufzeiten in diesem Beispiel sind bereits stark reduziert. Der letzte Test mit einem stark reduzierten Modell zeigt die gleiche Anforderung, jedoch nur mit benötigten Kompositionen. Die Laufzeiten hier sind, wie erwartet, niedrig 3.1.3.1. Dies deutet darauf hin, dass die Suche in den verschiedenen Modellen das Problem für die Löser hervorruft. Je tiefer eine Suche im Suchraum absteigen muss, umso stärker steigt der Suchaufwand.

Wenngleich das entwickelte Verfahren ein Offline- und modellbasierter Ansatz ist, ist die Laufzeit kritisch zu betrachten. Die Softwaremodelle werden in Zukunft größer werden und der Suchraum, den die Solver abarbeiten müssen, ist in Bezug auf die Suchkomplexität exponentiell wachsend.

**Zusammenfassung** Die PDDL-Solver liefern für die bisher evaluierten Modelle, korrekte Ergebnisse. Eine grundsätzliche Funktionsweise scheint gegeben. Jedoch ergeben sich verschiedene Probleme. Die produzierten Lösungen der Planer führen zum Zielzustand, jedoch ist der interne Suchraum nicht Bestandteil der Lösung. Dies zeigt sich in dem Beispiel zwischen `Minimales Start-Modell` und `Minimales Stop-Modell`. Der erste Schritt bestimmt die nötigen Schritte, jedoch ist die Information *(is-running lights)* nicht Bestandteil der Ausgabe. An dieser Stelle kann nur vermutet werden, dass der interne Suchzustand nur partiell verfolgt wird und für das Forschungsfeld von keinem weiteren Interesse ist. Jedoch ist für die hier genannte Problemstellung der Zielzustand sehr interessant. Dieser Zustand

wird benötigt, um Sequenzen von Netzwerken bestimmen zu können, da die Information, welche Tasks aktiv sind, als Eingabe für die Folgeiteration benötigt wird. Es wäre zwar möglich, den Zustand zu rekonstruieren, indem die Schritte der produzierten Lösung abgespielt werden, dies würde jedoch einen weiteren Schritt bedeuten, der neue Fehler in die späteren Lösungen integrieren könnte.

Ein weiterer Punkt, den die eingesetzten PDDL-Planer nicht erfüllen, ist das Identifizieren sämtlicher Lösungen. Die Planer finden jeweils nur die erste mögliche Lösung. Dies wird im Beispiel `Vollständiges Avalon-Modell` deutlich. Die Invalidierung der zuerst berechneten Lösung liefert eine weitere Lösung. Dieses Ergebnis wurde nur mit einem manuellen Eingriff in das Modell erreicht. Es wäre Nötig, sämtliche berechneten Lösungen zu evaluieren und künstlich ungültig zu machen, um eine weitere Iteration mittels der PDDL-Solver zu bestimmen. In Anbetracht der Laufzeit und des benötigten Wissens, um die Modelle abzuändern, scheint dieser Weg enorm aufwendig zu sein, sowohl seitens des Entwicklungsaufwandes als auch auf Grund der Such- und Zeitkomplexität. Auch wäre hier eine exponentielle Komplexitätssteigerung zu erwarten, da jede zusätzliche Selektionsmöglichkeit den Suchraum verdoppelt.

Table 3.4: Alternative Lösung zu Tabelle 3.1.3.1 bedingt durch die Invalidierung der vorherigen Lösung.

| | | |
|---|---|---|
| BFSF | Runtime<br>Solution | - |
| CEDALION | Runtime<br>Solution | 32:15.59<br>1,2,3,4,5,6,7,8,9,10,11 |
| FDAUTOTUNE1 | Runtime<br>Solution | 33:20.15<br>1,8,9,5,4,3,2,6,7,10,11 |
| FDAUTOTUNE2 | Runtime<br>Solution | 33:20.08<br>1,4,3,5,2,6,8,7,9,10,11 |
| FDSS1 | Runtime<br>Solution | 21:23.51<br>1,8,9,5,4,3,2,6,7,10,11 |
| FDSS2 | Runtime<br>Solution | 21:32.66<br>5,2,1,8,9,4,3,6,7,10,11 |
| LAMA | Runtime<br>Solution | - |
| LAMA2011 | Runtime<br>Solution | 33:20.11<br>1,8,9,5,4,3,2,6,7,10,11 |
| RANDWARD | Runtime<br>Solution | - |
| UNIFORM | Runtime<br>Solution | 32:12.55<br>1.4.3.5.2.6.8.7.9.10.11 |

Table 3.5: Laufzeiten der Alternative Lösung des Modells ohne das Vorhandensein von Ports.

| | | |
|---|---|---|
| BFSF | Runtime<br>Solution | - |
| CEDALION | Runtime<br>Solution | 32:09.80<br>1,2,3,4,5,6,7 |
| FDAUTOTUNE1 | Runtime<br>Solution | 33:20.15<br>1,4,5,2,3,6,7 |
| FDAUTOTUNE2 | Runtime<br>Solution | 33:20.08<br>1,2,4,3,5,6,7 |
| FDSS1 | Runtime<br>Solution | 21:19.82<br>1,4,5,2,3,6,7 |
| FDSS2 | Runtime<br>Solution | 21:26.23<br>2,1,3,4,5,6,7 |
| LAMA | Runtime<br>Solution | - |
| LAMA2011 | Runtime<br>Solution | 33:20.15<br>1,4,5,2,3,6,7 |
| RANDWARD | Runtime<br>Solution | - |
| UNIFORM | Runtime<br>Solution | 32:04.98<br>1,2,4,3,5,6,7 |

Table 3.6: Laufzeiten Alternative Lösung des Modells mit direkter IMU Abhängigkeit.

| | | |
|---|---|---|
| BFSF | Runtime<br>Solution | - |
| CEDALION | Runtime<br>Solution | 6:08.26<br>1,2 |
| FDAUTOTUNE1 | Runtime<br>Solution | 10:50.30<br>1,2 |
| FDAUTOTUNE2 | Runtime<br>Solution | 11:01.33<br>1,2 |
| FDSS1 | Runtime<br>Solution | 11:22.84<br>1,2 |
| FDSS2 | Runtime<br>Solution | 10:13.62<br>1,2 |
| LAMA | Runtime<br>Solution | - |
| LAMA2011 | Runtime<br>Solution | 11:01.70<br>1,2 |
| RANDWARD | Runtime<br>Solution | - |
| UNIFORM | Runtime<br>Solution | 5:21.92<br>1,2 |

Table 3.7: Laufzeiten Alternative Lösung des minimalen Modells mit direkter IMU Abhängigkeit.

| | | |
|---|---|---|
| BFSF | Runtime<br>Solution | - |
| CEDALION | Runtime<br>Solution | 0.35<br>1,2 |
| FDAUTOTUNE1 | Runtime<br>Solution | 0.08<br>1,2 |
| FDAUTOTUNE2 | Runtime<br>Solution | 0.08<br>1,2 |
| FDSS1 | Runtime<br>Solution | 0.42<br>1,2 |
| FDSS2 | Runtime<br>Solution | 0.28<br>1,2 |
| LAMA | Runtime<br>Solution | 0.26<br>1,2 |
| LAMA2011 | Runtime<br>Solution | 0.28<br>1,2 |
| RANDWARD | Runtime<br>Solution | 0.3<br>1,2 |
| UNIFORM | Runtime<br>Solution | 0.64<br>1,2 |

### 3.1.4 Überführung der Modelle in formale Constraints

Da gemäß den Ausführungen zuvor die vorhandenen für die Community typischen Solver wie PDDL keinen vielversprechenden Lösungsweg darstellen, wurde ein alternativer Solver für das gestellte Problem der Behandlung von Komponentennetzwerken gesucht. Andere Solver, wie beispielsweise HTN, scheinen jedoch explizit mit großen Problemklassen Schwierigkeiten zu haben. Dieses Verhalten wurde bereits in der Arbeit von Hartanto [25] beschrieben. Aus diesem Grund wurde von robotikspezifischen Planern abgewichen und zu einem constraintbasierten Verfahren gewechselt. Hierzu wurde Gecode als vielversprechender Basis-Solver gewählt. Gecode basiert auf der Arbeit von Schulte [55] und Erweiterungen wie Tack [61], Lagerkvist [39] sowie Schulte [56]. Gecode bietet im Gegensatz zu den zuvor vorgestellten Verfahren die Möglichkeit, Belegungen für verschiedene Variablentypen wie *bool, double, int* zu finden. Somit unterstützt Gecode im Gegensatz zu SAT-Solvern wie [10], die typischerweise nur *boolean* Variablen belegen können, mehr Problemklassen. Gecode ist ebenfalls in der Lage, mittels nutzergegebener Heuristiken Lösungen für globale Suchprobleme zu finden. Gecode ist jedoch domänenunabhängig entworfen, es besitzt keinen definierten Anwendungszweck wie beispielsweise PDDL, das auf Sequenz-Planung spezialisiert ist. Vielmehr stellt Gecode eine Infrastruktur für allgemeine constraintbasierte Planung dar. Die problem- oder gar domänenspezifische Lösung ist ein weiterer Schritt, der unabhängig von Gecode erfolgt.

Somit muss zunächst eine Modellierung der Anforderungen bei der Behandlung von Komponentennetzwerken auf Constraint-Basis erfolgen, bevor Gecode als Constraint Satisfaction Problem (CSP)-Solver genutzt werden kann. Diese Formalisierung muss numerisch auf die unterstützten Variablentypen und Mengenconstraints, die der Solver (hier Gecode) unterstützt, abbildbar sein. Die Erarbeitung und Erläuterung dieser Constraints wird der Kernbestandteil dieses Abschnittes 3.1.4 sein.

#### 3.1.4.1 Einleitung in Constraints

Die neu entwickelte Modellierung baut auf den Konzepten und Modellen von Syskit/Roby auf. Um jedoch die Modelle mittels CSP-Verfahren lösen zu können, muss eine Übersetzung von ihnen in formale Constraints stattfinden. CSP-Solver sind in der Lage, große Suchräume abzuarbeiten, um die definierten Constraints zu erfüllen. Ein einfaches Beispiel eines Constraints ist, dass für eine natürliche Zahl $x < 6$, aber zugleich $x > 3$ gelten muss. Die Lösung für dieses *Problem* ist offensichtlich, da $x$ mit den Werten $4, 5$ belegt werden kann. Somit gäbe es zwei *Lösungen* für das gestellte *Problem.* Die Suchräume können schnell sehr groß werden, was sowohl von der Art der Problemstellung als auch von den

Constraints selbst abhängt. Solver wie Gecode sind in der Lage, alle Lösungen für globale Suchprobleme zu liefern. Durch die veränderte Annahme der Syskit-Modelle (wie in 3.1.2 beschrieben) ergibt es sich, dass das Problem der Berechnung eines Komponentennetzwerkes ein globales Suchproblem ist. Dies resultiert aus der expliziten Erweiterung, dass Einflüsse zwischen den Komponenten bzw. Teilbäumen stattfinden dürfen. Durch die constraintbasierte Formalisierung ist es möglich, Anforderungen an andere Teilbäume zu stellen, wodurch ein Propagieren der Anforderungen entlang des Abhängigkeitsgraphs, wie es von Syskit realisiert wird, unzureichend ist. Im ersten Teil dieses Kapitels werden die zugrundeliegenden Modellkonzepte eines Komponentennetzwerkes formalisiert und aufgestellt. Diese Modelle werden im Weiteren mit Constraints belegt, die gelten müssen, damit das Komponentennetzwerk gültig ist. Jedes Constraint wird dabei erläutert und formal aufgestellt. Die Lösung, um ein Komponentennetzwerk zu bestimmen, ist in zwei Ebenen unterteilt. Eine der Ebenen ist die sogenannte *Klassenlösung* (Seite 98), die sämtliche Kinder einer konkreten Algorithmusklasse zuordnet. Dieser Schritt ermittelt für alle abstrakten DataServices konkrete Algorithmen bzw. Tasks, die die Anforderungen erfüllen. Die zweite Ebene zur Gesamtlösung ist die *Instanzlösung* (Seite 118). Sie bestimmt die Zuordnung der Algorithmen zu konkreten Instanzen und nimmt ihre Konfiguration vor.

#### 3.1.4.2 Einführung constraintbasierter Modelle

Im Folgenden werden die Modelle beschrieben, die dem neuen Verfahren zugrunde liegen. Dabei handelt es sich zum Teil um formalisierte Modelle, die aus den Syskit/Roby-Konzepten stammen, wie auch um weitere neue Attribute. Weiterhin sind Hilfsattribute für den Prozess der constraintbasierten Lösung vonnöten, die detailliert erläutert werden.

Zunächst wird die Menge aller vorhandenen Tasks benötigt. Konzeptuell ist ein Task ein Blatt im Abhängigkeitsgraph. Ein Task kann keine weiteren Abhängigkeiten haben und ist, nach der finalen Lösung, ein laufender Algorithmus auf dem Robotersystem.

***Definition 1*** *Sei $T$ Menge aller vorhandenen Tasks:*

$$T = \{t_1, \dots, t_n\} \tag{3.1}$$

*Wobei $n \in \mathbb{N}^+$ die Anzahl aller vorhandenen Tasks ist.*

Des Weiteren wird die Menge aller vorhandenen Kompositionen benötigt. Eine Komposition ist ein Verbund aus verschiedenen anderen Entitäten und kann mehrere Kinder haben. Somit ist sie ein Knoten innerhalb des Abhängigkeitsgraphs. Eine Komposition selbst kann kein Blatt sein, da sie nur innerhalb der Modellierung eine Funktion besitzt.

Es gibt keinen unterliegenden Algorithmus zu ihr. Sie ist lediglich ein Hilfsmittel zur Modellierung komplexer Systeme.

***Definition 2*** *Sei $C$ Menge aller vorhandenen Kompositionen:*

$$C = \{c_1, \ldots, c_n\} \tag{3.2}$$

*Äquivalent zu der Anzahl der Tasks ist $n \in \mathbb{N}^+$ die Anzahl der definierten Kompositionen.*

Um die Abstraktion zwischen verschiedenen Systemen zu ermöglichen, werden sogenannte DataServices definiert. DataServices sind abstrakte Lieferanten von Daten. Sie haben jedoch genauso wie Kompositionen keine aktive Funktion. Die DataServices müssen zu konkreten Kompositionen oder Tasks aufgelöst werden, die die Rolle des DataServices erfüllen.

***Definition 3*** *Sei $D$ eine Menge aller bekannten DataServices:*

$$D = \{d_1, \ldots, d_n\} \tag{3.3}$$

*Wobei $n \in \mathbb{N}^+$ die Anzahl der bekannten DataServices ist, die in dem Modell vorhanden sind.*

Das Ziel der Modellierung ist es, ein möglichst flexibles System aufzubauen. Dabei ist eine Entität im Netzwerk ein DataService, ein Task oder eine Komposition. Eine Komposition kann eine oder mehrere Entitäten beinhalten. Dadurch baut sich eine hierarchische Struktur von Modulen auf.

***Definition 4*** *Die Menge aller Entitäten $E$ setzt sich zusammen aus:*

$$E = T \cup C \cup D \tag{3.4}$$

***Definition 5*** *Sei $\Phi(c)$ eine Menge aller Abhängigkeiten der Komposition $c$:*

$$\Phi(c) = [e_1, \ldots, e_n] \tag{3.5}$$

*Wobei ein Kind $e \in E$, die Anzahl und der Typ des Kindes ein Bestandteil des Modells sind.*

Die Definition 5 legt die Kinder einer Komposition fest. Diese Funktion $\Phi(c)$ ist somit der Hauptbestandteil der hierarchischen Modellierung selbst. Die Kinder einer Komposition selbst können wieder Kompositionen, DataServices oder Tasks sein. Die Herausforderung besteht während der *Klassenlösung* darin, die abstrakten Kinder, also die *DataServices*, jeweils konkreten Kindern, also *TaskContexts* oder *Kompositionen* zuzuordnen. Es sei nochmal erwähnt, dass die DataServices die Grundlage der Abstraktion bilden und somit der Schlüssel zur flexiblen und wiederverwendbaren Modellierung sind.

#### 3.1.4.3 Constraint schritt eins - Die Klassenlösung

Wie in der Einführung erwähnt, ist es das Ziel der *Klassenlösung* zu bestimmen, welche konkreten Algorithmenklassen für die Aufgaben der DataServices genutzt werden können. Es liegt demnach die Absicht vor, alle DataServices aus $\phi(c)$ aufzulösen und durch konkrete Tasks oder Kompositionen zu ersetzen. Das Ergebnis der Klassenlösung ist ein Abhängigkeitsgraph, der definiert, welche Kompositionen welche Entitäten nutzen. Die Blätter des Abhängigkeitsgraphs sind dabei Tasks, die Knoten sind Kompositionen. Alle Tasks in $T$, die keine Verbindungen haben, sind für ein konkretes Netzwerk unnötig und folglich nicht aktiv, da sie nicht benötigt werden.

Es ist zu beachten, dass ein Task $t \in T$ mehrfach innerhalb des Abhängigkeitsgraphs vorkommen kann. Dieses mehrfache Vorkommnen definiert jedoch nicht, ob es später zur Ausführung auch zwei Instanzen des Tasks gibt. Diese Bestimmung ist Teil der *Instanzlösung*, die im folgenden Kapitel zur Instanzlösung (ab Seite 118) beschrieben wird.

Für die Bestimmung der Klassenlösung werden verschiedene Hilfsmengen bzw. Vektoren benötigt, die zur Berechnung der Lösung erforderlich sind. Das Ziel ist es, die Elemente der Mengen soweit einzugrenzen, dass möglichst nur ein Wert gültig ist oder nur ein Element der Menge übrigbleibt.

Die erste Hilfsmenge, die zur Bestimmung der Klassenlösung benötigt wird, ist die Menge der aktiven Elemente.

***Definition 6*** *Sei $A$ ein Vektor von booleschen Variablen:*

$$A = [a_1, \dots, a_{|E|}] \tag{3.6}$$

*wobei jedes Element $a_i \in \{0, 1\}$ angibt, ob die korrespondierende Entität $e_i$ aktiv ist oder nicht.*

Dieser Vektor von Elementen dient als Basis zur Bestimmung der aktiven Komponenten innerhalb des Komponentenpools. Er alleine ist noch ohne jede Funktion. Im weiteren

Verlauf werden diverse Constraints auf ihm und zwischen anderen Hilfselementen definiert. Ein CSP-Solver wäre lediglich auf dieser einen Menge nicht in der Lage, den Suchraum weiter einzuschränken. Das Ziel ist es, klar für jedes $a_i$ bzw. das korrespondierende Element $e_i$ zu definieren, ob es aktiv ($a_i = 1$) oder inaktiv ($a_i = 0$) ist.

***Definition 7*** *Sei $F$ die Menge aller umgekehrten Abhängigkeiten einer Komponente. Gegeben sei weiterhin $f \subset E$, dann ist die Menge der Abhängigkeiten aller Komponenten definiert durch:*

$$F = [f_1, \dots, f_{|E|}] \tag{3.7}$$

*wobei ein Element $f_i \subset E$ eine Teilmenge aller möglichen Komponenten ist. Das Element $f_i$ repräsentiert die Abhängigkeiten für die Komponente $e_i$.*

Die Menge der *umgekehrten* Abhängigkeiten der Komponente gibt an, wer ihre Eltern im Abhängigkeitsgraph sind. Als Beispiel sei gegeben $f_5 = \{e_3, e_8\}$, dies würde bedeuten, dass die Komponente $e_5$ als Eltern die Komponenten $e_3$ sowie $e_8$ besitzt. Im Umkehrschluss bedeutete es, dass die Komponenten $e_3, e_8$ die Komponente $e_5$ als Kind besitzen. Dies ist dann der Fall, wenn $\{e_3, e_8\} \in C$ und $e_3, e_8$ die Komponente $e_5$ als direktes Kind benötigen oder die Komponente $e_5$ als Ersatz für einen DataService $d \in D$ gewählt wurde.

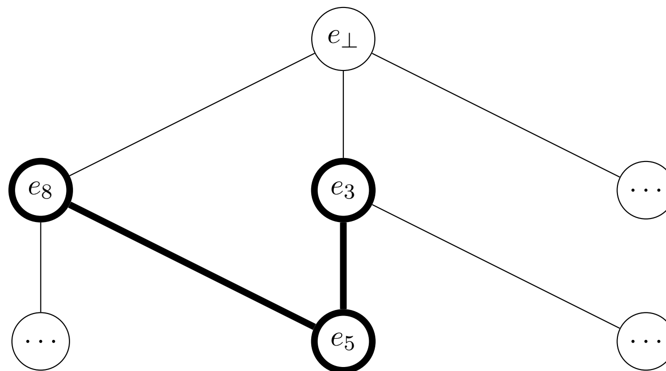



Figure 3.4: Verdeutlichung der umgekehrten Abhängigkeiten $f_5 = \{e_3, e_8\}$

***Definition 8*** *Sei $H$ die Menge aller rekursiven umgekehrten Abhängigkeiten aller Komponenten:*

$$H = [h_1, \dots, h_{|E|}] \tag{3.8}$$

*wobei ein Element $h_i \subset E$ die Teilmenge aller möglichen Komponenten ist. Das Element $h_i$ repräsentiert die rekursiven umgekehrten Abhängigkeiten für die Komponente $e_i$ (analog zu 7)*

Rekursive umgekehrte Abhängigkeiten beschreiben alle Knoten, die eine Komponente auf dem Weg zum Wurzelknoten $e_\perp$ hat. Somit folgt, dass $h_i \subseteq f_i$ gelten muss.

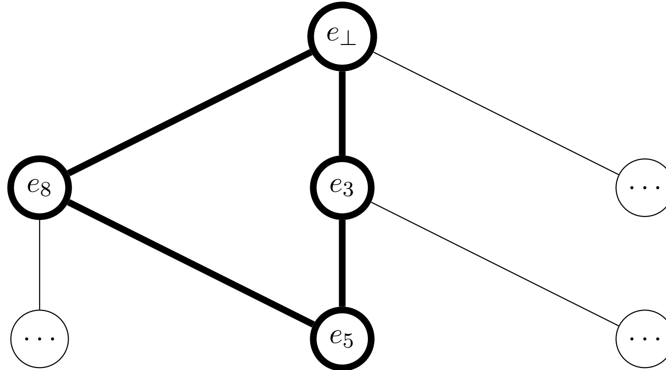



Figure 3.5: Verdeutlichung der umgekehrten Abhängigkeiten $h_5 = \{e_3, e_8, e_\perp\}$

Bei den Formalisierungen sind die folgenden Konventionen zu beachten:

- Die Indizes (wenn in einem Absatz verwendet) beschreiben das gleiche Element.
  $h_5$ beschreibt somit das Element $e_5$. Dies gilt ebenso für die Mengen: $A, T, C, D, F$ und (wie angeführt) $H$.

- Subscript-Operator zwischen Elementen und Mengen.
  Gegeben sei eine Komponente $e \in E$, dann bezeichnet $a_e$ das korrespondierende Element $a_e \in A$, das die Komponente $e$ repräsentiert.

- Die Komponente $e_\perp$ definiert die Wurzelkomponente.
  Alle anderen Komponenten müssen direkt oder indirekt von ihr abhängen, um aktiv zu sein. Diese Komponente $e_\perp \in C$ bildet die Wurzel im Abhängigkeitsgraph aller aktiven Komponenten. Die Wurzelkomponente muss immer aktiv sein. Das ist zugleich das erste Constraint, das im Folgenden definiert wird.

- Komponente $e_\emptyset$ ist die NIL-Komponente.
  Diese wird benötigt, um zu verdeutlichen, dass keine Zuordnung zu einer speziellen Instanz stattfindet. Das kann explizit gewünscht oder ein Hinweis darauf sein, dass die Elternkomponente inaktiv ist. Diese Komponente wird als $e_\emptyset \in T$ bezeichnet.

**Beginn der Definition der Constraints** Ab diesem Punkt werden die Constraints (bzw. Einschränkungen) beschrieben, die gelten müssten, damit die Berechnung des Komponentennetzwerkes erfolgen kann. Der Constraint-Solver wird anhand dieser Constraints versuchen, eine gültige Belegung der Mengen $A, F, H$ sowie der später eingeführten Abbildung $\phi(c)$ zu finden. Die Constraints sind somit die Formalisierung der constraintbasierten Komponentennetzwerkbehandlung. Ein Netzwerk ist nur dann gültig, wenn alle Constraints erfüllt sind. Der Constraint-Solver kann hierbei keine, eine oder auch mehrere Lösungen (bzw. Belegungskombinationen) finden.

- Keine Lösung
  Es ist nicht möglich, eine Belegung zu finden, die den Anforderungen entspricht. Dies kann im Beispiel der Klassenlösung dann der Fall sein, wenn es keine Algorithmen gibt, die die Anforderungen erfüllen können. Beispielsweise gibt es auf einem System keine IMU. In der Folge sind dann typischerweise keine Orientierungen verfügbar. Des Weiteren könnte eine fehlerhafte Modellierung vorhanden sein, indem Zyklen innerhalb des Abhängigkeitsgraphen entstehen, die jedoch verboten sind. Mehr zu Zyklen, später in Definition 25 (Seite: 108).
- Eine Lösung
  Der CSP-Solver hat eine Lösung für ein Problem identifiziert. Jeder abstrakten Anforderung konnte genau eine konkrete Komponente zugeordnet werden. Die erstellte Klassenlösung entspricht dabei allen Anforderungen. Es sind keine Mehrdeutigkeiten vorhanden. In diesem Fall ist die Lösung äquivalent zu der Lösung, die das Syskit bzw. Roby-System ermitteln würde.
- Mehrere Lösungen
  Es wurden mehrere gültige Belegungen für DataServices gefunden. Dies kann dann der Fall sein, wenn es mehrere Algorithmen oder Geräte gibt, die einer oder mehreren Anforderungen entsprechen. Das kann ein Fehler in der Modellierung oder ein Indiz für eine höhere Fehlersicherheit sein, da redundante Geräte oder Algorithmen zur Verfügung stehen.

Das erste Constraint, das definiert wird, wird im Folgenden dargelegt. Diese sowie alle weiteren Constraints bestehen aus einem Ausdruck, der vom Constraint-Solver gelöst werden muss.

***Definition 9*** *Gegeben sei $a_\emptyset$, die NIL-Komponente*

$$a_\emptyset = 0 \tag{3.9}$$

*wobei gelten muss, dass $e_\emptyset$ niemals aktiv sein darf.*

Die NIL-Komponente ist ein virtuelles Konstrukt der Modellierung, um eine Belegung von Abhängigkeiten definieren zu können, die im Abhängigkeitsgraphen nicht benötigt Werden, jedoch formal eine Belegung durch den CSP-Solver benötigen. Jede Komponente muss unabhängig davon, ob sie später im System aktiv sein wird, durch den Constraint-Solver identifiziert werden. Es ist nicht möglich zu definieren, dass bestimmte Variablen für die Lösung irrelevant sind.

Das zweite Constraint definiert, dass alle Komponenten, die virtuell sind, nicht lauffähig sein können. Diese sind Hilfskonstruktionen, um die Abstraktion zu ermöglichen, bilden aber keine lauffähige Programminstanz, die Arbeit auf einem System verrichten kann.

***Definition 10*** *Gegeben sei, dass alle virtuellen Komponenten (DataServices) nicht lauffähig sind:*

$$\forall e \in E, e \in D : \quad a_e = 0 \tag{3.10}$$

$$\forall e \in E, e \in D : \quad f_e = \{\emptyset\} \tag{3.11}$$

*wobei nicht lauffähige Komponenten weder aktiv sein noch umgekehrte Abhängigkeiten (zum Pfad des Wurzelknotens) haben dürfen.*

Die Definition 3.10 besagt, dass virtuelle Komponenten wie DataServices nicht aktiv sein dürfen. Dies ist der Tatsache geschuldet, dass DataServices keine unterliegenden Algorithmen besitzen. DataServices sind Hilfskonstrukte der Modellbeschreibung, um die lose Kopplung zwischen Komponenten zu ermöglichen. Die Definition 3.11 schließt aus, dass sie im Abhängigkeitsgraph auftauchen dürfen. Da sie keine Eltern besitzen dürfen, dürfen sie im Graphen auch nicht vorkommen.

Das Constraint 3.12 definiert, dass alle Komponenten, die aktiv sind, auch aktiv sein sollten. Dieses trifft immer auf die Wurzelkomponente zu.

***Definition 11*** *Gegeben sei, dass sie Wurzelkomponente $e_\perp$ immer aktiv sein muss:*

$$e_\perp = 1 \tag{3.12}$$

Auch wenn der Wurzelknoten des Abhängigkeitsgraphen keine reale Komponente ist, sondern als Hilfskonstrukt der Modellierung fungiert, so bildet er dennoch die Wurzel des Baums und muss somit immer aktiv sein. Sollte diese virtuelle Komponente nicht aktiv sein, läge ein Fehler im Netzwerk oder dessen Berechnung vor. Diese Fehler können jedoch später für die Verhaltenssequenzierung, wie in Kapitel 3.1.4.5 beschrieben, genutzt werden.

***Definition 12*** *Gegeben sei, dass die Wurzelkomponente $e_\perp$ keine weiteren Eltern haben darf:*

$$h_\perp = \{\emptyset\} \tag{3.13}$$

Die Definition 3.13 beschreibt und legt fest, dass die Wurzelkomponente von nichts anderem abhängen darf. Es wäre ansonsten möglich, dass der CSP-Solver andere (auch inaktive Komponenten) als Eltern der Wurzel definieren könnte, da es keine anderweitigen Regeln gibt, die dies verhindern würden. Es ist zu beachten, dass diese Einschränkung nicht verbietet, dass andere Komponenten die Wurzelkomponente als Elternknoten besitzen, was offensichtlich nötig ist.

***Definition 13*** *Gegeben sei, dass keine Komposition von sich selbst abhängen darf:*

$$\forall c \in C: \quad c \notin f_c \tag{3.14}$$

Dieses verhindert in erster Linie, dass es keine Zyklen im Abhängigkeitsgraph geben darf. Diese Bedingung ist an dieser Stelle nicht rekursiv. Sie dient im Vergleich zu späteren Einschränkungen nur der Reduktion des Suchraums. Diese Einschränkung ist auch im Graph 3.6 dargestellt.

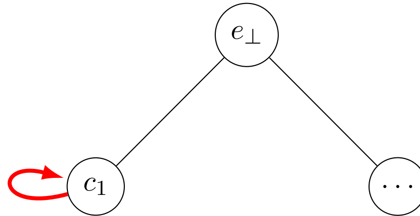



Figure 3.6: Verbildlichung der Bedingung das eine Komposition nicht von sich selbst abhängen darf (siehe Def 3.14)

***Definition 14*** *Es sei gegeben, dass der NIL-Task $e_\emptyset$ niemals als Elternknoten für eine andere Komponente selektiert werden darf, weder direkt noch rekursiv.*

$$\forall c \in C: \quad e_\emptyset \notin f_c \tag{3.15}$$
$$\forall c \in C: \quad e_\emptyset \notin h_c \tag{3.16}$$

Da der NIL-Task keine Abhängigkeiten hat und selbst nicht aktiv sein darf, darf er keine Kinder besitzen. Diese Einschränkung des Suchraums ist im Graph 3.7 verdeutlicht.

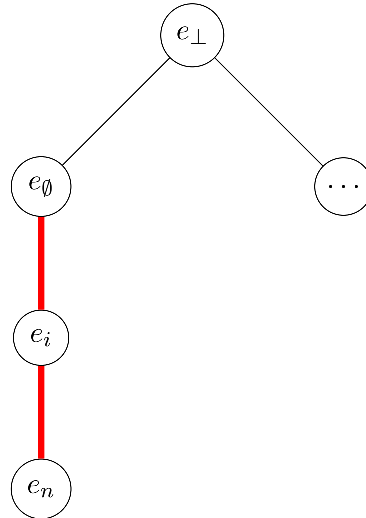



Figure 3.7: Verbildlichung der Bedingung 3.15 und 3.16, der NIL-Task $e_\emptyset$ darf niemals im Graphen (als Abhängigkeit) auftauchen.

***Definition 15*** *Gegeben sei, dass der NIL-Task nichts anderes als Elternknoten besitzen darf und dass der NIL-Task keine Abhängigkeiten haben darf*

$$f_{e_\emptyset} = \{\emptyset\} \tag{3.17}$$
$$h_{e_\emptyset} = \{\emptyset\} \tag{3.18}$$

Durch die Bedingungen 3.15, 3.16, 3.17, 3.18 wird sichergestellt, dass der NIL-Task nicht im Abhängigkeitsgraph erscheinen kann. Dies wird dadurch erreicht, dass die NIL-Komponente weder als Elternknoten dienen kann noch etwas anderes als Elternknoten besitzt. Die Exklusion der rekursiven Abhängigkeiten dient dabei nur als optimierende Einschränkung für den Constraint-Solver, um den Suchraum schneller verkleinern zu können.

***Definition 16*** *Gegeben sei $\phi(c)$, der Assoziationsvektor von Mengen der Kinder der Komposition $c$:*

$$\phi(c) = [E_0, \ldots, E_{|\Phi(c)|}] \tag{3.19}$$

*wobei $|\Phi(c)|$ die Anzahl der ursprünglichen Kinder angibt. Jedes Kind kann zunächst jede mögliche Entität aus $E$ sein.*

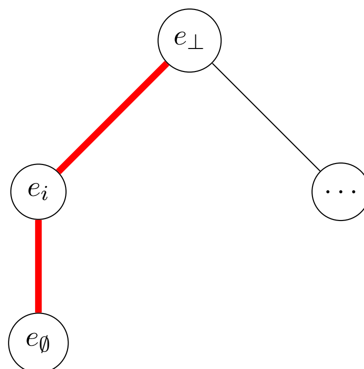



Figure 3.8: Verbildlichung der Bedingung 3.17 und 3.18

Der Assoziationsvektor $\phi(c)$ behandelt, analog zu $\Phi(c)$, die Zuordnung von Kindentitäten $e$ zu einer Komposition $c$. Im Gegensatz zu $Phi$ ist die Zuordnung jedoch variabel. Wie zu Beginn dieses Kapitels angeführt, ist eine Hauptaufgabe das Identifizieren von konkreten Kindentitäten. Die abstrakten Kinder, wie DataServices, müssten zu konkreten Elementen aufgelöst werden. Um dieses Problem zu reformulieren, wird angenommen, dass jede mögliche Kindentität $e \in E$ eine mögliche Belegung für die Kindentität ist. Im Folgenden werden iterativ zusätzliche Constraints definiert, die den möglichen Lösungsraum weiter einschränken, um am Ende der CSP-Berechnung nur noch valide Zuordnungen zu erhalten.

***Definition 17*** *Gegeben sei, dass eine Komposition $c \in C$, die keine Kinder hat, nicht als Elternknoten selektiert werden darf.*

$$\forall c \in C, \forall e \in E : \quad |\psi(c)| = 0 : c \notin f_e \tag{3.20}$$

Dieses Constraint verbietet, dass leere Kompositionen (falls sie fehlerhafterweise im Modell vorkommen) als Elternteil für Kinder selektiert werden dürfen. Dies liegt darin begründet, dass leere Kompositionen keine Aufgabe erfüllen können.

***Definition 18*** *Gegeben sei, dass alle Kindabhängigkeiten einer Komposition $c \in C$ bestimmt sein müssen (also nicht der leeren Komponente zugeordnet sind), wenn die Komposition $c$ aktiv ist, also $a_c = 1$ gilt.*

$$\forall c \in C, \forall j \in \phi(c) : \quad a_c = 1 \Rightarrow j \neq e_\emptyset \tag{3.21}$$

*wobei $j$ die Menge der möglichen Kindzuordnungen von $c$ ist. Ein Graph, der dieses Constraint darstellt, ist in 3.9 zu sehen.*

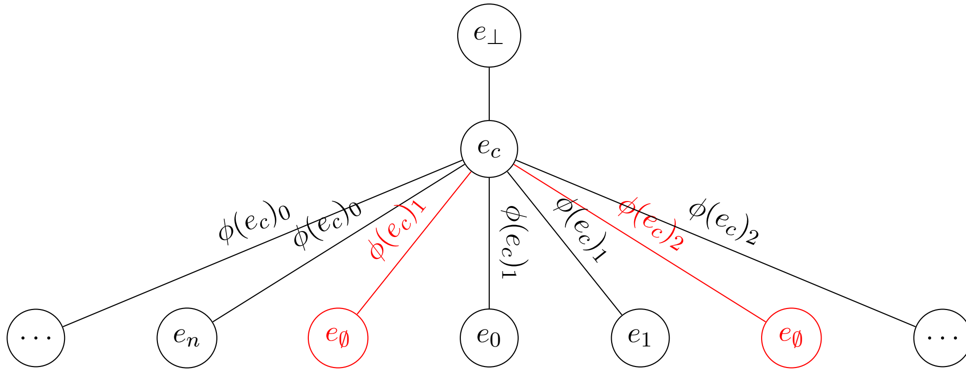



Figure 3.9: Eine Kindzuordnung darf nicht der leeren Menge entsprechen, es muss eine konkrete Zuordnung gefunden werden. (Vergl. Definition 3.21)

***Definition 19*** *Gegeben sei, dass eine Komposition $c$ nicht sich selbst als Kind $c \neq \phi(c)$ selektieren darf:*

$$\forall c \in C, \forall j \in \phi(c): \quad j \neq c \tag{3.22}$$

Diese Bedingung ist ähnlich wie 3.14, jedoch definiert dieses Constraint nicht die Abhängigkeitsbeziehung $f_c$, sondern die Kindselektion $\phi(c)$. Die Kinder $\phi(c)$ definieren somit, welche Komponenten von $c$ abhängen. $\phi(c)$ sind somit die nächsten absteigenden Knoten im Abhängigkeitsgraph von $c$.

***Definition 20*** *Gegeben sei die Definition, dass alle Kinder einer Komposition dem NIL-Task entsprechen, wenn diese inaktiv ist. Siehe auch Graph 3.10.*

$$\forall c \in C, \forall j \in \phi(c): \quad a_c = 0 \Rightarrow j = e_\emptyset \tag{3.23}$$

***Definition 21*** *Gegeben sei, dass alle Komponenten, die aktiv sein sollen, die Wurzelkomponente als rekursive umgekehrte Abhängigkeit beinhalten müssen.*

$$\forall c \in C, c \neq c_1: \quad a_c = 1 \Rightarrow c_1 \in h_c \tag{3.24}$$

Die Definition 3.24 zählt zu den wichtigsten Definitionen. Sie definiert, dass alle Komponenten, die aktiv sind, einen Pfad zur Wurzelkomponente benötigen (Graph 3.11). Mittels

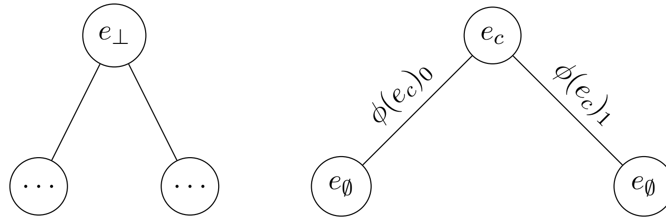



Figure 3.10: Alle Kinder inaktiver Komponenten sind der NIL-Task (20)

dieser Bedingung wird zugleich erzwungen, dass von der eigentlichen Wurzelkomponente losgelöste Verbunde inaktiv sein müssen, da sie keine Verbindung zum Wurzelknoten besitzen.

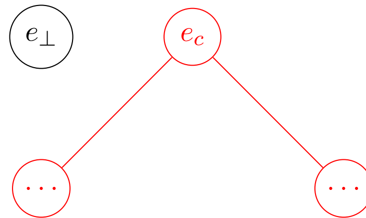



Figure 3.11: Durch die Definition 20 wird erzwungen, dass alle losgelösten Komponenten inaktiv sind (hier rot dargestellt)

***Definition 22*** *Es muss definiert werden, ob ein Kind innerhalb einer Komposition benötigt wird. Dazu wird die Hilfsabbildung* $used(e, c)$ *definiert:*

$$used(e, c) := 0, 1 \tag{3.25}$$

*Wobei* $e \in E$ *und* $c \in C$ *gilt. Das Ergebnis bildet die Menge* $0, 1$. *Das Ziel ist es, falls eine Komponente* $c$ *als Kind genutzt wird, dass* $used(e, c) = 1$ *gilt, andernfalls* $used(e, c) = 0$.

***Definition 23*** *Gegeben sei die Abbildung* $used(e, c)$. *Diese liefert 1 (true), wenn das Element e in den Kindern phi der Komposition c vorkommt, und 0 (false) andernfalls:*

$$used(e, c) = \begin{cases} 1, & \textit{if } e \in \phi(c) \\ 0, & \textit{otherwise} \end{cases} \tag{3.26}$$

***Definition 24*** *Für das Komponentennetzwerk muss gelten, dass alle Komponenten, die nicht direkt benutzt werden, nicht als direkte rekursive Abhängigkeit diese Komponente besitzen dürfen:*

$$\forall c \in C, \forall e \in E: \quad !used(e,c) \Leftrightarrow c \notin f_e \tag{3.27}$$

Die Definitionen 22 bis 24 bilden die komplexe Definition, wann eine Komponente als direkte Abhängigkeit auftreten darf und wann nicht. Man beachte die logische Äquivalenz ($\Leftrightarrow$) aus 24. Diese bestimmt, dass ein Kind in den Abhängigkeiten vorkommen muss, wenn es benutzt wird (durch $used(e,c)$ definiert). Der Umkehrschluss gilt jedoch ebenso. Wenn ein Kind nicht genutzt wird, dann darf es auch nicht in den rekursiven Abhängigkeiten auftreten. Des Weiteren gilt, wenn es eine Abhängigkeit geben sollte, muss es als Kind genutzt werden. Falls es als Kind nicht vorkommt (oder vorkommen kann), kann es auch nicht als Abhängigkeit genutzt werden.

***Definition 25*** *Gegeben sei, dass eine Komponente niemals rekursiv von sich selbst abhängen darf:*

$$\forall e \in E: \quad e \notin h_e \tag{3.28}$$

Die Definition 25 legt fest, dass es keine Zyklen innerhalb des Graphs geben darf. Für einen Abhängigkeitsgraph ist dieses Verhalten intuitiv eindeutig. Zwar ist die Einschränkung - Zyklen innerhalb von Komponentenverbunde zu verbieten - nicht immer intuitiv wünschenswert, jedoch formal korrekt und strikt nötig. Um dieses Konzept zu erläutern, wird das folgende Szenario definiert:

Gegeben sei eine Sensorfusionskomponente $t_s \in T$, eine IMU $t_i \in T$ und eine FOG $t_f \in T$, ein DataService $d_o \in D$, der eine Orientierung liefert, sowie ein DataService $d_r$, der eine 1D-Rotation (wie sie der FOG bietet) repräsentiert. Des Weiteren existiert eine Komposition $c \in C$, die die IMU und den FOG zu einer höherwertigen Orientierung zusammenfügen kann. Die Komposition besteht aus einer IMU und einer FOG sowie der konkreten Komponente, die die Fusion übernimmt. Daraus entsteht das folgende hypothetische Modell:

$$C = \{c\} \quad (3.29)$$
$$T = \{t_s, t_i, t_f\} \quad (3.30)$$
$$\Phi(c) = \{t_s, d_o, d_r\} \quad (3.31)$$
$$\succ (c, d_o) = 1 \quad (3.32)$$
$$\succ (c, d_r) = 1 \quad (3.33)$$
$$\succ (t_i, d_o) = 1 \quad (3.34)$$
$$\succ (t_f, d_r) = 1 \quad (3.35)$$
$$(3.36)$$

Gegeben sei weiterhin die Anforderung, den DataService $d_o$ an einer beliebigen Stelle des Netzwerkes zu erfüllen. Hieraus ließen sich, sofern die Definition 25 nicht aufgestellt würde, folgende Lösungen bestimmen.

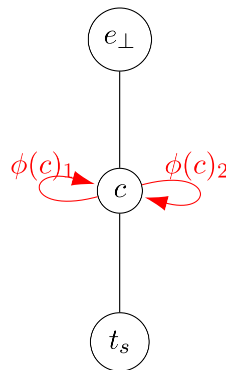



Figure 3.12: Ungültige Lösung, da die Komposition sich selbst als Bereitsteller für Orientierung und Rotation gewählt hat.

Die vorgestellten Lösungen 3.12, 3.13, 3.14 zeigen deutlich, warum Zyklen innerhalb der Modelle von Komponentennetzwerken nicht vorkommen dürfen. Es wäre möglich, dass die Komponente invalide Daten produziert. Wie die Beispiele zeigen, existieren dennoch zwei valide Lösungen. Die Lösungen 3.15 und 3.16 erfüllen beide die Anforderungen. Zwar würde die Lösung 3.16 vermutlich nicht die besten Daten liefern, jedoch ist sie technisch eine gültige Erfüllung der Anforderungen.

***Definition 26*** *Es muss gelten, dass alle Abhängigkeiten einer Komponente in den rekur-*

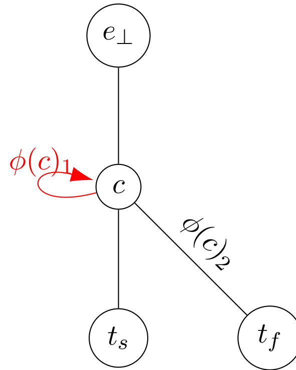



Figure 3.13: Ungültige Lösung, da die Komposition sich selbst als Bereitsteller für Orientierung gewählt hat.

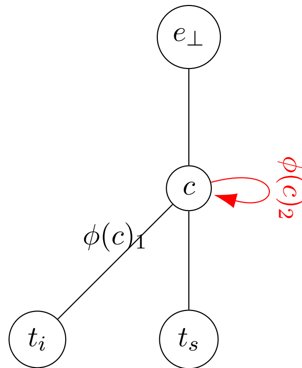



Figure 3.14: Ungültige Lösung, da die Komposition sich selbst als Bereitsteller für die Rotation gewählt hat.

*siven Abhängigkeiten enthalten sind, sofern die Komponente selbst benutzt wird:*

$$\forall e \in E, \forall c \in C : \quad used(e, c) = 1 \Rightarrow (h_e = h_c \cup f_e) \tag{3.37}$$

Die Definition 26 legt die Zusammensetzung aller rekursiven Abhängigkeiten einer Komponente fest, bzw. dass alle direkten Abhängigkeiten einer Komponente auch in den rekursiven Abhängigkeiten vorkommen müssen. Dadurch ist jedoch noch nicht definiert, welche Komponenten nicht in den rekursiven Abhängigkeiten vorkommen dürfen. Dies wird in den weiteren Abschnitten erläutert.

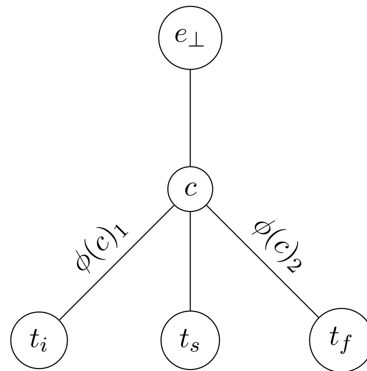



Figure 3.15: Gültige Lösung, es existieren keine Zyklen innerhalb des Graphs.

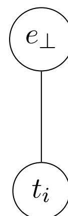



Figure 3.16: Gültige Lösung, jedoch ggf. nicht optimal.

***Definition 27*** *Es gilt, dass DataServices nicht als (konkrete) Kinder genutzt werden können:*

$$\forall c \in C, \forall k \in \phi(c), \forall d \in D : \quad k \neq d \tag{3.38}$$

$$\forall c \in C, \forall d \in D : \quad d \notin f_c \tag{3.39}$$

$$\forall c \in C, \forall d \in D : \quad used(d, c) = 0 // \tag{3.40}$$

*wobei sie weder direkt noch im Abhängigkeitsgraph als Kind auftreten dürfen.*

***Definition 28*** *Gegeben sei eine Abbildung $\succ (e_1, e_2)$, die beschreibt, ob eine Komponente eine Anforderung erfüllen kann:*

$$\succ (e_1, e_2) \in 0, 1 \tag{3.41}$$

Die Definition 3.41 ist eine externe Eingabe, also Modellwissen. Sie wird von der Person definiert, die die Modelle des Netzwerkes implementiert. Ein abstraktes Beispiel hierfür wäre $\succ (imu, orientation) = 1$, was beschreibt, dass eine $imu \in T$, eine $orientation \in D$ bereitstellen kann. Dieses Wissen wird nicht über Constraints bestimmt oder gebildet, es ist ein explizites Wissen, das zum Zeitpunkt des CSP-Lösens bereitstehen muss.

***Definition 29*** *Es sei gegeben, dass für alle Kompositionen $c \in C$, all ihre Kinder $j \in \phi(c)$ und alle Entitäten $e \in E$ gilt, dass:*

$$\forall c \in C, \forall j \in \phi(c), e \in E : \begin{cases} e = j \Rightarrow f_e \supseteq \{c\}, & \textit{if } \succ (e, j) = 1 \\ j \neq e, & \textit{if } \succ (e, j) = 0 \wedge e \neq e_\emptyset \end{cases} \tag{3.42}$$

Diese Definition 29 beschreibt die Zuordnung von Kindern zu Kompositionen und ihre Effekte. Der zweite Teil der Klausel beschreibt, dass die Komposition, wenn keine Entität $e$ für ein aktuell gegebenes Kind $j$ von $c$ gewählt wird, $c$ in den umgekehrten Abhängigkeiten von $e$ als Elternknoten vorkommen muss, sofern die Entität $e$ die Anforderungen des Kindes $j$ erfüllen kann. Bedingung 3.42 hingegen beschreibt, dass die gegebene Entität $e$ nicht als Kind $j$ der Komposition $c$ gewählt werden darf, dies ist dann der Fall wenn $e$ nicht die Anforderungen von $j$ erfüllen kann und es sich bei $e$ nicht um die NIL-Entität $e_\emptyset$ handelt. Diese Behandlung des NIL-Tasks ist nötig, da bedingt durch die früheren Constraints (wie 3.21) die Zuordnung der NIL-Komponente immer möglich ist, sofern die Elternkomponente $c$ nicht aktiv ist.

***Definition 30*** *Gegeben sei, dass alle inaktiven Komponenten keine Abhängigkeiten haben und nichts von ihnen abhängt:*

$$\forall e \in E: \quad a_e = 0 \Rightarrow f_e = \emptyset \tag{3.43}$$

$$\forall e \in E: \quad a_e = 0 \Rightarrow h_e = \emptyset \tag{3.44}$$

$$\forall e \in E \land e \neq e_\perp: \quad h_e = \emptyset \Leftrightarrow a_e = 0 \tag{3.45}$$

Somit beschreibt die Bedingung 3.43, dass alle nicht aktiven Komponenten keine umgekehrten Abhängigkeiten haben dürfen. Die Bedingung 3.44 definiert diese Einschränkung analog dazu für alle rekursiven Abhängigkeiten. Konzeptionell würde die Exklusion der rekursiven Abhängigkeiten ausreichen, jedoch erleichtert die explizite Definition die Reduktion des Suchraums zu einem früheren Zeitpunkt. Das Constraint 3.45 erweitert diese Annahme um eine Äquivalenz für alle Knoten, die nicht der Wurzelknoten $e_\perp$ sind. Es wird gefordert, dass alle nicht aktiven Knoten keine reversiblen Abhängigkeiten haben und somit im Baum nicht vorhanden sind. Der Wurzelknoten muss hier ausgenommen werden, da er selbst keine Eltern im Graph besitzt. Im Umkehrschluss bedeutet dies aber auch, dass alle aktiven Komponenten einen Elternknoten besitzen müssen.

**Suchverfahren** Die im vorherigen Abschnitt aufgestellten Constraints bilden die Grundlage zur Berechnung der Klassenlösung. Der Suchraum der Möglichkeiten ist aber nach wie vor enorm groß. CSP-Löser suchen, wie bereits zuvor erwähnt, nach gültigen Belegungen für Variablen und steigen dann innerhalb ihres internen Suchbaum weiter ab, um die fehlenden Constraints zu überprüfen. Falls ein Teilbaum zu keiner Lösung führt, wird das Backtracking-Verfahren angewendet und in einem anderen Teilbaum weitergesucht. Dies ist auch der Fall, falls ein Teilbaum ein gültiges Ergebnis geliefert hat. In diesem Fall wird durch Backtracking die nächste Belegung einer Variable evaluiert, was einen neuen gültigen Teil-Suchbaum generiert. Die Selektion, welche Variable mit welchem Wert belegt wird, formt den Suchbaum und ist die Heuristik, die die Suche optimieren kann. Für die Berechnung der Klassenlösung wird eine adaptive Heuristik verwendet, die die zu belegenden Variablen entlang des umgekehrten Suchbaums belegt. Erst am Ende, wenn der Abhängigkeitsgraph gebildet wurde, werden alle noch offenen Variablen explizit belegt. Das hat zur Folge, dass der Abhängigkeitsgraph iterativ während der CSP-Lösung generiert wird. Dieses Vorgehen hat sich als das performanteste herausgestellt. Im Folgenden werden die Verzweigungs-Bedingungen (branch conditions) festgelegt. Ein Zweig ist dabei die Selektion einer oder mehrerer Variablen mit einer Belegungsempfehlung.

***Definition 31*** *Gegeben sei als Startpunkt die Entscheidung, ob die Wurzelkomponente aktiv sein soll. Im Anschluss wird mittels der Funktion festgelegt pb(), welche die nächsten Entscheidungen sind.*

$$assign(a_\perp, MAX) \tag{3.46}$$
$$pb() \tag{3.47}$$

*Die Komponente $e_\perp$ ist immer aktiv, da es sich um den Wurzelknoten im Abhängigkeitsgraph handelt. Sie dient damit als definierter Einstiegspunkt für die Suche, ähnlich wie der Startpunkt bei einer Navigation.*

***Definition 32*** *Gegeben sei, im zweiten Schritt, die Hilfsfunktion pb:*

$$pb() = \begin{cases} mb(), & if\ fb(e_\perp) \\ pb(), & otherwise \end{cases} \tag{3.48}$$

*wobei die verschachtelte Funktion $fb$ weitere Verzweigungsbedingungen festlegt und bei jeder Iteration von $pb()$ bestimmt wird.*

Die Funktion $pb()$ evaluiert entweder alle fehlenden Verzweigungsbedingungen mittels der Hilfsfunktion $mb()$ oder baut iterativ den Abhängigkeitsbaum mittels der Hilfsfunktion $fb()$ auf. Erst wenn $fb()$ keine Bedingungen ($fb()$ evaluiert zu 1) bestimmt hat, werden die fehlenden Verzweigungsbedingungen mittels $mb()$ bestimmt.

***Definition 33*** *Gegeben sei die rekursive Funktion $fb(c \in C)$, die den ersten nicht evaluierten Knoten im Abhängigkeitsbaum sucht, um weitere Verzweigungsbedingungen festzulegen:*

$$\begin{aligned}
fb(c \in C) :=& \quad && (3.49)\\
finished =& \quad 1 && (3.50)\\
if \neq assigned(f_c) :& \quad assign(f_c, MIN) \wedge finished = 0 && (3.51)\\
if \neq assigned(h_c) :& \quad assign(h_c, MIN) \wedge finished = 0 && (3.52)\\
\forall k \in \phi(c) :& \quad \begin{cases} assign(k, MIN) \wedge & \\ assign(a_k, MIN) \wedge & if\ \neq assigned(k) \\ finished = 0, & \\ fb(k) = 0 \Rightarrow finished = 0, & otherwise \end{cases} && (3.53)\\
return\ finished& && (3.54)
\end{aligned}$$

Die Definition 33 definiert die Verzweigungsbedingungen für den nächsten nicht bestimmten Kindknoten von $c \in C$. Formel 3.51 und 3.52 definieren, dass die rekursiven Abhängigkeiten dieser Komponente, wenn sie noch nicht definiert wurden, als nächstes definiert werden müssen. Damit werden die Elternknoten dieser Komponente zugeordnet, um sie im Baum korrekt einzuhängen. Die Formel 3.53 definiert, dass jedes bisher nicht zugewiesene Kind sowie dessen Abhängigkeit festgelegt werden muss, sofern dies bisher nicht geschehen ist. In diesem Schritt wird somit das nächste Kind eines bestimmten Knotens $c$ gesucht. Falls jedoch die Kinder eines Knotens sowie dessen Aktivitätszustand zugeordnet waren, wurden bisher keine weiteren Verzweigungsbedingungen definiert. Für diesen Fall werden die Kinder der aktuellen Komposition weiter untersucht. Dies geschieht mittels eines rekursiven Aufrufs der Funktion $fb()$ selbst.

***Definition 34*** *Gegeben sei die Funktion $fb(e \notin C)$. Diese Funktion definiert alle nötigen Bedingungen, sofern es sich bei der übergebenen Entität $e$ um keine Komposition $\notin C$ handelt:*

$$fb(e \notin C) := \tag{3.55}$$
$$finished = \quad 1 \tag{3.56}$$
$$if 1 \neq assigned(a_e): \quad assign(a_e, MIN) \wedge finished = 0 \tag{3.57}$$
$$return\ finished \tag{3.58}$$

Analog zur Definition 33 wird mittels 34 festgelegt, dass bisher nicht aufgelöste Bedingungen der Kindzuordnung einer Komposition $c$ als Nächstes behandelt werden müssen. Die Sonderbehandlung für Kinder, die keine Kompositionen sind (in der Regel Tasks), ist nötig, da diese Entitäten keine weiteren Kinder besitzen können. Daher beschränkten sich die weiteren Verzweigungsbedingungen dieser Kinder auf die Selektion, ob dieses Kind aktiv sein sollte. Ist auch diese Selektion bereits durch den CSP-Löser festgelegt, sind keine weiteren Assoziationen in diesem Teilsuchbaum nötig. Diese Information wird zurückgegeben, bis alle nötigen Zuweisungen, die durch $fb()$ abgearbeitet werden müssen, getroffen wurden.

***Definition 35*** *Gegeben sei die Definition der Funktion $mb()$, die für alle bisher nicht im Abhängigkeitsbaum befindlichen Entitäten festlegt, dass:*

$$mb() := \tag{3.59}$$
$$for all e \in E \wedge \neq assigned(a_e): \quad a_e = 0 \wedge f_e = \{\emptyset\} \wedge h_e = \{\emptyset\ \} \tag{3.60}$$
$$assign(F, MIN, MIN) \tag{3.61}$$
$$assign(H, MIN, MIN) \tag{3.62}$$

*diese Komponenten nicht aktiv sein können und sie keine umgekehrten Abhängigkeiten besitzen dürfen.*

In der Definition 35 werden alle nicht benötigten Entitäten aufgelöst und als inaktiv deklariert. Jeder bisher nicht festgelegte Task kann somit nur inaktiv sein, auch kann er keine Abhängigkeiten besitzen. Diese Optimierung deaktiviert alle verbliebenen Knoten, anstatt eine teure Suche anhand der bisherigen Constraints ausführen. Das Problem während des CSP-Lösens wäre ansonsten, dass ohne diese Einschränkung Teilbäume gebildet werden könnten, die unabhängig vom Wurzelknoten wären. Diese würden erst dann erkannt werden können, wenn die unabhängigen Teilbäume vollständig gebildet wurden. Der Grund dafür ist, dass das Ermitteln der fehlenden Kante zum Wurzelgraphen erst nach vollständiger Bestimmung aller Kanten möglich ist (Vergleich 3.24). Somit müssten vorher viele teure Möglichkeiten evaluiert werden, wodurch die Suchkomplexität enorm stiege.

**Generierung des Abhängigkeitsgraphen** Wie es zur Einleitung dieses Teils der Arbeit beschrieben wurde, ist es das Ziel, den Abhängigkeitsgraph auf Klassenebene zu bilden. Die zuvor gestellten Constraints und Verzweigungsbedingungen, die die Heuristik der Suche bilden, generieren die nötige Lösung. Es ist jedoch noch erforderlich, aus dieser Lösung einen Graph zu generieren. Die bisher genutzten Datenstrukturen sind lediglich (Hilfs-)Mengen, die zum Lösen des Problems mittels CSP-Verfahren genutzt wurden. Die Extraktion einer Datenstruktur als Baum reduziert die Speicherkomplexität, da alle nicht mehr benötigten Informationen entfernt werden. Hierzu zählen unter anderem explizite Aktivitätsinformationen sowie alle nicht mehr benötigten Komponenten und die internen Suchzustände des CSP-Solvers.

***Definition 36*** *Gegeben sei die Funktion $t(c)$, die einen gerichteten Graph $G$ erstellt. Die Menge aller Knoten $G_V$ sei dabei die Menge aller Entitäten $G_V = E$, und die Menge aller Kanten sei definiert als Teilmenge zwischen zwei Entitäten $G_E = \{(e_{parent}, e_{child})\}$. Dabei beginnt die Baumgenerierung mit der Wurzel des Aktivitätsgraphs $t(e_\perp)$.*

$$t(c) := \tag{3.63}$$

$$\forall j \in \phi(c) : \tag{3.64}$$

$$(c, j) \in G_E \wedge \tag{3.65}$$

$$j \in C \Rightarrow t(j) \tag{3.66}$$

Die Definition 36 bildet den Abhängigkeitsgraph auf Klassenbasis. Die Formel 3.65 beschreibt, dass alle Kinder einer Komposition $j \in \phi(c)$ als Kante zwischen der Komposition

und dem Kind im Graph vorkommen müssen. Die Formel 3.66 definiert, dass der Graph für die Kinder eines Kinds $j$, sofern dieses selbst eine Komposition $j \in C$ ist, rekursiv gebildet werden muss. Dies geschieht über den rekursiven Aufruf der Funktion $t(c)$ selbst. Das Ergebnis ist der sogenannte Abhängigkeitsgraph auf Klassenebene. Dieser wird im nächsten Absatz dieser Arbeit weiter genutzt, um die Instanzlösung eines Netzwerkes zu bilden. Mit diesem Schritt ist die Bestimmung der Klassenlösung abgeschlossen.

#### 3.1.4.4 Constraint schritt zwei - Die Instanzlösung

Im vorherigen Abschnitt zur Klassenlösung (Seite 98) wurde beschrieben, wie der Abhängigkeitsgraph auf Klassenebene gebildet wird. Dazu wurden alles Constraints und Heuristiken erstellt und ausführlich erörtert. Das Endergebnis dieses vorangegangenen Abschnitts war der Abhängigkeitsgraph $G$ auf Klassenebene.

Dieser Teil der Arbeit widmet sich der Bestimmung und der Konfigurationen der Instanzen, die zu einem bestimmten Zeitpunkt aktiv sein müssen. Die Klassenlösung unterscheidet nicht zwischen Instanz und Klasse eines Algorithmus. Auch hat ein Algorithmus bisher keine Konfiguration. Beides sind jedoch wesentliche Elemente, die beachtet werden müssen, um eine anwendbare Lösung für Robotersysteme zu ermitteln. Es wird in diesem Teil der Arbeit bestimmt, ob eine mehrfach verwendete Klasse aus der vorherigen Lösung mehrfach instanziiert werden kann oder sogar muss. Dazu wird ihre Benutzung innerhalb des Netzwerkes untersucht, aber auch welche Anforderungen an Konfigurationen existierten. Es ist möglich, dass die Verwendungen einer Komponente in einem Teilbaum anderen Anforderungen unterliegt als die gleiche Klasse in einem anderen Teilbaum.

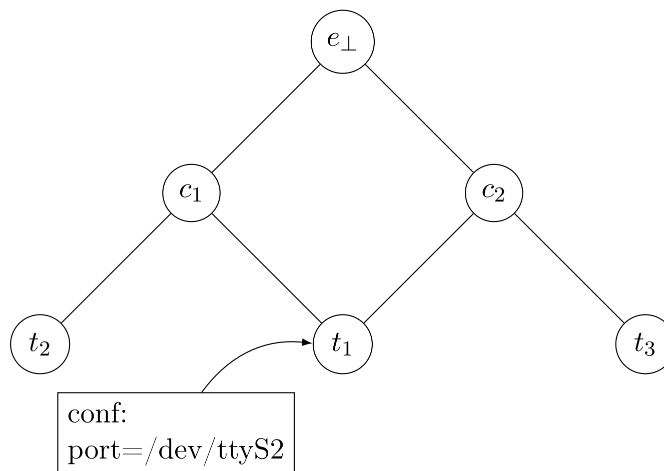



Figure 3.17: Beispiel einer Instanzlösung aus einer Klassenlösung.
Der Graph beschreibt dabei den Abhängigkeitsgraphen inkl. einer konfiguration *conf* der komponente *T1*

Ein Beispiel hierfür ist in Bild 3.17 bzw. in dem korrespondierenden Bild 3.18 zu sehen. Der Graph aus der Klassenlösung 3.17 zeigt, dass der Knoten $t_1$ von zwei Elternknoten verwendet wird. Sowohl $c_1$ als auch $c_2$ benötigen diesen Knoten als Kind. Für die Klassenlösung ist nicht ersichtlich, ob dies bedeutet, dass der Task $t_1$ einmal oder

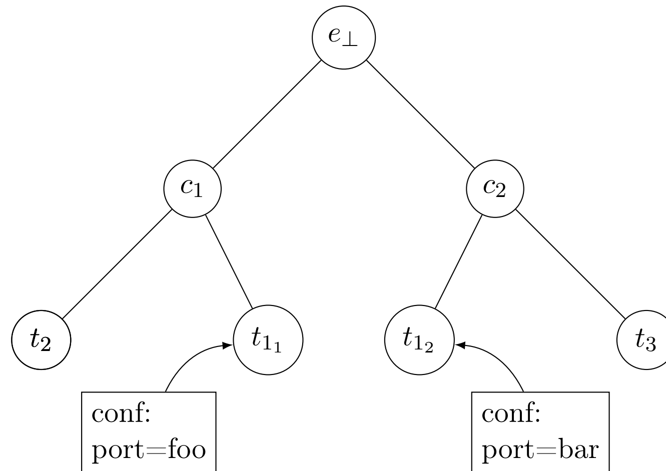



Figure 3.18: Beispiel einer anderen möglichen Instanzlösung für die gleiche Klassenlösung aus 3.17

zweimal instanziiert werden kann oder sogar muss. Eine mögliche Lösung wäre, den Task zweimal zu instanziieren, wie in Bild 3.18 gezeigt. Eine andere Option wäre, den Task nur einmal zu instanziieren und als geteilte Ressource von $c_1, c_2$ zu verwenden. Dies ist jedoch nicht in allen Fällen möglich. Sobald sich sie die Konfigurationsanforderungen von $c_1$ und $c_2$ unterscheiden und dies zu einem Konflikt führen würde, muss eine Trennung stattfinden und die geteilte Ressource zweimal instanziiert werden. Andererseits gibt es Fälle, in denen dieser Task beispielsweise aus mangelnden Ressourcen nicht mehrfach gestartet werden darf. Unter diesen Umständen wäre die Klassenlösung nicht als Instanzlösung lösbar. Dieses Szenario kann beispielsweise eintreten, falls es sich bei $t_1$ um ein Gerät handeln würde, das nur exklusiv genutzt werden darf. Bestehen zwei unterschiedliche Anforderungen an dieses eine Gerät, ist dieses auf der Ebene der Instanzbestimmung nicht lösbar. An dieser Stelle sei darauf hingewiesen, dass ein Gerät von $c_1, c_2$ gleichzeitig genutzt werden kann, wenn die gleichen Anforderungen an das Gerät existieren.

Um die Klassenlösung formal zu definieren, soll das Problem um folgende Fragestellungen erweitert werden:

- Soll eine Klasse mehrfach als die gleiche Instanz verwendet werden?
- Welche Konfiguration soll eine spezifische Instanz besitzen?

Beide Fragen sind miteinander verbunden. Um auch hier eine möglichst flexible Lösung bereitzustellen, werden manche Konzepte, denen auch Syskit folgt, übernommen. Zu diesen zählen:

- Elternknoten (Komposition) können Konfigurationswünsche an Kinder propagieren bzw. determinieren.

- Wenn ein Kind mehrfach verwendet wird, müssen die unterliegenden Teilbäume ab dem Kind beider Eltern identisch sein.
  Dieses Kriterium ist relevant, wenn eine Komposition von zwei Eltern zeitgleich verwendet wird. Es ist von hoher Relevanz, dass die Kinder der gemeinsam genutzten Komposition bis zum Erreichen eines Blattes identisch sind. Ein Beispiel, in dem dieses nicht möglich wäre, ist in Bild 3.19 zu sehen.

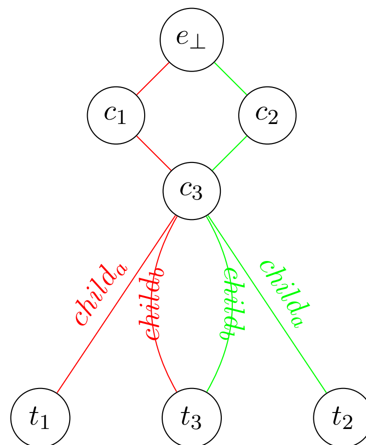



Figure 3.19: Widersprüchliche Verwendung von $c_3$ von Seiten der Kompositionen $c_1, c_2$. Die Kinder der Komposition $c_3$ sind unterschiedlich je nach Instanziierung von $c_1, c_2$

Das Beispiel aus Abbildung 3.19 ist deswegen ungültig, weil es erfordern würde, dass der funktionale Verbund $c_3$ unterschiedliche Kinder je nach Elternteil benutzte. Unter der Annahme, $c_3$ sei eine Sensorfusion, $t_1, t_2$ IMUs an unterschiedlichen Orten im System sowie $t_3$ der eigentliche Fusionstask, ergibt sich, dass $t_3$ sowohl mit $t_1$ als auch mit $t_2$ verbunden werden müsste, was ungültige Daten produzieren würde. In diesem Fall müsste $c_3$ entweder zweimal instanziiert werden oder es müsste $t_1 = t_2$ gelten.

Um die Probleme, die bei der Verwendung von Syskit/Roby auftraten, konzeptionell zu lösen, wird im weiteren Verlauf von den Syskit-Konzepten abgewichen. Im Gegensatz zu Syskit, das das Zusammenfassen von gleichen Graphen nur dann erlaubt, wenn sie identisch sind, wird in diesem neuen Konzept das Zusammenfassen von Teilbäumen auch dann gestattet, sofern sie nicht widersprüchliche Anforderungen an ihre Kindelemente stellen. Ein Beispiel für nicht widersprüchliche Konfigurationen ist das Sonarkonfigurationsbeispiel,

das zu Beginn dieses Kapitels unter 3.1.2 eingeführt wurde. In Syskit ist lediglich eine Modellierung möglich, wie sie im Graph 3.20 zu sehen ist.

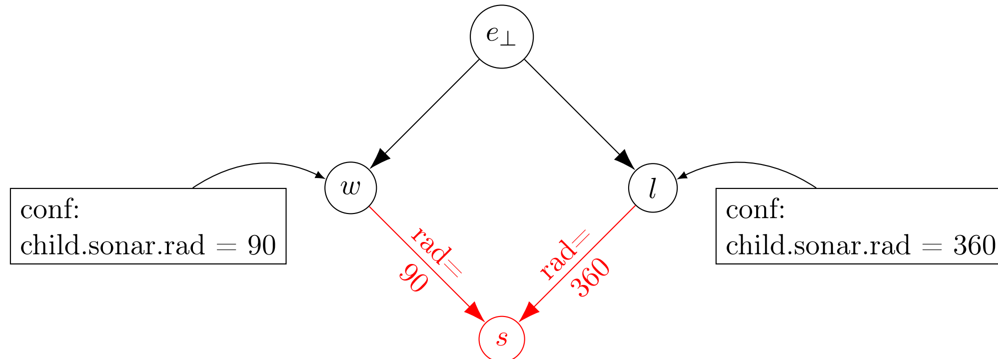



Figure 3.20: Widersprüchliche Verwendung von $c_3$ von $c_1, c_2$. Dabei bedeuten $w$: wall, $l$: localization und $s$: sonar.

Hierbei wird deutlich, dass verschiedene Anforderungen aus verschiedenen Teilbäumen an das Sonar bestehen. Das Erstellen von Modellen, die in der Lage Sind, diese Art von Problemen konzeptuell valide zu behandeln, ist eine der Kernerweiterung. Dazu wird das Syskit-Modell um die Annahme erweitert, dass verschiedene Anforderungen erlaubt sind und dass sich die dadurch entstehenden Teilbäume beeinflussen dürfen. Die Frage ist, wie diese Modelle aussehen könnten, die eine valide und praktikable Syntax bereitstellen. In dem Beispiel 3.21 ist hierfür die konzeptuelle Grundidee anhand des genannten Beispiels verdeutlicht. Anstatt feste Werte der Sonarkonfiguration zu definieren, wird die Modellbeschreibung dahingehend erweitert, dass es möglich ist, Constraints für Konfigurationsattribute festzulegen. Der Netzwerkdesigner bzw. die Person, die die Modelle für (Teil-)Systeme erstellt, erhält somit die Möglichkeit, flexibler zu definieren, in welchem Rahmen sein Algorithmus arbeiten kann.

Im Beispiel der Sonarkonfiguration ist es für die unterliegenden Algorithmen möglich, eine flexiblere Konfiguration zu definieren, mit der sie dennoch in der Lage sind, ihre Operation auszuführen. Die Lokalisierung kann auch mit einer eingeschränkteren Wahrnehmung (wenngleich mit schlechterer Qualität) operieren. Die Wandverfolgung kann ebenso mit einem flexibleren Wertebereich des Sonars die Wand noch zuverlässig detektieren. Daher wäre es möglich, die Konfiguration für das Sonar zu wählen, die für beide (Teil)-Anforderungen akzeptabel ist.

Eine andere Erweiterung ist die explizite Modellierung der Propagation von Konfigurationsattributen. Syskit erlaubt zwar die Modellierung von Attributen, diese werden jedoch nur imperativ zur Ausführung propagiert. Im Gegensatz dazu schlägt diese Arbeit eine

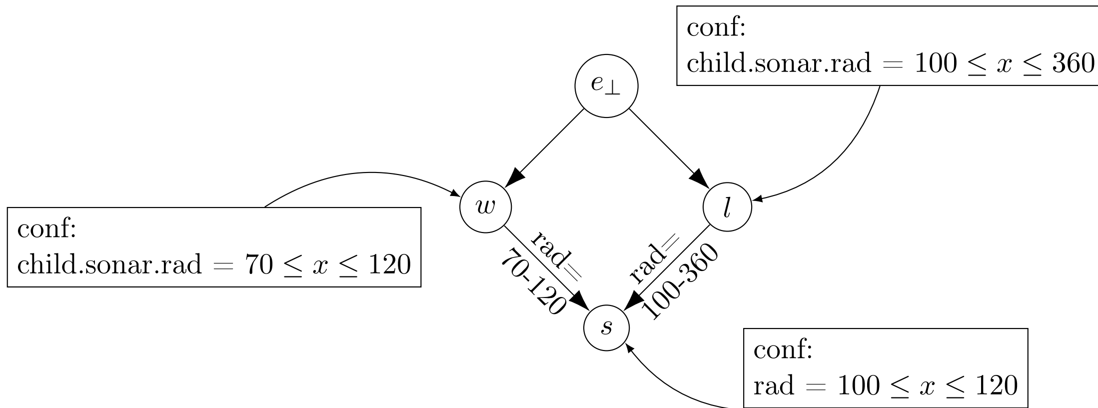



Figure 3.21: Erweiterung der Modellierung um Konfiugrations-Constraints zur Lösung des Problems aus 3.20

formale Betrachtung dieser Konfigurationseigenschaften vor. Die Lösung ist eine Propagation von Konfigurationen mittels der Kompositionen auf Modellbasis. Eine Komposition selbst kann eine Konfiguration besitzen, die (bzw. auch deren Teile) sie an ihre Kinder propagieren kann. Die Kinder müssen, wenn nötig, diese Konfiguration annehmen und selbst an ihre Kinder weiter propagieren, bis eine Eigenschaft ein Blatt im Abhängigkeitsgraph, also ein Task, erreicht. Die Propagation von Attributen über mehrere Kompositionen ist in Abbildung 3.22 dargestellt.

**Erweiterung des Formalismus** Um die zuvor definierten Eigenschaften der Modelle mittels des Constraint- Solvers lösen zu können, muss das neue Modell zunächst formalisiert werden. Im Folgenden wird somit der neue Formalismus aufgestellt und erläutert.

***Definition 37*** *Jede Komponente des Modellwissens $e \in E$ besitzt weitere Merkmale:*

$$\Pi = (\alpha, \beta) \tag{3.67}$$

$$\Psi(e \in C \wedge T) = \{\Pi\} \tag{3.68}$$

$$\Gamma(c \in C) = \{(\Pi, e \in E(e \in \Phi(c)), \Pi)\} \tag{3.69}$$

$$\Xi(t \in T) = n \in \mathbb{N} \tag{3.70}$$

*wobei $\Pi$ eine Konfiguration ist, $\alpha$ der Name einer Konfiguration, $\beta$ der belegte Wert oder ein Constraint der Konfiguration mit Namen $\alpha$, $\Psi(e)$ die Menge aller Konfigurationen*

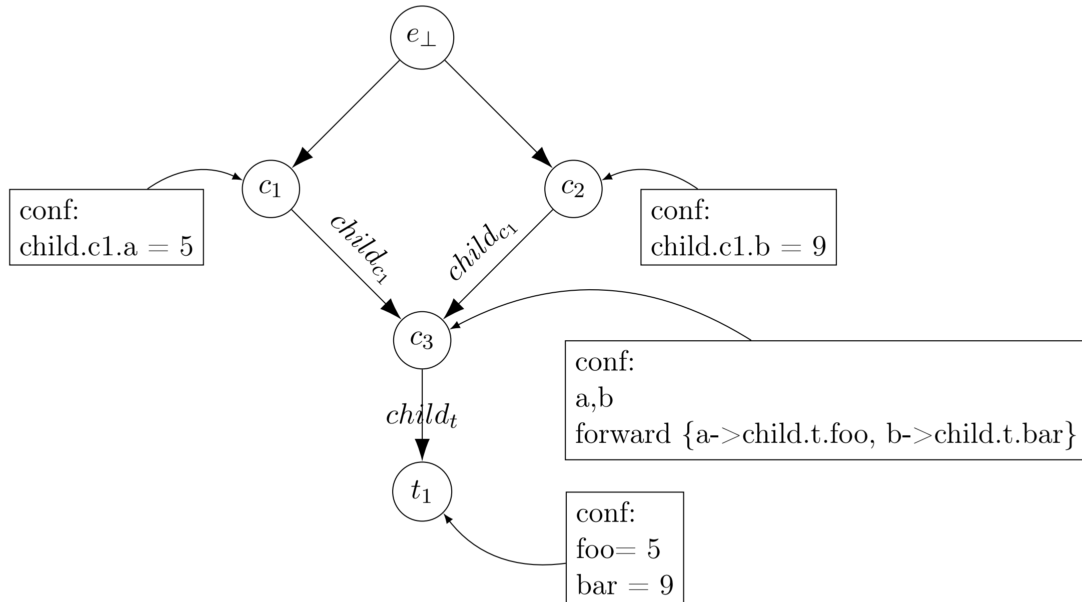



Figure 3.22: Beispiel für das modellbasierte Propagieren von Konfigurationen

*für einen Task oder eine Komposition,* $\Gamma(e)$ *die Menge aller Argumentweiterleitungen zur Propagation inkl. der Abbildung und* $\Xi$ *die Anzahl, wie häufig eine Komponente innerhalb des Komponentennetzwerkes genutzt wurde.*

Die erweiterten Attribute des Modells erklären sich wie folgt. Eine Komposition hat als weitere Attribute: eine Menge an Konfigurationen $\Psi$ sowie die einzelne Konfiguration $\Pi$, die aus einem Namen $\alpha$ und dem Constraint (bzw. wert) $\beta$ besteht. $\Gamma(c)$ liefert die Menge an Verschränkungen zwischen der Konfiguration einer Komposition und ihren Kindern (wie im Bild 3.22 dargestellt). Jede Konfiguration der Komposition kann mit einer anderen Konfiguration eines Kindes verschränkt sein. Sofern sie verschränkt ist, muss sie die gleiche Wertigkeit besitzen. Diese kann ein Wertebereich oder ein fest definierter Wert sein.

Die Funktion $\Xi$ bildet einen Task auf die maximal vorhandene Instanzanzahl ab. Dies ist nötig, wenn beispielsweise ein Task $t$ ein Gerätetreiber ist. In diesem Fall ist es algorithmisch wenig sinnvoll, diesen Treiber mehrfach zu instanziieren, da das unterliegende Gerät nur einmal benutzt werden kann. Es wäre in diesem Beispiel sinnvoll, $Xi(Imu) = 1$ zu definieren.

***Definition 38*** *Gegeben sei die Verschränkungsmatrix* $\Theta$*, die angibt, ob ein Element mit*

*einem anderen verschränkt ist.*

$$\Theta = [[0,1]^{|G_V|}]^{|G_V|} \tag{3.71}$$

Wenn für einen Knoten im Graph eine Verschränkung existiert, so gilt $\Theta_{3,2} = \Theta_{2,3} = 1$ und $\Psi(e_2) = \Psi(e_3)$, wobei die Größe des Vektors $|G_V|^{|G_V|}$ entspricht. Dies ergibt sich daraus, dass es bis zu $|G_V|$ Komponenten gibt, die mit bis zu $|G_V|$ anderen Komponenten verschränkt sein können. Ein Beispiel für diese Verschränkung wurde bereits zu Beginn der Instanzlösung durch die Abbildungen 3.17 und 3.18 erläutert.

Die Identifizierung der Belegung von $\Theta$ ist, im Gegensatz zu $\Pi, \Psi, \Gamma, \Xi$, kein Modellwissen, das durch den Anwender mit den Modellen definiert wird. $\Theta$ ist eine Hilfsvariable, die mittels des CSP-Solvers während des CSP-Lösungsprozesses bestimmt wird. Der CSP-Löser muss die Matrix $\Theta$ dahingehend belegen, dass eindeutig alle Verschränkungen identifiziert werden.

**Constraints der Instanzlösung** Im folgenden Absatz werden, genauso wie zuvor für die Klassenlösung, die Constraints aufgestellt, die der CSP-Solver beachten muss, um eine gültige Instanzlösung bestimmen zu können. Diese Constraints müssen, analog zu den Constraints der Klassenlösung (Seite 98), abgearbeitet werden.

***Definition 39*** *Gegeben sei, dass für jede Kante* $(e_1, e_2)$ *bzw. äquivalent alle Konfigurationsweiterleitungen, die* $e_1$ *definiert,* $\Gamma(e_1)$ *gilt:*

$$\forall(e_1, e_2) \in G_E, \forall(a, b, c) \in \Gamma(e_1): \quad a \Leftrightarrow c \tag{3.72}$$

*dass der Konfigurationswert der Konfiguration a dem Konfigurationswert c vom Kind b entspricht.*

***Definition 40*** *Es gilt, dass alle Konfigurationswerte, die im Modell gegeben sind, eine Abbildung auf die Modell-Entitäten besitzen müssen:*

$$\forall e \in G_V, \forall(\alpha, \beta) \in \Psi(e): \quad \alpha = \beta \tag{3.73}$$

Die Formel 3.72 realisiert somit das Propagieren von Konfigurationsattributen zwischen Kompositionen und ihren Kindern. Die Formel 3.73 hingegen definiert, dass Konfigurationswerte aus dem Modell innerhalb der CSP-Lösung berücksichtigt werden. Diese beiden ersten Definitionen widmen sich hauptsächlich der Behandlung von Konfigurationen.

Die Constraints, die im Folgenden aufgestellt werden, beschreiben das Verhalten der Verschränkung und den Einfluss von Verschränkungen auf verschiedene Teile wie Kindselektion oder die Behandlung von Attributen.

***Definition 41*** *Für Verschränkungen gilt, dass sie symmetrisch sind:*

$$\forall e_1 \in G_V, \forall e_2 \in G_V : \quad \Theta_{e_1,e_2} = \Theta_{e_2,e_1} \tag{3.74}$$

*Ist ein Element $e_x$ mit $e_y$ verschränkt, so gilt, dass auch $e_y$ mit $e_x$ verschränkt sein muss.*

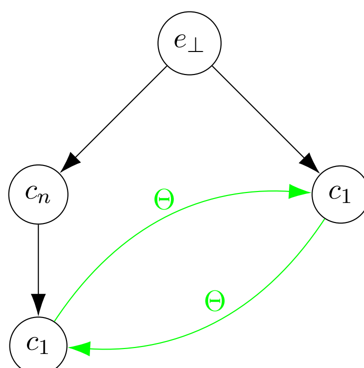



Figure 3.23: Beispiel für eine Verschränkung zwischen $c_1, c_1$ aus verschiedenen Teilbäumen, basierend auf Definition 41. Verschränkungen müssen immer symmetrisch sein.

***Definition 42*** *Gegeben sei die Abbildung equals:*

$$equals(e_1, e_2) \in 0, 1 \tag{3.75}$$

*wobei $e_1, e_2 \in G_V$. Die Abbildung liefert* 1*, falls der unterliegende Algorithmus zwischen zwei Knoten im Graph identisch ist. Dies ist der Fall, wenn durch die Klassenlösung zwei Knoten im Graph eingehangen wurden, die den gleichen Algorithmus repräsentieren. Dies kann durch eine doppelte Repräsentation des gleichen Knotens im Graph der Fall sein oder durch Spezialisierungen.*

***Definition 43*** *Eine Kante kann immer nur mit dem gleichen Typ verschränkt sein. Ist dies nicht gegeben, darf keine Verschränkung vorgenommen werden:*

$$\forall e_1 \in G_V, \forall e_2 \in G_V \overline{equals(e_1, e_2)} : \Theta_{e_1,e_2} \neq 1 \tag{3.76}$$

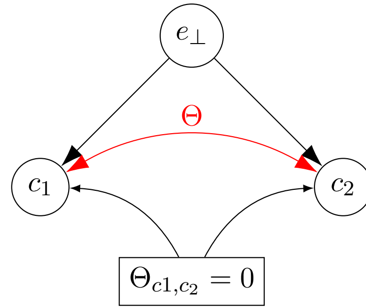



Figure 3.24: Beispiel für unerlaubte Verschränkung zwischen $c_1, c_2$ basierend auf Definition 43

***Definition 44*** *Es ist nicht möglich, dass eine Komponente mit sich selbst verschränkt ist:*

$$\forall e_1 \in G_V, \forall e_2 \in G_V, e_1 = e_2 : \Theta_{e_1,e_2} \neq 1 \tag{3.77}$$

Das Constraint 44 dient als Definition, sie hat keinen weiteren Einfluss auf die Lösung.

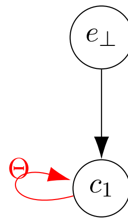



Figure 3.25: Laut Definition 44 ist eine Komponente niemals mit sich selbst verschränkt.

***Definition 45*** *Es gilt, dass die Konfiguration von Komponenten, wenn diese zusammengefasst werden, identisch sein muss:*

$$\begin{aligned} \forall e_1 \in G_V, \forall e_2 \in G_V \; equals(e_1, e_2) \wedge e_1 \neq e_2 : \\ \Theta_{e_1,e_2} = 1 \Rightarrow \Psi(e_1) = \Psi(e_2) \end{aligned} \tag{3.78}$$

Dies ist nötig, da bei einer Verschränkung nur eine Instanz des Algorithmus existiert. Die Konfiguration dieser einen Instanz muss konsistent sein. Weiterhin wird dadurch gewährleistet, dass die Konfiguration des jeweils anderen Teilbaums auch im aktuellen Teilbaum gültig ist.

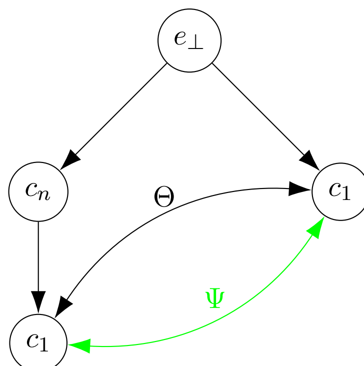



Figure 3.26: Beispiel für eine Verschränkung zwischen $c_1, c_1$. Wenn sie verschränkt sind, müssen ihre Konfigurationen $\Psi$ identisch sein (siehe Definition 45)
.

Wie in Definition 45 beschrieben, ist es nötig, dass die Konfigurationen identisch sein müssen, sofern eine Verschränkung vorliegt. Wie eingangs in diesem Kapitel beschrieben, ist es möglich, dass die verschränkten Knoten unterschiedliche Konfigurationsanforderungen durch ihre Elternknoten besitzen. Jedoch ist es Aufgabe des Constraintlösers, diese Anforderungen in einen konsistenten Zustand zusammenzuführen. Ansonsten dürfte keine Verschränkung vorgenommen werden und der Task müsste zweimal auf dem Zielsystem ausgeführt werden, sofern dies gemäß anderen Randbedingungen möglich ist.

***Definition 46*** *Es muss gelten, dass alle Kinder von zwei Knoten, falls diese verschränkt werden, identisch verschränkt sein müssen:*

$$\begin{aligned} \forall e_1 \in G_V, \forall e_2 \in G_V \ equals(e_1, e_2) \wedge e_1 \neq e_2 : \\ \Theta_{e_1, e_2} \Rightarrow \forall k \in [0, \ldots, |Phi(e_1)|] \ \Theta_{e_{1_k}, e_{2_k}} \end{aligned} \tag{3.79}$$

*wobei für alle Knoten, sofern sie nicht die gleichen sind, sich aber algorithmisch erfüllen, gilt, dass sofern sie verschränkt sind, alle ihre Kinder $e_{1_k} = e_{2_k}$ verschränkt sein müssen $\Theta_{e_{1_k}, e_{2_k}}$.*

Die Definition 46 beschreibt die Ungültigkeit des in Bild 3.19 vorgestellten Falls. Sofern ein Elternknoten verschränkt wird, müssen alle Kinder von ihm ebenfalls verschränkt werden. Diese Anforderung gilt, da sonst ungültige Datenflüsse in der Instanziierung entstehen würden. Die Erklärung hierzu wurde in der Einleitung dieses Kapitels beschrieben und wird durch Definition 46 formalisiert.

Mit dieser Definition 46 sind alle benötigten Constraints für die Instanzlösung aufgestellt. Im Kapitel 3.1.6 werden die Ergebnisse der Instanzlösung anhand verschiedener Modelle

erläutert. Im folgenden Absatz 3.1.4.5 wird eine Erweiterung der Instanzlösung eingeführt, die es neben den bisher statischen Komponentennetzwerken erlaubt, eine Verhaltensmodellierung über mehrere Zustände und somit Komponentennetzwerke vorzunehmen.

#### 3.1.4.5 Erweiterung der Modellierung um Sequenzierung von Verhalten

In dem vorherigen Kapitel wurde die Erstellung eines Komponentennetzwerkes für einen Zeitpunkt formal aufgestellt. Dieser Abschnitt der Arbeit widmet sich hingegen der Erarbeitung einer Modellierung für ein Roboterverhalten, das über mehrere Zeitpunkte und demzufolge über mehrere Komponentennetzwerke entsteht. Wie bereits in der Einleitung zu diesem Kapitel beschrieben, ist die modellbasierte Analyse von Komponentennetzwerken bzw. die Sequenzverifikation von keinem bestehenden Ansatz des Standes der Technik abgedeckt.

Die Arbeit von Joyeux [33] zeigt ein eventbasiertes System, das reaktiv auf Änderungen reagieren kann. Dieses System folgt dabei einem Event/Action-Pattern, dem auch große Kommunikationsframeworks wie NokiaQT folgen. Das Event/Aktion-System von Roby bietet dem Entwickler viele Freiheiten in der Modellierung seiner Verhalten, da eine Aktion letztendlich auf einen durch den Entwickler frei definierbaren Codeblock abgebildet wird. Dieser gewählte Ansatz hat jedoch konzeptionelle Schwächen. Einerseits wirken sich Fehler in der Programmierung, wie bereits in 2.5.4 beschrieben, erst zur Laufzeit aus, andererseits ist eine formale Verifikation hier nicht möglich, da dazu bestehender Code analysiert werden müsste.

Diese Arbeit schlägt einen anderen Weg vor, ein Roboterverhalten formal zu beschreiben. Die Modellierung von Komponentennetzwerken, wie sie in dieser Arbeit vorgestellt wurde, basiert auf reinen Modellen. Daher sollte auch das Roboterverhalten als formales Modell verstanden werden. Dieses Modell kann verifiziert und interpretiert werden. Somit wurde die Trennung zwischen Ausführung und Modellbeschreibung erreicht, die zu Beginn dieser Arbeit angestrebt wurde.

Anstatt dem Programmierer vielfältige Freiheiten zu erlauben, wird eine formale Einschränkung vorgenommen. Diese Limitierungen stoßen vielmals auf Widerstand in der klassischen Programmierung, da die Entwickler (oder Designer) dies als Einschränkung ihrer Fähigkeiten der Beschreibung verstehen. Daher ist es von Bedeutugn, dass durch diese Einschränkungen der Modelldefinitionsfreiheit keine Reduktion des Funktionsumfangs resultiert.

Durch klarere Schnittstellen werden jedoch Strukturen gefestigt. Es existieren weniger unterschiedliche Wege, ein Ziel zu erreichen. Dieses erleichtert langfristig das Verständnis und reduziert die Einarbeitungszeit neuer Entwickler. Es hat sich in der Praxis gezeigt,

dass die höhere Flexibilität in Systemen wie Syskit/Roby nicht wie ursprünglich vermutet die Akzeptanz in der Community erhöht, sondern nur zu schwerer verständlichen Lösungen führt, da Aufgaben unterschiedlich gelöst werden können und die Entwickler somit mit einer zu hohen Komplexität belastet werden. Diese unterschiedlichen Wege vergrößern zudem die Komplexität des Plan-Managers selbst, was wieder zu einer höheren Fehleranfälligkeit führt.

**Anforderungen an die zeitliche Sequenzierung von Aktionen** Die zeitlichen Aktionen, die das Systemverhalten determinieren und die somit der Plan-Manager bestimmen muss, lassen sich wie folgt zusammenfassen:

1. Änderung des Netzwerkes aufgrund einer Zustandsänderung innerhalb des Netzwerkes.
2. Reagieren auf Fehler im Netzwerk und Einleiten von alternativem Verhalten.
3. Selektion von alternativem Verhalten, falls Mehrdeutigkeiten in der Instanz/Klassenlösung bestehen.

Viele Arbeiten, wie [36], [51], [1], beschäftigen sich explizit mit Fehlern innerhalb von Netzwerken und ihrer Behandlung. Es wird oftmals zwischen dem eigentlich gewünschten Systemverhalten und Fehlerzuständen unterschieden. Diese konzeptionelle Trennung ist jedoch für das hier vorgestellte System irrelevant. Ob ein Folgenetzwerk aufgrund eines Fehlers oder einer erwarteten Änderung aufgebaut wird, spielt für die Berechnung des Folgenetzwerkes keine Rolle. Der einzige Unterschied bei Fehlerfällen ist, dass gegebenenfalls einzelne Komponenten nicht mehr existieren oder funktionsunfähig sind.

Um den Zustand einer Komponente zu überwachen, definiert Orocos/RTT Komponenten als Zustandsautomaten (siehe Abbildung 3.27). Die Arbeiten von [34] erweitern die Komponenten um eine Rückmeldung weitere beliebige Zustände (siehe Abbildung 3.28). Eine grundlegende Annahme für die modellgetriebene Modellierung von (Komponenten-) Netzwerken ist, dass die Komponenten selbst keine Entscheidungen über das Systemverhalten (bzw. das Komponentennetzwerk) treffen. Somit besteht eine klare funktionale Trennung zwischen allen Komponenten und dem Plan-Manager. Komponenten melden ihre Zustände an den Plan-Manager, der auf Basis seiner internen Zustände und des Zustands aller Komponenten entscheidet, ob eine Änderung im Komponentennetzwerk selbst (bzw. dadurch im Verhalten des Systems) nötig ist.

Diese Annahme wird jedoch von aktuellen Lösungen immer wieder durchbrochen, da es möglich ist, in der Modellierung von Syskit/Roby nicht nur die Zustände, sondern auch Datenflüsse zwischen Komponenten zu überwachen. Dadurch steigen die Komplexität und

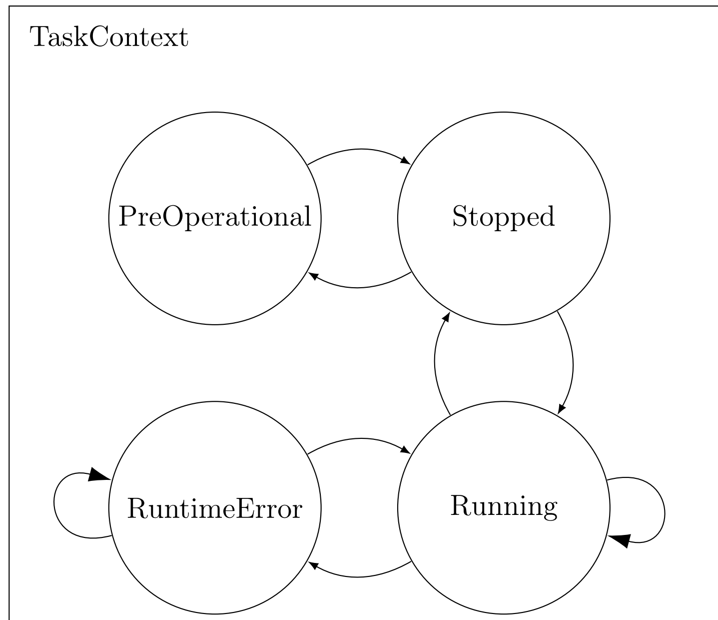



Figure 3.27: Zustandsautomat einer RTT Komponente

der Aufgabenumfang in einer unnötigen Art und Weise. Syskit/Roby ermöglicht es hierdurch, dass Kompositionen eine Verbindung mit den Komponenten selbst eingehen können. Kompositionen sind im Roby/Syskit-Zusammenhang daher nicht rein virtuelle Knoten, die zur Modellierung des Systems dienen. Sie sind aktiv mit Komponentennetzwerk verwoben, was eine Verletzung der klaren hierarchischen Trennung darstellt. Im Abbildung 3.29 ist ein Beispiel aufgezeigt, das verdeutlicht, dass jede in Syskit durchgeführte Operation umformbar in ein klares Modell ist, das die formale Trennung zwischen beiden Ebenen einhält.

Die bisherige Definition eines Komponentennetzwerkes instanziiert ein Komponentennetzwerk. Es wurde bisher nicht erläutert, wann dieses Komponentennetzwerk seine Aufgabe erfüllt hat oder ob es eine Aufgabe ist, die unbegrenzt aktiv sein soll. In dieser Arbeit wurde daher wissentlich auf die Terminologie Aktion oder Fähigkeit verzichtet. Aus der Sicht eines Komponentennetzwerkmanagers ist es formal irrelevant, ob eine Anforderung zu einem Zeitpunkt oder für eine gewisse Dauer besteht, somit also kontinuierlich unbegrenzt aktiv ist oder zu einem gegebenen Zeitpunkt terminiert. Dieses Modell lässt sich direkt auf die Orocos/RTT-Modellierung übertragen. Für einen Task ist es irrelevant, ob er infinit aktiv ist oder irgendwann seinen Zustand wechselt. Das gleiche Konzept lässt sich auch auf die deliberative Verhaltensmodellierung, die im Folgenden vorgestellt wird, übertragen.

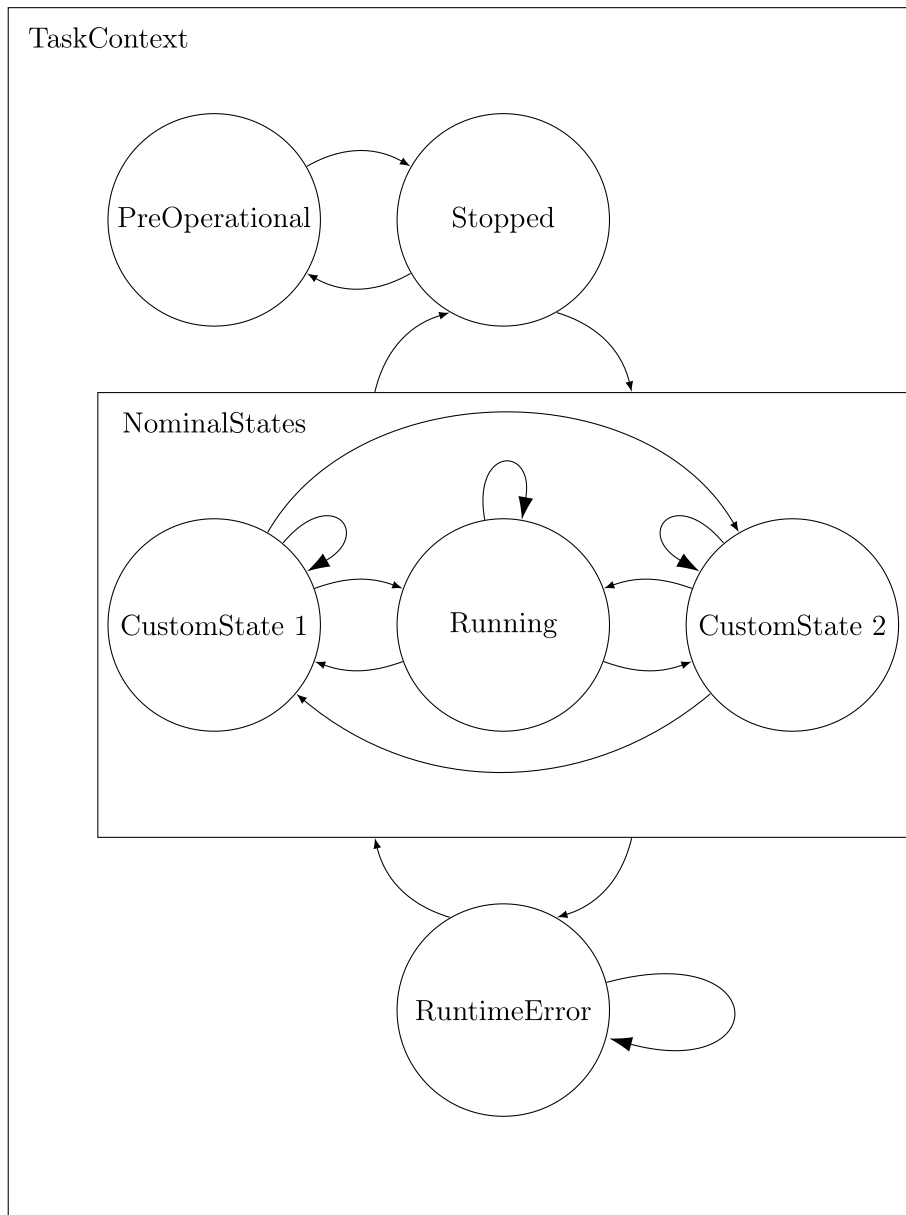



Figure 3.28: Erweiterter Zustandsautomat einer RTT Komponente

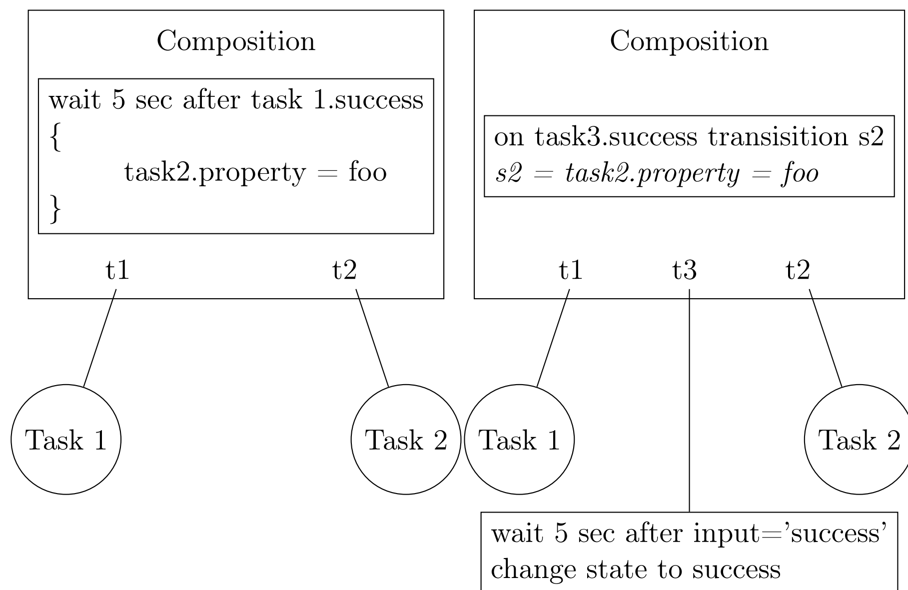



Figure 3.29: Umformung von Kompositionscode

### StateMachines

Wie im vorherigen Absatz 3.1.4.5 beschrieben, werden Komponenten von Orocos/RTT, beziehungsweise deren Erweiterungen, als endliche Zustandsautomaten (FSM) interpretiert. Um dieses Konzept auf die Modellierungsebene zu übertragen, werden im Folgenden erweiterte Kompositionen, die sogenannten *Kompositions-StateMachines* (KSMs), eingeführt. Diese Kopositions-StateMachine (KSM) verhalten sich ähnlich wie Kompositionen und können zunächst äquivalent zu ihnen aufgefasst werden. Die KSMs bieten jedoch weitere Besonderheiten. Sie können einen oder mehrere Zustände haben, wobei nur ein Zustand zu einem Zeitpunkt aktiv sein darf. Der Zustand definiert zugleich, welche Kind(er) die *Kompositions-StateMachine* hat. Die Erweiterung umfasst, dass es Transitionen innerhalb einer KSM gibt, die in Abhängigkeit ihrer Kind-Zustände ausgeführt werden. Diese Änderungen im Zustand bewirken eine Änderung des Modells der KSM. Konkret können sich sich die Anforderungen an Kinder verändern. Das Komposition-Modell erhält somit eine dynamische Komponente und erfordert eine Netzwerk-Transition bei einer Änderung des Zustandes einer KSM.

Eine *Kompositions-StateMachine* hat dabei:

- $\Sigma$ Events, auf die sie hört
- $S$ mögliche Zustände
- Einen Startzustand $s_0 \in S$
- $\delta$ als Transitionsfunktion, wobei $\delta : S \times \Sigma \Rightarrow S$
- $F \subset S$ als Zielzustände.

***Definition 47*** *Eine Entität e kann beliebige Zustände einnehmen.*

$$S(e) = s_0, \ldots, s_n \tag{3.80}$$

***Definition 48*** *Gegeben sei eine KSM c, ihre möglichen Events $\Sigma$, auf die sie hören kann. Diese Events entsprechen zugleich den Zuständen, die ihre Kinder einnehmen können.*

$$\Sigma(c) \subseteq \{S(\Phi(c)_0) \cup S(\Phi(c)_1 \cup \ldots S(\Phi(c)_n)\} \mid n = |\Phi(c)| \tag{3.81}$$

***Definition 49*** *Gegeben sei eine KSM c. Sie kann nur ein aktives Kind zu einem Zeitpunkt besitzen und das aktive Kind ist gleich dem aktuellen Zustand. s entspricht dabei dem aktuellen Ausführungszustand.*

$$\Phi(c)_s \in E \tag{3.82}$$

Daraus lassen sich folgende Eigenschaften ableiten:

***Definition 50*** *Falls eine Komposition keine Zielzustände besitzt, ist sie immer aktiv, sofern sie im Komponentennetzwerk verwendet wird. Sie beendet sich von sich aus nie.*

$$F(c) = \emptyset \tag{3.83}$$

***Definition 51*** *Falls sich eine Komposition in einem Zielzustand befindet, kann sie nicht aktiv innerhalb des Komponentennetzwerkes verwendet werden.*

$$s_{current} \in F(c) \rightarrow a_c = 0 \tag{3.84}$$

Die Definition 51 beschreibt eine wesentliche Eigenschaft der zeitlichen Behandlung bzw. der Definition von Roboterverhalten. Falls eine Komponente in einen Zustand wechselt, in dem sie nicht mehr aktiv ist (Endzustand), ist sie nicht mehr als aktive Komponente im Netzwerk benutzbar. Diese hierarchische Kapselung wurde bereits von mehreren Arbeiten, wie [48] und [49], beschrieben. Die Eigenschaft der Abhängigkeit ist jedoch von zentraler Bedeutung für die Behandlung von Komponentennetzwerken. Diese Arbeit setzt dabei erstmals die vorgestellten Konzepte in einer formalen Definition für Komponentennetzwerke ein.

Diese hierarchische Kapselung erlaubt die Bildung von funktionalen Teilbäumen basierend auf verschiedenen Anforderungen. Um die Funktionsweise dieser hierarchischen Kapselung von zeitlich unbegrenzten und begrenzten Abfolgen zu verdeutlichen, sei an dieser Stelle ein Beispiel skizziert.

Gegeben sei das Szenario der Pipelineverfolgung für ein AUV (wie in [2] beschrieben). Ziel ist das Auffinden einer Leckstelle auf einer Pipeline. Es ist dabei jedoch unbekannt, an welcher Position das Leck exakt ist. Es ist möglich, dass das AUV zunächst zum Ende bzw. zu einem Kontrollpunkt der Pipeline gelangt und dort umdreht. Das Bild 3.30 zeigt die Mission, die das AUV absolvieren soll.

Die Beispiele 3.31 bis 3.33 zeigen einen abstrakten Zustandsautomaten innerhalb eines Netzwerkes. Das Bild 3.31 stellt abstrahiert das Modell der KSM mit ihren Zuständen dar. In Bild 3.32 ist die Integration der StateMachine in ein bestehendes Netzwerk zu sehen. Da sich die FSM innerhalb des *Start* Zustandes befindet, ist die Kind-Abhängigkeit aufgelöst zu dem Requirement *PipelineFollower*. Die Transition zwischen dem *Start*- und *Surface*-Knoten in Bild 3.33 geschieht auf Basis einer Zustandsänderung des *PipelineDectors*. In diesem Modell propagiert der *PipelineFollower* das Event *leakDetected* weiter an seinen

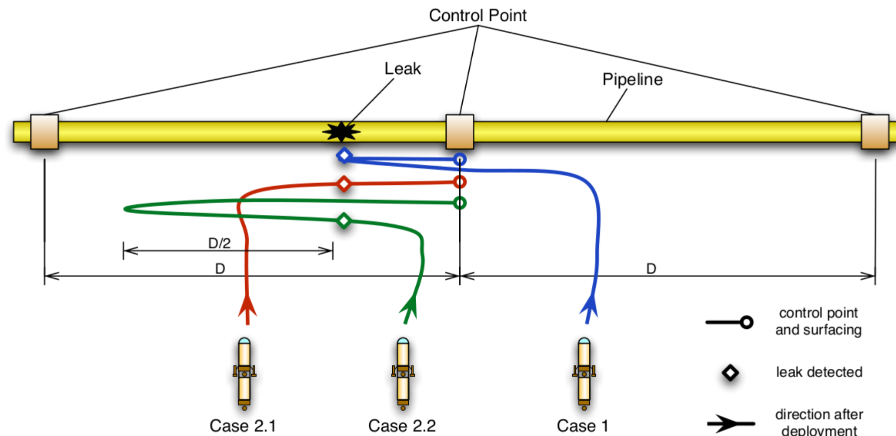



Figure 3.30: Beispielmission Pipelineverfolgung (Quelle: [2])

Elternknoten, in diesem Fall die Komposition *LeakDetection*. In diesem Szenario ist das Zusammenspiel zwischen dauerhaft aktiven und atomaren Anforderungen gut zu erkennen. Die Anforderung *PipelineDectors* ist an sich eine dauerhafte Anforderung, die durch die Einbindung in die KSM *LeakDetection* jedoch nur endlich lange ausgeführt wird.

Die Aufgabe *LeakDetection* ist somit eine einmalige Aufgabe, die sich aus verschiedenen Teilaufgaben zusammensetzt. Der PipelineFolower stellt eine dauerhaft aktive Instanz zur Verfügung. Obwohl PipelineFollower keine Daten bereitstellt, die die *LeakDetection* direkt benutzt, wird der Service *PipelineFollowing* kontinuierlich bereitgestellt. Erst durch die Kapselung in die *LeakDetection* wird das Verhalten endlich und terminiert unter den gegebenen Bedingungen. Von besonderer Bedeutung ist hier die funktionale Kapselung. Weder der *PipelineDetector* noch der *PipelineFollower* treffen Entscheidungen über das Systemverhalten. In anderen Architekturen, wie Subsumption [9], wird der Systemzustad durch die Module selbst definiert. Diese Homöostase muss implizit von allen Modulen als Kombinationsverbund erreicht werden, ohne dass eine höhere Steuerung erfolgt. Durch eine explizite Koordination eines Supervision-Layers in Verbindung mit einer klaren hierarchischen Modularisierung wird es in dieser Arbeit jedoch ermöglicht, dass der PipelineDetector unabhängig von einer Steuerung genutzt werden kann. Somit könnte in anderen Moulverbänden das gleiche Modul verwendet werden, ohne dass bereits aktive Steuerung und Kontrolle des Systems besteht.

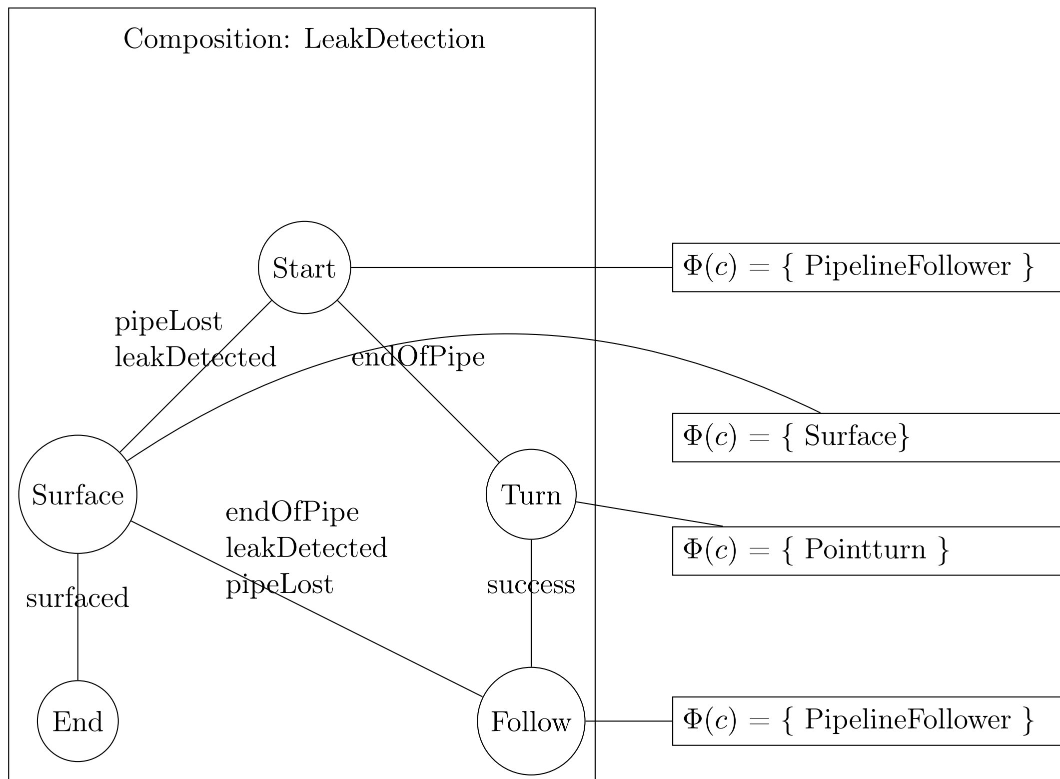



Figure 3.31: Abstrakte StateMachien-Beschreibung der Komposition LeakDetection

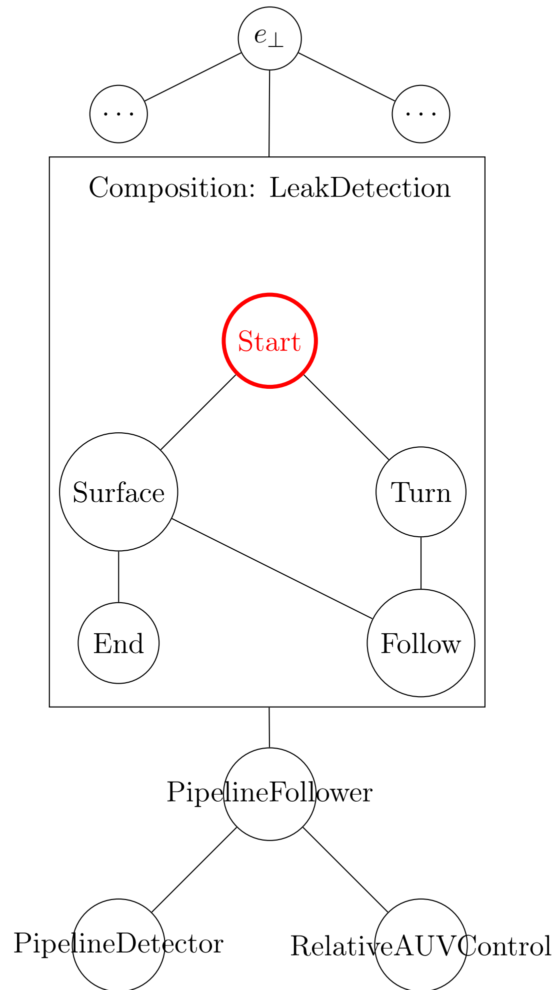



Figure 3.32: Abstrakte StateMachine innerhalb eines Abhängigkeitsgraphen

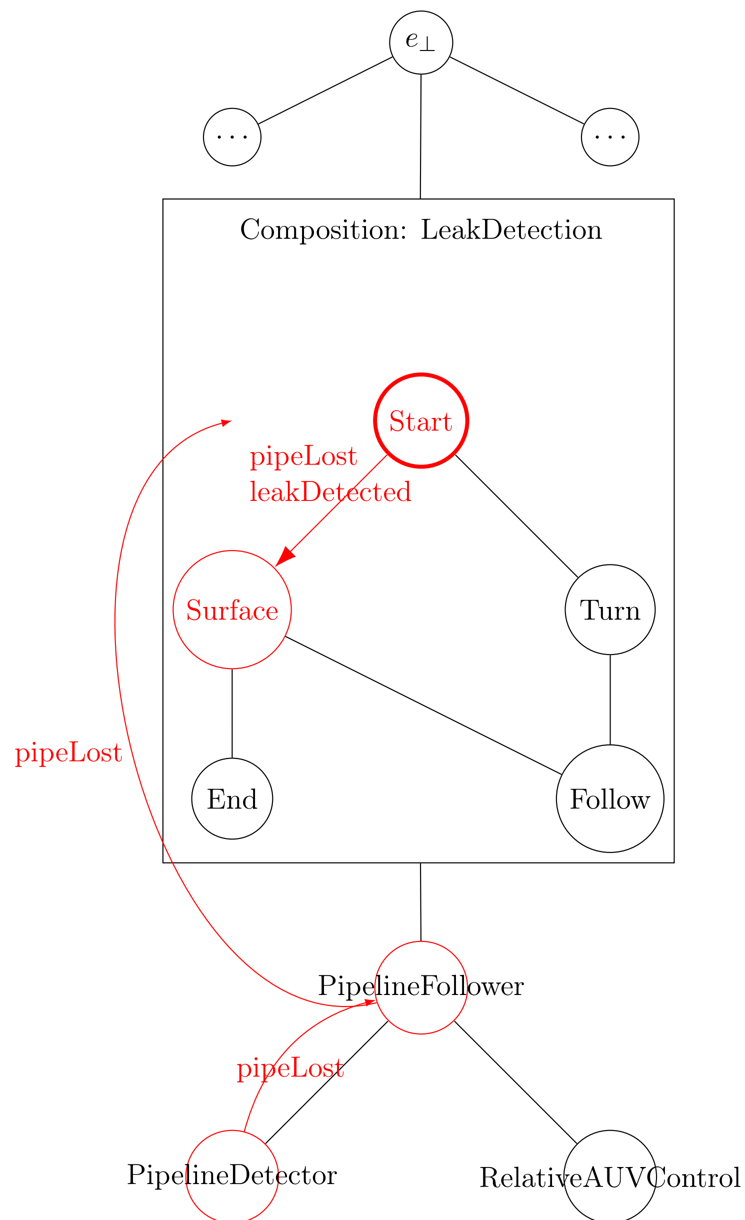



Figure 3.33: Einfluss einer Zustandsänderung eines Kindes auf die Ausführung

Zur Planung oder Bestimmung des Systemverhaltens ist es, aus der Sicht des Komponentennetzwerkmanagers, nur nötig, dass die Kinder im Abhängigkeitsgraph ihre Zustandsinformation an den Elternknoten propagieren. Dabei ergeben sich zwei Eigenschaften, die das System erfüllen muss.

- Kann ein Kind seine Aufgabe nicht mehr absolvieren, muss eine Reaktion stattfinden. Dies ist der Fall, falls ein Knoten durch einen (beliebigen) Fehler ausfällt oder er seine Aufgabe (erfolgreich) absolviert hat und damit seine Arbeit einstellt. Diese Zustandsänderung ist nicht abfangbar und erfordert eine Behandlung von einer höheren Ebene.

- Eine Komponente meldet eine Zustandsänderung in ihrem Ausführungszustand, kann jedoch weiter operieren.
  Beispielsweise wurde eine Pipeline detektiert oder das Ende der Pipeleine erreicht. In beiden Fällen kann die Komponente weiterarbeiten, erfordert jedoch i. d. R. eine Behandlung einer übergeordneten Schicht. Die Entscheidung, welcher und ob ein neuer Folgezustand nötig ist, hängt dabei von der aktuellen Aufgabe des Gesamtsystems ab. Diese Änderungen können behandelt oder ignoriert werden, da sich das System weiterhin in einem definierten kontrollierten Zustand befindet und das Komponentennetzwerk weiterhin in der Lage ist, die gegebene Anforderung zu erfüllen.

Um dieses zu formalisieren, muss das StateMachine-Modell erneut um die Propagation von Zuständen erweitert werden.

***Definition 52*** *Eine Komposition kann die Zustände ihrer Kinder propagieren oder ignorieren. Dabei wird festgelegt, ob eine Propagation stattfindet, indem:*

$$\zeta(c) = \{(s_1, s_2 \in S(c))\} \tag{3.85}$$

*ein Kindzustand $s_1 \in \Phi(c)_{i_s}$ resultiert in einem Zustand $s_2 \in S(c)$ der Komposition $c$.*

Um diese Anforderungen formal zu definieren, wird das Event/Action-System, das durch die Arbeit [33] entworfen wurde, abgewandelt, sodass es die Propagation von Zuständen auf Modellbasis ermöglicht. Jeder Task und jede Komposition besitzen eine Menge an Zuständen $S$, in denen sie sich befinden können. Falls eine Entität im Netzwerk ihre Aufgabe nicht mehr wahrnehmen kann, ist sie fehlgeschlagen und dieser Fehler muss auf dem Pfad zur Wurzelabhängigkeit propagiert werden. Falls die Propagation den Wurzelknoten erreicht, ist das komplette Netzwerk fehlgeschlagen. Eine Komponente ist dann fehlgeschlagen, wenn sie selbst algorithmisch einen Fehler meldet (Programmabstürze, Exception-States), oder wenn Kompositionen einen Zielzustand erreicht haben und ab da

nicht weiter operieren können. Ein Beispiel für Letzteres ist, wenn eine StateMachine in einen Endzustand übergeht.

Es ist relevant, dass selbst der Zielzustand in Standardfall einen Fehlerfall darstellt. Gegeben sei das Beispiel aus 3.31. Sollte beispielsweise die LeakDetection in den Endzustand übergehen, ist unklar, ob die Mission erfolgreich war oder ob sie sich beendet hat, weil niemals ein *Leak* gefunden wurde. Da das Ziel dieser Abhandlung ist, ein möglichst wenig fehleranfälliges System zu entwerfen, sind alle nicht erwarteten Zustände per Definition ein Fehler. Hierbei gilt die Annahme, dass es einfacher möglich ist, einen erwarteten Erfolgszustand explizit zu propagieren, als alle möglichen Fehlerfälle vorauszusehen bzw. in den Modellen zu berücksichtigen. Die formale Überprüfung von sequentiellem Verhalten geschieht dementsprechend durch Propagation aller Kindzustände aller im Netzwerk befindlichen Tasks. Dabei kann es zu den folgenden Fällen kommen:

- Ein Nicht-Fehlerzustand wird vom Elternteil nicht propagiert.
  In diesem Fall wird das Event nicht weiter beachtet und es ist für die Sequenzierung irrelevant, da kein Folgezustand auf diesem Event basiert.
- Ein Nicht-Fehlerzustand wird von einem Elternknoten propagiert.
  In diesem Fall verhält sich die Komposition selbst wie ein Task. Das Event wird propagiert, als sei es innerhalb des Elternknotens aufgetreten. Dessen Elternteil kann wiederum entscheiden, ob dieses Event relevant ist und behandelt oder propagiert werden soll.
- Ein Nicht-Fehlerzustand wird von einer KSM behandelt.
  In diesem Fall wird korrelierende Transitionsfunktion *delta* der Komposition ausgeführt, wodurch ein neues Folgenetzwerk bestimmt werden muss, da sich die Abhängigkeit $\Phi$ der KSM ändert.
- Ein Fehlerfall wird rekursiv aufsteigend vom keinem Elternknoten behandelt.
  Dies bedeutet, dass ein nichterwarteter Zustand aufgetreten ist. Da der Fehler bis zur Wurzelabhängigkeit nicht behandelt wurde, sind sämtliche Anforderungen fehlgeschlagen. Dies stellt den kritischsten Fehlerfall dar. Das komplette Systemverhalten muss hieraufhin abgebrochen werden.
- Ein Fehlerfall wird von einer StateMachine auf dem Weg zur Wurzelabhängigkeit behandelt.
  In diesem Fall muss eine Transition der StateMachine ausgeführt werden. Alle bisherigen Abhängigkeiten der StateMachine müssen als ungültig deklariert werden. Die StateMachine ist dafür zuständig, einen Folgezustand als neue Kindabhängigkeit $\Phi$ bereitzustellen, was die bereits angesprochene Neuberechnung des kompletten Netzwerkes erfordert.

Es kann vorkommen, dass ein Knoten, der eine Transition durchläuft, in mehreren Teilbäumen vorkommt. Die Teilbäume müssen dabei dieses Event getrennt voneinander behandeln. Falls eine geteilte Komponente einen Fehler aufweist, müssen beide Teilbäume diesen Fehlerzustand behandeln, da sonst (wie in Punkt 4) das komplette Netzwerk fehlschlagen müsste. Ein Beispiel für eine solche Integration einer Komponente ist in Abbildung 3.34 dargestellt.

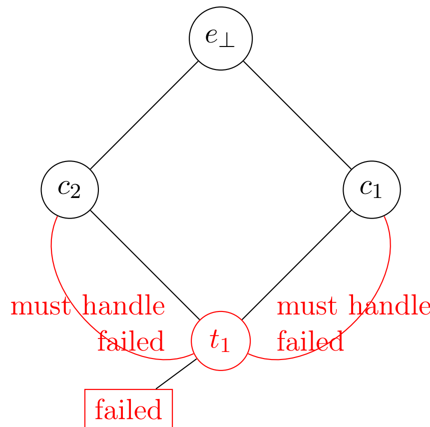



Figure 3.34: Fehler müssen von allen Eltern bzw. beiden Teilbäumen behandelt werden

Da die möglichen Events nach Abschluss der Modellierung ebenso wie die Menge aller Tasks und Anforderungen bekannt sind, lässt sich das komplette Sequenzverhalten von Netzwerken auf Modellebene bestimmen und verifizieren. Jedes beliebige Netzwerk besitzt eine endliche Menge an Zuständen. Diese endlichen Zustände können nur von einer endlichen Menge von StateMachines behandelt werden. Die Folgezustände zu einem Netzwerk sind aus diesem Grund bekannt. Ein Folgezustand entspricht einem Folgenetzwerk. Dieses Netzwerk ist ebenso analysierbar wie zum ersten Zeitpunkt. Es ist möglich, dass durch die StateMachines Zyklen innerhalb des Verhaltens entstehen. Jedoch ist der Zustand der StateMachines für einen Folgezustand bekannt. Somit können Sequenzen im Verhalten einfach erkannt werden. Diese Sequenzen können in Abhängigkeit von der Missionsanforderung gewünscht oder unerwünscht sein. Beispielsweise kann es sein, dass eine Pipeline immer von einem zum anderen Ende (als permanente Inspektionsaufgabe) verfolgt wird. Dies würde in einem unendlichen Zyklus für das Verhalten resultieren. Eine höhere StateMachine könnte dieses Verhalten jedoch nur solange aktivieren, bis die Batterie einen gewissen Ladezustand unterschreitet oder bis mittels eines Kommunikationskanals ein Abbruchswunsch in das System eingeschleust wird.

An dieser Stelle sei darauf hingewiesen, dass trotz vollständiger Vorberechnung sämtlicher Zustände ein System weiterhin von außen steuerbar bleibt. Es wäre beispielsweise

möglich, gezielt von außen Komponentenzustände zu beeinflussen. Diese extern eingeschleusten Nachrichten (wie in Abbildung 3.35 skizziert) würden dann zu einem Zustandswechsel des Systems führen. Die externen Einflüsse sind, aus Sicht der Komponentennetzwerke, vorherbestimmt und weiterhin endlich. Somit ist das vorgestellte System weiterhin in der Lage, von Operatoren gesteuert zu werden.

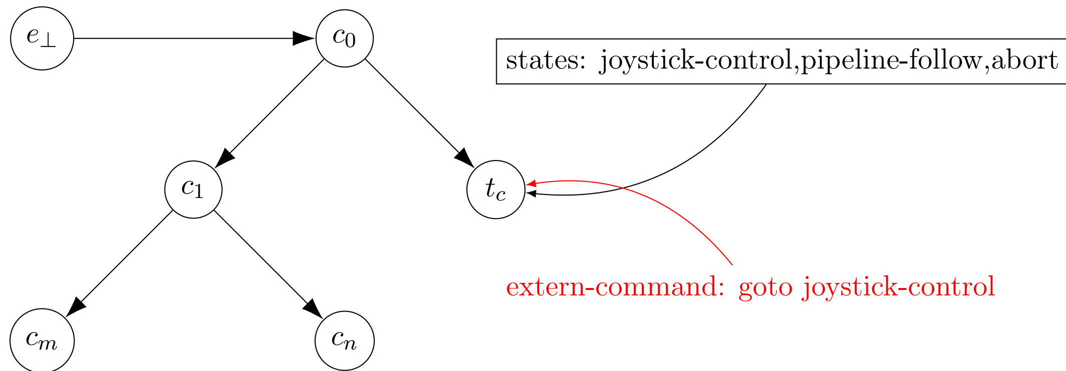



Figure 3.35: Beispiel für eine externe Steuerung, der Control Task $t_c$ wechselt sein Zustand in Abhängigkeit externer Kommandos. Daraufhin würde $c_0$ ein anderes Netzwerk anfordern

### 3.1.5 Zusammenfassung der formalen Modelleigenschaften

An dieser Stelle folgt die Zusammenfassung sämtlicher Modellattribute, die für die constraintbasierte Modellierung von Komponentennetzwerken und der zeitlichen Abfolge zur Missionsgenerierung im Rahmen dieser Arbeit definiert wurden. Diese Aufzählung dient dem Verständnis der Modellbeschreibungen. Die folgenden Definitionen beschreiben die Modellparameter, die der Modelldesigner festlegen muss. Nicht Bestandteil der Zusammenfassung sind die internen Variablen des Planers, die in den vorherigen Kapiteln der Klassenlösung (Seite 98) bzw. der Instanzlösung (Seite 118) genutzt wurden, um eine Lösung für Komponentennetzwerke identifizieren zu können.

- $T = \{t_1, \dots, t_n\}$ entspricht der Menge aller Tasks
- $C = \{c_1, \dots, c_n\}$ die Menge aller Kompositionen
- $D = \{d_1, \dots, d_n\}$ die Menge aller DataServices
- $E = T \cup C \cup D$ die Menge aller Entitäten
- $\succ (A, B) = \{0, 1\}$ definiert, ob $A$ die Anforderungen von $B$ erfüllt
- $\Phi(c) = [e_1, \dots e_n]$ der Vektor aller Kinder einer Komposition
- $\Pi = (\alpha, \beta)$ eine Konfiguration mit dem Namen $\alpha$ und einer Wertebereichsdefinition $\beta$
- $\Psi(e) = \{\Pi\}$ die Zuordnung von Konfigurationen zu Entitäten
- $\Gamma(c \in C) = \{(\Pi, e \in E(e \in \Phi(c)), \Pi)\}$ die Konfigurationspropagation von Kompositionen
- $\Xi(t \in T) = n \in \mathbb{N}$ die maximale Anzahl der gleichzeitigen Instanziierungen eines Tasks $t$
- $S(c) = \{s_0, \dots, s_n\}$ die States einer Kompositions-StateMachine KSM
- $s = (e, n)$ der Zustandsdefinition, bestehend aus einer referenzierten Entität und einem Namen $n$
- $F(c) = \text{failed}, \text{success}, f_2, \dots, f_n$ die Menge der Zielzustände einer KSM
- $\delta(c) = \{(s_1, \Phi(c)_{1_s}, s_2\}$ die Menge der Transition einerKSM, wobei $s_1$ bei Zustandsänderung des Kindzustands $\Phi(c)_{1_s}$ zu $s_2$ transitiert
- $\zeta(c) = \{(s_1, s_2)\}$ die Propagation der Kindzustände, wobei der Kindzustand $s_1 \in \Phi(c)_{i_s}$ zu dem Zustand $s_2 \in S(c)$ von $c$ propagiert wird

### 3.1.6 Ergebnisse der constraintbasierten Komponentennetzwerkbehandlung

#### 3.1.6.1 Vorstellung der verschiedenen Testmodelle

Die folgende Evaluation zeigt diverse Modelle, die betrachtet wurden, um die Funktionsweise des vorgestellten Verfahrens zu analysieren. Es soll gezeigt werden, dass sich die generierten Lösungen für die Anwendung auf robotischen Systemen eignen und die Lösungen nicht nur in Anbetracht der definierten Constraints valide sind, sondern auch sinnig und in sich eine konsistente Lösung darstellen. Insbesondere soll gezeigt werden, dass die zuvor identifizierten Probleme mit dem erarbeiteten Ansatz gelöst werden können.

**Minimales Start-Testmodell** In dem ersten Experiment wird ein Systemmodell erstellt, das in der Lage ist, einen hierarchischen Graphen aus Kompositionen aufzubauen. Modell 1 stellt dabei das Modellwissen vor.

Das Modell besteht dabei aus drei Kompositionen: dem *root-knot* und den Kompositionen $A$, $B$. Des Weiteren existiert ein Task $T$, das als Kind von $B$ definiert wurde. $B$ selbst ist dabei das Kind von $A$. $A$ wurde selbst als aktiv deklariert, deswegen ist $A$ eine Abhängigkeit des Wurzelknotens. In diesem Beispiel ist leicht zu verstehen, dass eine Abhängigkeitskette $A \rightarrow B \rightarrow T$ besteht. Das Ergebnis ist in Abbildung 3.37 zu sehen.

Wie zu erkennen ist, entspricht der Abhängigkeitsgraph in Abbildung 3.37 der erwarteten Lösung. Es wurde nur eine Lösung gefunden und zugeordnet. Die Abhängigkeitskette zwischen allen Kompositionen bis zum Task $T$ wurde korrekt bestimmt. Die Beschriftung der Kanten beschreibt hier (und in allen folgenden Experimenten) die Zuordnung des Kindes aus dem Modell 1.

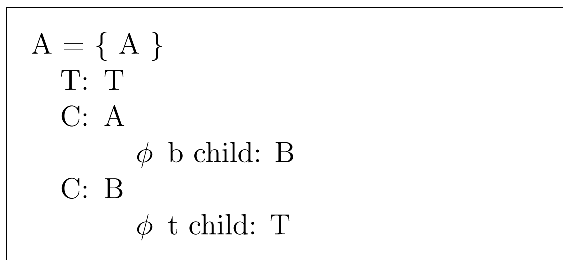



Figure 3.36: Minimales Testmodell zum Starten von Kompositionen und Tasks. Dabei ist: als Aktiv deklariert *{A}*, ein Task *T* sowie die Kompositionen *A,B* mit den korrespondierenden Kindern *B,T*

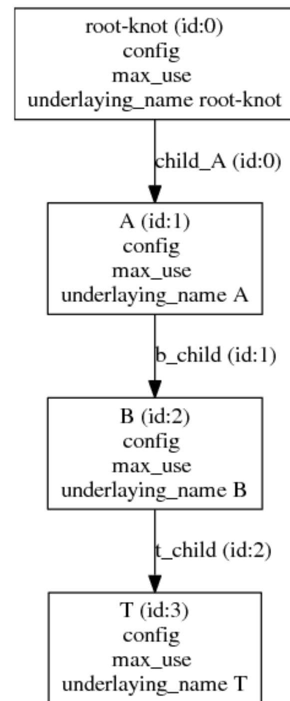



Figure 3.37: Programmausgabe des Solvers. Zu sehen ist das aufgelöste minimale Start-Modell. Erkennbar ist die Abhängigkeitskette $root \rightarrow A \rightarrow B \rightarrow T$.

**Testfall für nicht benötigte Elemente** Der nächste Testfall beschreibt ein weiteres, komplexeres Beispiel für ein Komponentennetzwerk. Hier werden nicht alle Kompositionen benötigt, da manche keine aktive Rolle in diesem Szenario haben. Es existieren die Kompositionen *B, C* sowie der Task *T1*. Diese bilden einen funktionalen Verbund, werden jedoch für das aktuelle Problem nicht benötigt. Es wird erwartet, dass nur eine Lösung identifiziert wird und die nicht aktiven Komponenten *B, C.T1* nicht innerhalb des Abhängigkeitsgraphs vorkommen.

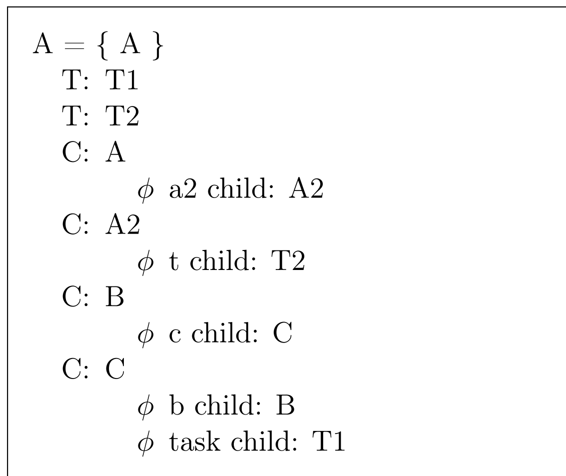



Figure 3.38: Modell mit nicht benötigten Elementen

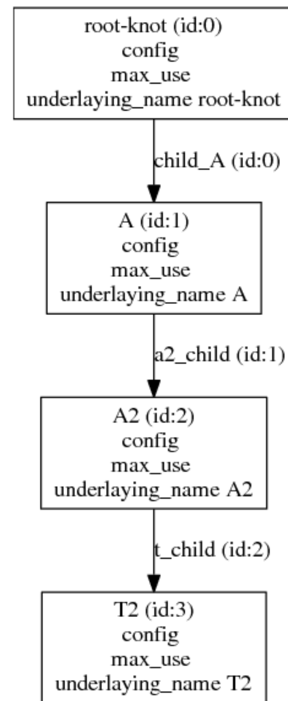



Figure 3.39: Testmodell mit ungenutzten Elementen

Die Abbildung 3.39 zeigt die errechnete Lösung. Bei der korrekten Identifizierung der Lösung spielt das Constraint aus Definition 25 eine wesentliche Rolle. Die Kompositionen *B* und *C* könnten einen Kreis bilden, indem sie von sich selbst abhängen und sich somit gegenseitig benötigen. Weiterhin relevant für diesen Testfall ist die Definition 3.24, die besagt, dass es keine losgelösten Verbunde geben darf.

**Testfall mehrdeutige Lösungen** Ein weiterer Testfall ist die Identifizierung von mehrdeutigen Lösungen. Wie eingangs im Abschnitt 3.1.4.3 beschrieben, ist das Bestimmen und Erlauben von mehrdeutigen Lösungen die Grundidee, um die Robustheit und Fehlersicherheit von komplexen Systemen, unter Anbetracht einer gegebenen Mission, bestimmen zu können. Mehrdeutige Lösungen entstehen, wenn eine Anforderung (konkret ein DataService) durch mehrere mögliche Quellen erfüllt werden kann. Das Modell 3 definiert hierfür den Testfall. Es werden ein DataService *DS* und eine Komposition *A*, die den DataService *DS* benötigt, definiert. Der DataService *DS* selbst kann von den Tasks *T* sowie *T2* erfüllt werden ($\succ$).

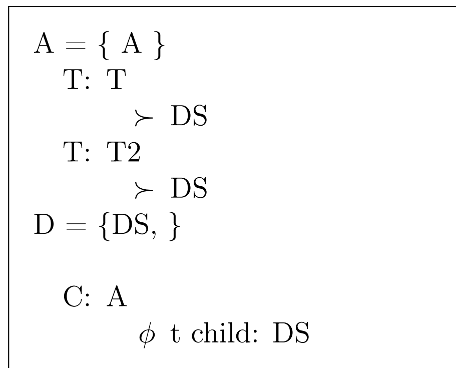



Figure 3.40: Modelldefinition eines Mehrdeutigkeitsproblems, $T, T2$ erfüllen den DataService *DS*, der von der Kompositiotn *C* gefordert wird.

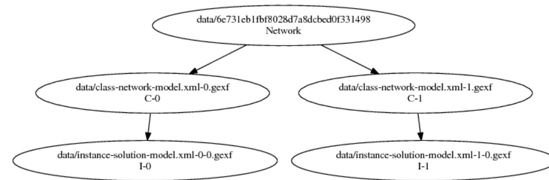



Figure 3.41: Lösungsgraph für mehrdeutige Lösung. Zu erkennen ist, dass zwei Klassenlösungen ($C-1, C-2$ mit jeweils einer Klassenlösung ($I-1, I-2$) erstellt wurden

Das Ergebnis der Auswertung des Modells 3 ist in Abbildung 3.41 zu sehen. Die Grundanforderung des Wurzelknotens resultiert in zwei verschiedenen Klassenlösungen *C-0, C-1*. Jede Klassenlösung besitzt im Folgenden jeweils eine Instanzlösung *I-0, I-1*. In den Abbildungen 3.42 und 3.43 sind die korrespondierenden Lösungen gezeigt. Zu sehen ist, dass in der Lösung *I-0* der Task *T* als Kind *t_child* der Komposition *A* gewählt wurde. Im Fall der Lösung *I-1* hingegen wird der Task *T2* als Kind selektiert. Beide Lösungen erfüllen die Modellanforderung und stellen gültige Lösungen für das Problem dar.

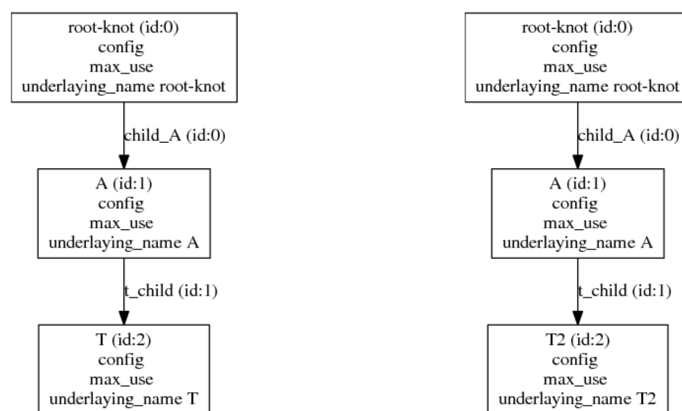



Figure 3.42: Instanzlösung 1 (I-0) Figure 3.43: Instanzlösung 1 (I-1)

Figure 3.44: Lösungen für mehrdeutige Anforderungen, beide gültigen Abhängigkeitsgraphen wurden korrekt bestimmt.

**Validierung von Konfigurationseigenschaften** Die bisherigen Modelle evaluierten die Generierung von Abhängigkeiten und verschiedenen Problemstellungen. Das Modell 4 evaluiert die Propagation von Konfigurationsattributen über Kompositionen hinweg. Interessant ist die Propagation der *compostion-config* aus der Komposition *ConfigurableComposition 4*. Der gesetzte Wert der *compositon-config* soll an das Kind propagiert werden, das für *task* selektiert wird. Diese Konfiguration soll im weiteren Verfahren final auf das Attribut *task-config* geschrieben werden. Wie in Abbildung 3.46 zu sehen ist, wurde sowohl die Konfiguration auf der Komposition als auch auf dem Task korrekt festlegt.

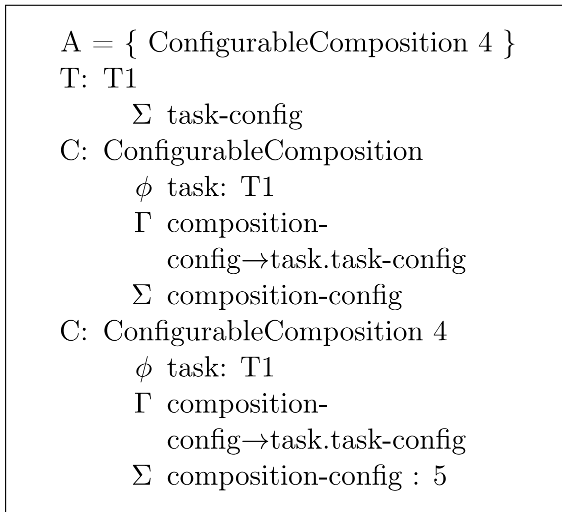



Figure 3.45: Modelldefinition

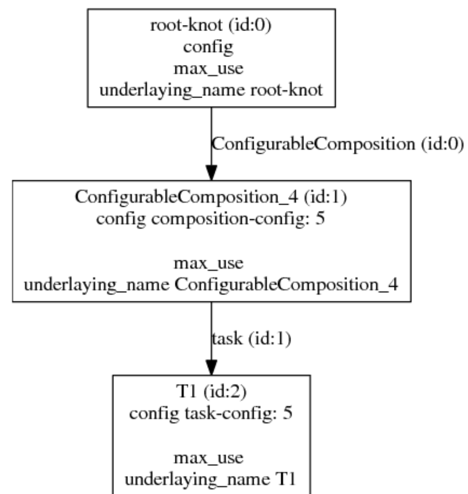



Figure 3.46: Ergebnis

Figure 3.47: Modelle der Propagation von Konfigurationen

**Modell von mehreren Anforderungen an eine Sensorkonfiguration** Das folgende Modell 5 zeigt anhand des mehrfach erwähnten Grundproblems, das zu dieser Arbeit führte, dass das neue Konzept in der Lage ist, mit mehrdeutigen Anforderungen umzugehen. Das Modell besteht aus den (vereinfachten) Modulen: Lokalisierung und Wandverfolgung. Beide Einheiten benötigen das Sonar in unterschiedlichen Konfigurationen. Die Modelle sollen so aufgelöst werden, dass sämtliche Freiheiten bis zur Endlösung an das Sonar propagiert werden.

```
A = { Localization 5 }
  T: Sonar
       Σ scan-angle
       Ξ 1
  C: Localization
       φ sonar: Sonar
       Γ sonar-angle→sonar.scan-angle
       Σ sonar-angle
  C: Wall
       φ sonar: Sonar
       Γ sonar-angle→sonar.scan-angle
       Σ sonar-angle
  C: Localization 5
       φ sonar: Sonar
       Γ sonar-angle→sonar.scan-angle
       Σ sonar-angle : 0 to 360
  C: Wall 6
       φ sonar: Sonar
       Γ sonar-angle→sonar.scan-angle
       Σ sonar-angle : 70 to 120
```

Model 5: Modell einer mehrdeutigen Sensorkonfiguration

Abbildung 3.48 zeigt dabei die Lösung der Anforderung für $A = \{Wall\ 6\}$. Wie erwartet wurde die Konfiguration an das Sonar weiter propagiert. Die möglichen Öffnungswinkel entsprechen dabei $70 - 120°$. Im Gegensatz dazu zeigt Abbildung 3.49 die aktive Lokalisierung mit der entsprechenden Konfiguration. Diese einzelnen Anforderungen an das Netzwerk entsprechen der Erwartung und wären auch mit den bestehenden Verfahren lösbar gewesen, da es keine sich gegenseitig beeinflussenden Rahmenbedingungen gibt.

Die Abbildung 3.50 zeigt hingegen, wie sich zwei unabhängige Teilbäume (der Lokalisierung und Wandverfolgung) gegenseitig beeinflussen. Es führt dazu, dass das Modell das Sonar mit dem eingeschränkten Öffnungswinkel definiert. Diese Lösung ermöglicht es somit, flexible, auch konkurrierende Anforderungen an Geräte auf einer Modellbasis zu behandeln.

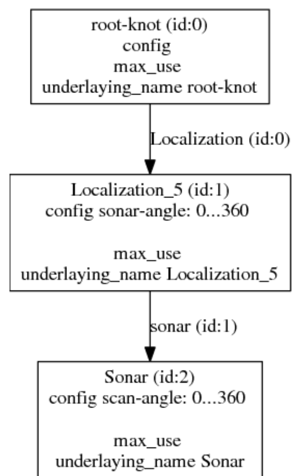



Figure 3.48: Aktives Wall-Servoing des Modells 5

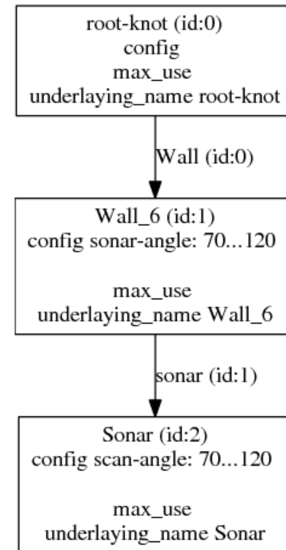



Figure 3.49: Ergebnis: nur Lokalisierung des Modells 5

Abbildungen 3.52 und 3.51 zeigen erneut die Bedeutung von $Xi = 1$ für das Sonar. Würde diese Einschränkung nicht gesetzt werden, wäre es möglich, das Sonar zweimal zu instanziieren. Dies mag aus der Softwaresicht sinnig erscheinen, da jedoch das Gerät nur von einem Treiber genutzt werden kann, würde dies zu einem Fehler führen, der durch das Attribut $Xi$ vermieden werden kann.

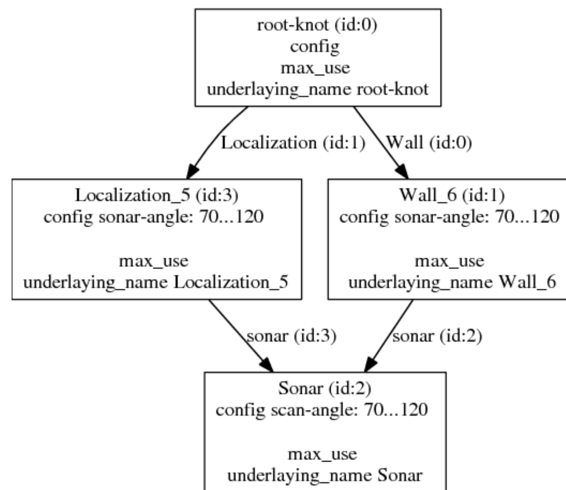



Figure 3.50: Lokalisierung und Wall-Servoing des Modells 5. Beide Anforderungen zugleich konkurrieren in der Sonarkonfiguration, können jedoch bestimmt werden

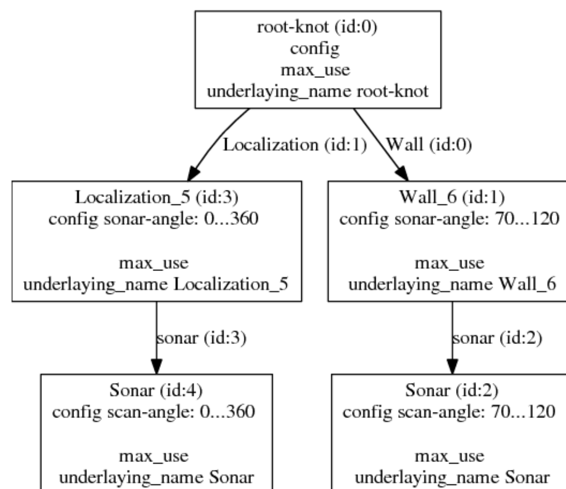



Figure 3.51: Mehrfach instantiiertes Sonar, I-0 aus Abbildung 3.52

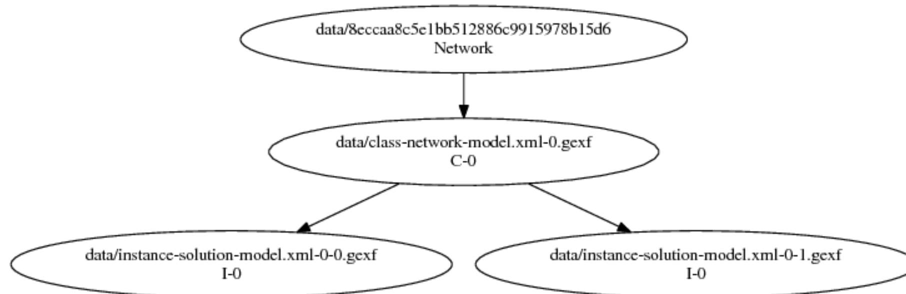



Figure 3.52: Das Modell 5 ohne Einschränkung von $Xi = 1$ für das Sonar

**Avalon Modell** Die Abbildung 3.53 zeigt das Verhaltensmodell für die Demonstrationsmission, wie im Abschnitt 2.1.3, beschrieben. In der Abbildung sind, im Gegensatz zum vollständigen Graphen 3.54, nur die Klassenlösungen dargestellt. Sämtliche vorgestellten Graphen sind direkte Ergebnisse aus der Implementierung der Modelle. Wichtige Details sind hervorgehoben, um die konzeptuelle Funktionsweise des Algorithmus darzulegen. In dem Graphen 3.53 wird dabei nicht zwischen verschiedenen Instanzlösungen unterschieden, um den Fokus auf die Sequenzierung zu legen. Die Knoten des Graphen beschreiben dabei komplette Netzwerke. Die Netzwerke werden lediglich anhand einer ID identifiziert, da sie keinen Namen besitzen. Ihre Generierung erfolgte vollautomatisch auf Modellebene basierend auf den Events, denen das System unterliegen kann. Eine mögliche (kürzeste) Sequenz ist in den Knoten des Graphen annotiert. Die Kanten zwischen den Netzwerken beschreiben dabei ein mögliches Event, welches zu diesem Wechsel führen kann. Das Modell zeigt somit den Verhaltenszyklus, der in Abschnitt 2.1.3 eingeführt wurde. Der Graph 3.54 ist hingegen die expandierte Version des Graphen 3.53. Dieser expandierte Graph beinhaltet sämtliche Instanz- und Klassenlösungen, sowie sämtliche Events, die zu den Transitionen zwischen Netzwerken führen.

Der Verbund *B* beinhaltet beispielsweise die Events, die zu der Transition von Netzwerk *A2* zu Netzwerk *C* führen können. Somit sind die Events *B* die Konditionen für einen Verhaltenswechsel. Die Instanzlösungen *A1-A3* sind ein Indiz für mehrdeutige Lösungen. Es ist möglich die gleiche Anforderung mittels verschiedenen Netzwerken zu erfüllen. Im Detail sind die Lösungen *A1-A3* in Abbildungen 3.55 bis 3.57 dargestellt. Sie unterscheiden sich in der Art wie oft verschiedene Instanzen von Algorithmen instantiiert werden. Konkret werden in diesen Beispielen mehrere oder einzelne Fusionsalgorithmen instantiiert. Da diese Algorithmen mit den gleichen Sensorinformationen befüllt werden, würde sich hierbei kein funktionaler Unterschied ergeben.

Mittels des vollständigen Modells konnte gezeigt werden, dass der neue Ansatz zum bestehenden System kompatibel ist und auch mit großen Suchräumen effizient umgehen kann. Des Weiteren konnte gezeigt werden, dass auch das Systemverhalten über die Zeit mit dem neuen Konzept, ohne eigentliche Ausführung der Komponenten, analysierbar ist. Der Graph 3.53 zeigt dabei, dass es sich um ein unendlich sequenzierbares Verhalten handelt. Es besteht die Möglichkeit, dass direkt in ein Fallbackverhalten gewechselt werden kann, falls die Pipeline nicht gefunden wird.

Figure 3.53: Das reduzierte Sequenzdiagramm des Avalon Modells der Demonstrationsmission
Knoten beinhalten den eindeutigen generierten Bezeichner sowie den kürzesten Pfad zu diesem Zustandsknoten.
Kanten beinhalten das erste gefundene Event, welches zur Transition zwischen den Zuständen führt.

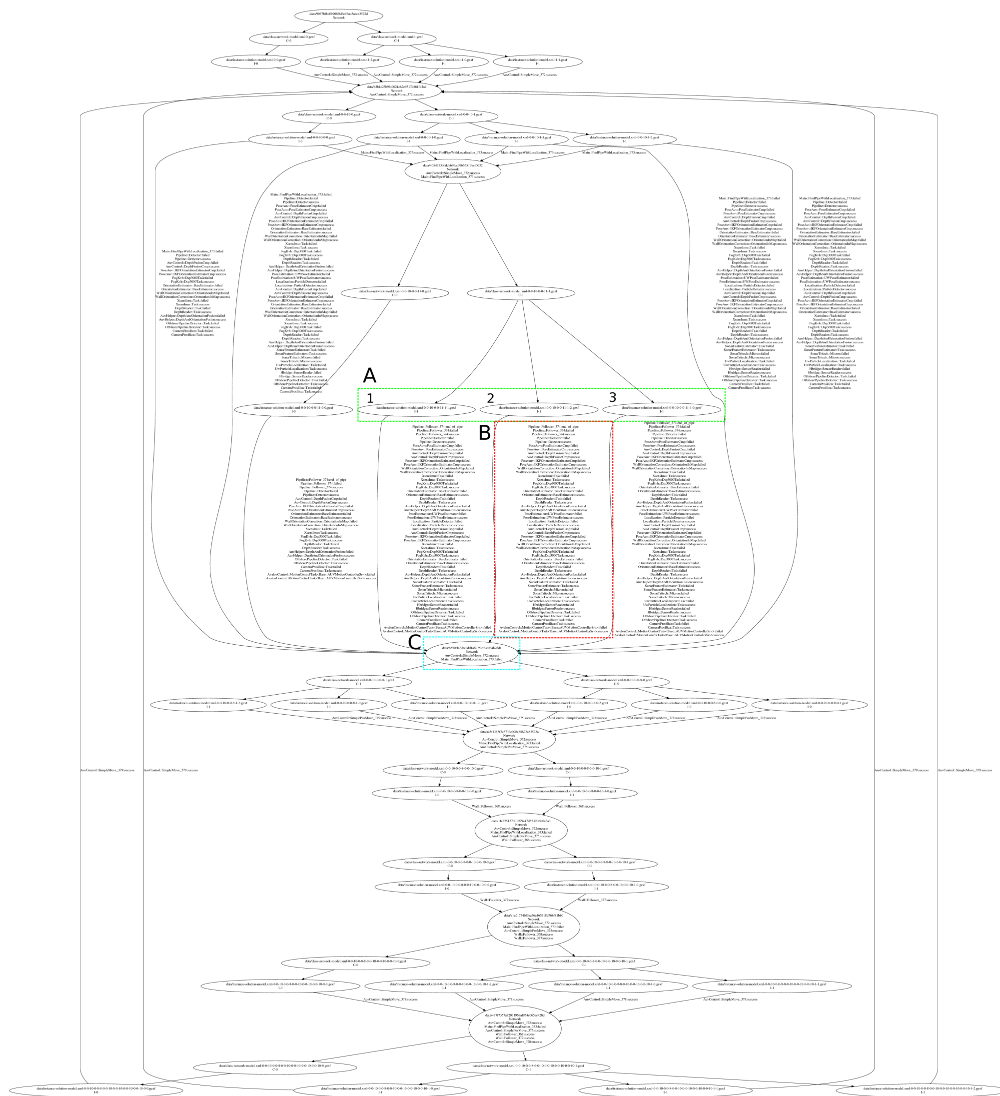



Figure 3.54: Expandiertes Modell von 3.53 mit allen Instanzlösungen und möglichen Events aus dem Modell
Knoten: Instanz/Klassenlösungen
Kanten: Transitionen zwischen Netzwerken
A: Verschiedene Instanzlösungen für ein Problem
B: Events, die zur Transition von A zu C führen
C: Folgezustand von A, gegeben des Auftretens eines Events aus B

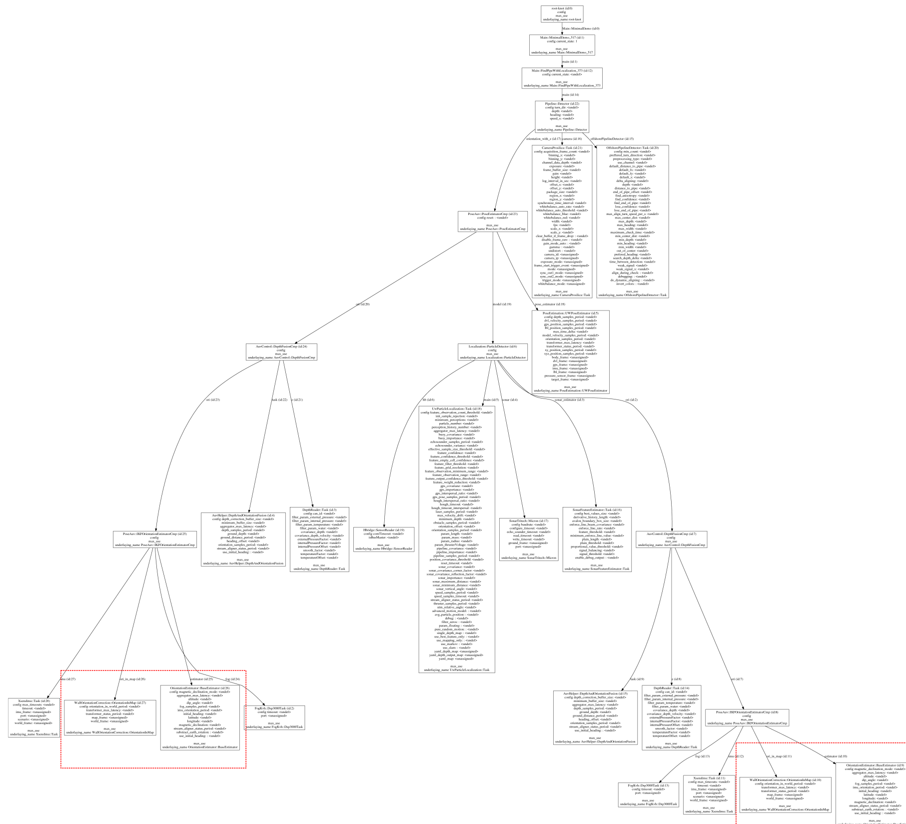


Figure 3.55: Komponentennetzwerk A1 aus 3.53 - einzelne Instantiierungen

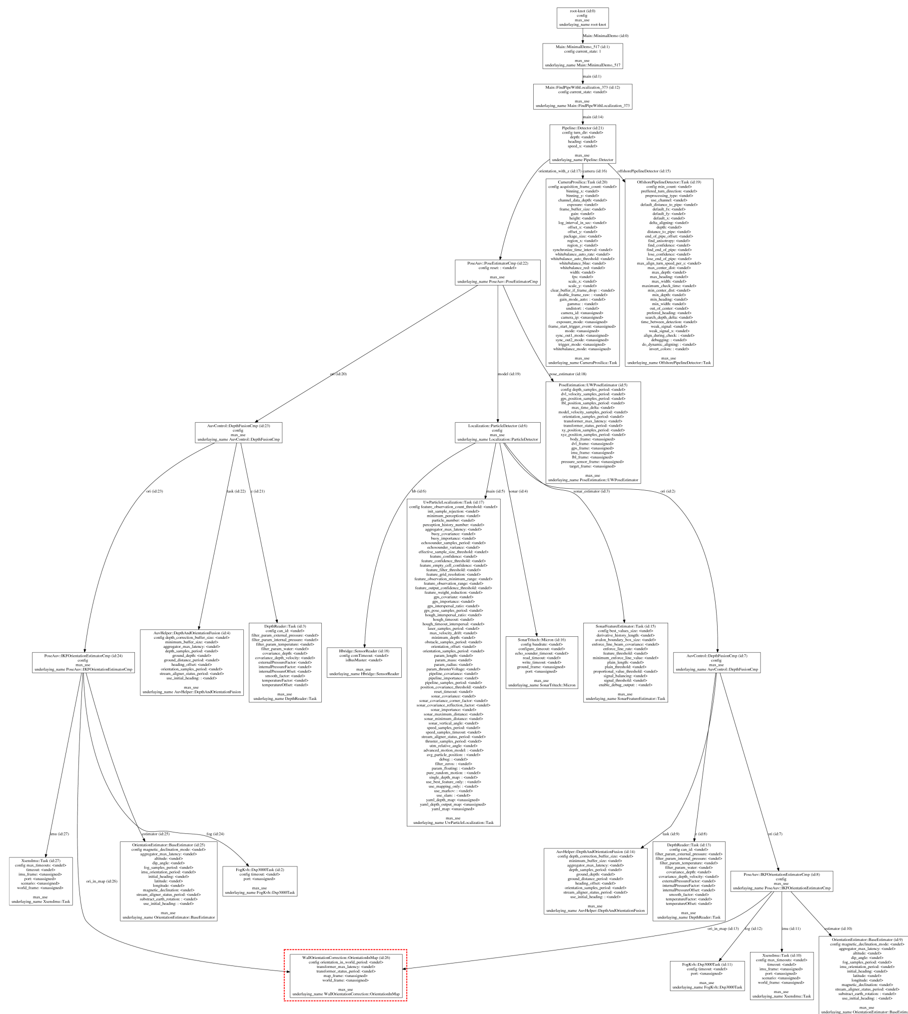


Figure 3.56: Komponentennetzwerk A2 aus 3.53 - geteilte Orientierungskorrektur

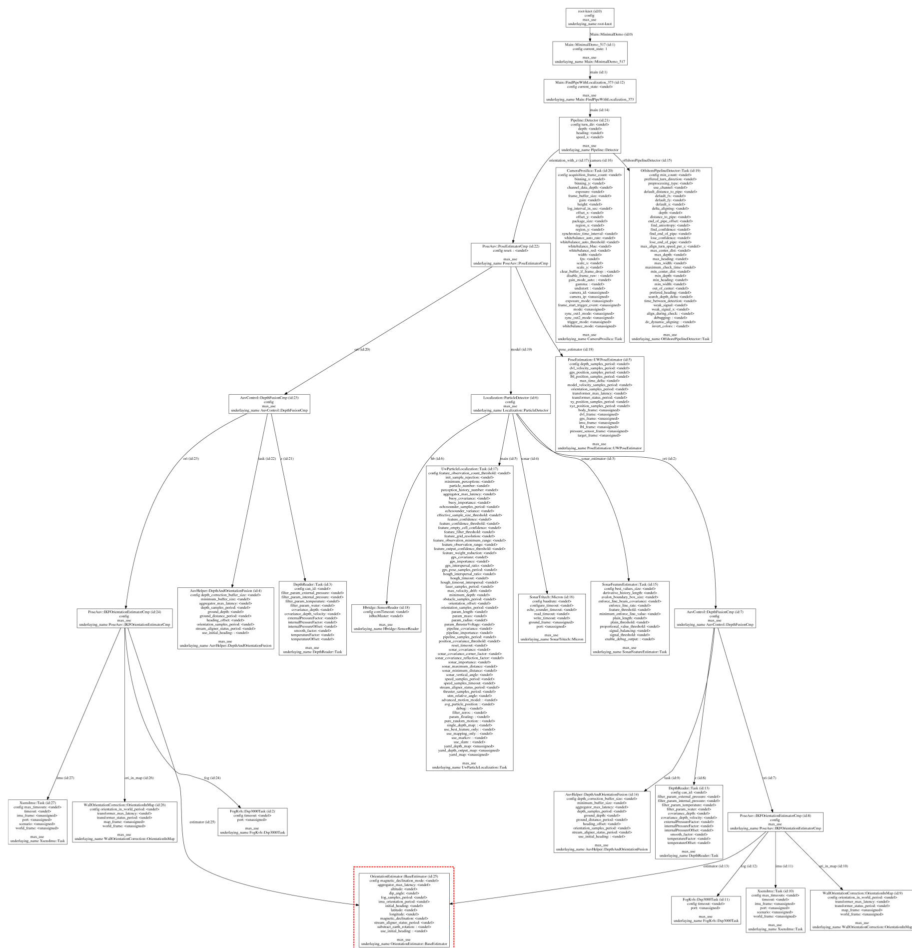


Figure 3.57: Komponentennetzwerk A3 aus 3.53 - geteileter Orientierungsschätzer

## 3.2 Zusammenfassung der constraintbasierten Netzwerkbehandlung

In diesem Kapitel der Arbeit wurde ein formales System erarbeitet, das in der Lage ist, Komponentennetzwerke zu behandeln. Im Gegensatz zu allgemeinen Sprachen wie PDDL wurden spezielle Konzepte bereitgestellt, die es erlauben, moderne robotische Systeme effizient modellgetrieben zu entwickeln.

Dabei wurde die Portierbarkeit der Modelle des Standes der Technik erhöht, indem globale Constraints eingeführt wurden. Anstatt wie in bisherigen Systemen Mehrdeutigkeiten zu verbieten, ist das neu entworfene Konzept in der Lage, diese Mehrdeutigkeiten automatisch aufzulösen. Dies reduziert die Portierung einzelner Konzepte auf neue robotische Plattformen erheblich und erleichtert somit die Verhaltensdefinitionen.

# 4 Zusammenfassung der Arbeit

In dieser Arbeit wurden zwei große Hauptbereiche wissenschaftlich untersucht. Einerseits wurde die Plattform Avalon entwickelt. Auf der Basis dieses Systems wurden die Herausforderungen moderner Robotikforschung untersucht. Als größte Herausforderung wurde die Behandlung von Komponentennetzwerken aufgezeigt. Diese Komponentennetzwerke wurden dann im zweiten Hauptteil dieser Arbeit mittels eines komplett neuartigen constraintbasierten Verfahrens interpretiert. Dies hatte zur Folge, dass ein innovatives, portierbares und formal verifizierbares System zum Systemverhaltensdesign entwickelt wurde.

## Avalon

In dem Kapitel 2 wurden intensiv die Probleme der Robotik anhand der Systeme Dagon und Avalon untersucht. Avalon wurde im Rahmen dieser Arbeit entwickelt und mit allen nötigen Fähigkeiten ausgestattet, um verschiedene komplexe Missionen zu absolvieren. Diese Fähigkeiten wurden während der Wettbewerbe SAUC-E und euRathlon unter Beweis gestellt. Die soliden Grundkonzepte führten hier zu verschiedenen Preisen in denen die Systeme über mehrere Jahre kontinuierlich ihre Fähigkeiten beweisen konnten.

- 2009 3. Platz beim SAUC-E
- 2011 3. Platz beim SAUC-E
- 2012 "best qualifier award" beim SAUC-E
- 2014 1. Platz beim SAUC-E
- 2014 3. Platz für das "Combined scenario" beim Eurathlon
- 2014 2. Platz für die "Leak localisation and structure inspection beim Eurathlon
- 2014 2. Platz für die "Interaction with underwater structures" beim Eurathlon
- 2014 1. Platz für das "Environmental survey of an accident area" beim Eurathlon

Bedingt durch die lange Entwicklungs- und Nutzungszeit des Systems war es möglich, tiefe und detaillierte Einsichten in verschiedene typische Probleme verschiedener Forschungsbereiche zu erlangen. Die gesammelten Ergebnisse führten einerseits zur stetigen Verbesserung des Systems. Hervorzuheben sind dabei beispielsweise das sicherere Steuerungskonzept, das intensiv in Abschnitt 2.2.1.5 beschrieben wurde. Es machte das System unter allen Umgebungsbedingungen sehr zuverlässig steuer- und benutzbar. Aber auch die Reglerkette und die Konzepte modularer Softwareentwicklung haben sich als wertvolle Methodiken herausgestellt. Diese Konzepte und Entwicklungen flossen in viele Nachfolge- und Parallelprojekte ein.

## Constraintbasierte Behandlung von Komponentennetzwerken

Der zweite Teil der Dissertation in Kapitel 3 stellt ein vollständig neues Konzept zur Behandlung von Komponentennetzwerken vor. Das neue Konzept bereinigt Design-Fehlentscheidungen aus vorangegangen Konzepten und bildet eine solide Basis für zukünftige Forschungen im Bereich der Komponetennetzwerkplanung. Auch wird die Lücke zwischen der Technischen Komponetennetzwerkplanung und der Verhaltensgenerierung verkleinert. Es wurde ein Übergang von dieser technischen Sichtweise zur Verhaltensplanung geschaffen. Dieses wurde in Abschnitt 3.1.4.5 dargelegt und erörtert.

Um das neu entworfene Konzept zu evaluieren, wurden die auf Avalon genutzten Modelle in die neu entwickelte Modellrepräsentation überführt. Mittels dieser wurde das constraintbasierte Konzept auf den realen Modellen des Avalon-Systems angewendet. Es konnte gezeigt werden, dass sich konzeptuell sämtliche aufgetretenen Probleme, die das ursprüngliche eventbasierte Konzept bereitstellte, lösen lassen. Die neue, klarere Modellbeschreibung deckt sämtliche Fälle ab, die nötig sind, um ein solch komplexes System wie Avalon auf einer modellbasierten Ebene zu entwerfen. Das neu entworfene Konzept stellt somit eine solide Basis für zukünftige Forschungen im Bereich der Verhaltensplanung dar.

# 5 Diskussion und Ausblick

Es hat sich herausgestellt, dass die Herausforderungen der Informatik im untersuchten Teilbereich der Unterwasserrobotik keine ausgefallene Hardware benötigen. Die Forschung im Bereich der Autonomie kann vollständig mit einfachen Hardware-Plattformen unternommen werden. Die Informationen, die selbst aus einfachen Sensoren wie einem Scanning-Sonar gewonnen werden können, reichen für viele Anwendungen vollständig aus. Dabei kann, wie das Avalon-System gezeigt hat, sogar auf ein sonst so oft verwendetes DVL verzichtet werden. Die eingesetzten Komponenten, also die BMS/Thruster/Kameras, waren einfacher gehalten als im Vergleichssystem Dagon. Diese einfacheren Komponenten erfüllten jedoch sämtliche Anforderungen, um moderne Forschung auf dem Trägersystem Avalon ausführen zu können, und waren oftmals weniger fehleranfällig.

## 5.1 Komponentennetzwerkplanung

Es wurde durch den Einsatz strikter modellbasierter Konzepte sowie die Formalisierung der Anforderungen an Komponentennetzwerke und durch die Interpretation der Problemstellung als Graphproblem ein System erarbeitet, das auf der einen Seite ohne Simulation das Verhalten eines Komponentennetzwerkes auf Modellbasis analysieren lässt und auf der anderen Seite deutlich striktere und klarere Modellierungen erlaubt. Anstatt Probleme zur Laufzeit reaktiv zu lösen, wurde ein Konzept erarbeitet, wie ohne funktionelle Einschränkung eine vollständige Vorberechnung sämtlicher vom Roboter erreichbarer Zustände möglich ist.

Des Weiteren zeigte diese Arbeit, dass die Planung von Komponentennetzwerken viel dichter mit klassischen Weltzustandsplanungen verwoben ist, als es in bisherigen Arbeiten dargestellt wird. Die Sequenzierung von Aufgaben hängt stark vom Komponentenzustand ab. Ein Sensorausfall kann unter Umständen ein komplett anderes Verhalten hervorrufen, als es ohne diesen Ausfall möglich gewesen wäre. Auf der anderen Seite können durch redundante Sensoren/Algorithmen Aufgaben auch bei Fehlerfällen erreicht werden, die durch klassische Planungsansätze verworfen worden wären, da keine adaptive Planung bis auf die einzelnen Softwarekomponenten stattfindet.

## 5.2 Ausblick über zukünftige Arbeiten der Verhaltensplanung

Durch die Verbindung der Planung von Komponentennetzwerken und Weltzuständen ist eine neue Möglichkeit der Betrachtung von Planungsproblemen entstanden. Während klassische Ansätze sich auf die Sequenzierung zumeist virtueller atomarer Aktionen beschränken, zeigt diese Arbeit, dass diese Separierung unnötig ist.

Spätere Arbeiten können die Verhaltensweisen der StateMachines als *Service* (in Anlehnung zum *DataService*) verstehen. Eine StateMachine *NimmBecherAusSchank* kann somit beispielsweise den Service *HabeBecher* bereitstellen. Im *Failed-noMug* Fall könnte ein Plan-Manager automatisch (da es redundante StateMachines gibt, die den Service *Habe-Becher* bereitstellen) die StateMachine *WascheBecherAb* auswählen. Diese Art der Problemmodellierung definiert Teilverhaltensblöcke und Aktionen auf einem Weltzustand, der nicht Teil dieser Arbeit ist. Die Idee dahinter ist jedoch, dass die *DataServices* nicht darauf beschränkt sein müssen, Daten bereitzustellen.

Fortführende Arbeiten könnten die *DataServices* auch als Symbole interpretieren, die erfüllt werden müssen. Durch die bereits vorhandenen Kernregeln dieser Arbeit kann somit einfach eine Abbildung auf die besten *Services* erfolgen. Somit könnte die StateMachine *NimmBecherAusSchank* bevorzugt gegenüber *WascheBecherAb* selektiert werden. Hat ein robotisches System nur ein Arm, kann das Abwaschen nicht möglich sein. Dieses muss jedoch nicht explizit definiert werden, da dies durch die gestellten Constraints automatisch erkannt werden kann, weil die StateMachine *WascheBecherAb* nie ausgeführt werden kann. Somit wird der Robotik-Community ein System zur Verfügung gestellt, das stetig iterativ erweitert werden kann, ohne dass einzelne Teile für spezielle Systeme adaptiert werden müssen. Durch das Durchbrechen der Ebene der Komponentennetzwerke hin zu den komplexeren Weltaktionensplänen wird es möglich werden, automatische Abbildungen robust, ohne Overhead und ohne Speziallösungen vorzunehmen.

# A Referenzen


[1] Rachid Alami, Raja Chatila, Sara Fleury, Malik Ghallab, and Félix Ingrand. An architecture for autonomy. *The International Journal of Robotics Research*, 17(4):315–337, 1998.

[2] Jan Albiez, Sylvain Joyeux, and Marc Hildebrandt. Adaptive auv mission management in under-informed situations. In *OCEANS 2010*, pages 1–10. IEEE, 2010.

[3] Jan Christian Albiez. *Verhaltensnetzwerke zur adaptiven Steuerung biologisch motivierter Laufmaschinen*. GCA-Verlag, 2007.

[4] Charles André. Syntax and semantics of the clock constraint specification language (ccsl). 2009.

[5] Gianluca Antonelli and G Antonelli. *Underwater robots*. Springer, 2014.

[6] Ronald C Arkin. *Behavior-based robotics*. MIT press, 1998.

[7] PG Balakrishnan, R Ramesh, and T Prem Kumar. Safety mechanisms in lithium-ion batteries. *Journal of Power Sources*, 155(2):401–414, 2006.

[8] Dana H Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern recognition*, 13(2):111–122, 1981.

[9] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, 2(1):14–23, Mar 1986.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[11] The Rock Developers. Processing data - the stream aligner. `http://rock-robotics.org/stable/documentation/data_processing/stream_aligner.html`. Accessed: 15.07.2016.

[12] The Rock Developers. Rock, the Robot Construction Kit. `http://rock-robotics.org/`, 2012. http://www.rock-robotics.org.

[13] Stefan Edelkamp and Jörg Hoffmann. Pddl2. 2: The language for the classical part of the 4th international planning competition. *4th International Planning Competition (IPC'04)*, 2004.

[14] M. Einhorn, W. Roessler, and J. Fleig. Improved performance of serially connected li-ion batteries with active cell balancing in electric vehicles. *Vehicular Technology, IEEE Transactions on*, 60(6):2448–2457, July 2011.

[15] Austin Eliazar and Ronald Parr. Dp-slam: Fast, robust simultaneous localization and mapping without predetermined landmarks. In *IJCAI*, volume 3, pages 1135–1142, 2003.

[16] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[17] T Fossen. *Guidance and Control of Ocean Vehicles.* John Wiley and Sons, 1994.

[18] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res.(JAIR)*, 20:61–124, 2003.

[19] Ángel García-Olaya, Sergio Jiménez, and Carlos Linares López. The 2011 international planning competition. 2011.

[20] Michael R Genesereth and Nils J Nilsson. Logical foundations of artificial. *Intelligence. Morgan Kaufmann*, 1987.

[21] Alfonso Gerevini and Derek Long. Plan constraints and preferences in pddl3. *The Language of the Fifth International Planning Competition. Tech. Rep. Technical Report, Department of Electronics for Automation, University of Brescia, Italy*, 75, 2005.

[22] Alfonso Gerevini and Derek Long. Preferences and soft constraints in pddl3. In *ICAPS workshop on planning with preferences and soft constraints*, pages 46–53, 2006.

[23] Matthias Goldhoorn. Unterwasser slam (underwater slam). diploma thesis, University of Bremen, Faculty of Computer Science, 2010.

[24] Matthias Goldhoorn and Sylvain Joyeux. Extension of a plan-based component manager for real time adaptation. In *ISR/Robotik 2014; 41st International Symposium on Robotics; Proceedings of*, pages 1–6. VDE, 2014.

[25] Ronny Hartanto. *A hybrid deliberative layer for robotic agents: fusing DL reasoning with HTN planning in autonomous robots.* Springer-Verlag, 2011.

[26] M Helmert. Changes in pddl 3.1. *Unpublished summary from the IPC-2008 website*, 2008.

[27] Javier Hidalgo Carrio, Sascha Arnold, and Pantelis Poulakis. On the design of Attitude Heading Reference Systems using the Allan variance. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, PP(99):1, jan 2016.

[28] J. Hidalgo-Carrió, S. Arnold, and P. Poulakis. On the design of attitude-heading reference systems using the allan variance. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 63(4):656–665, April 2016.

[29] Marc Hildebrandt. *Development, Evaluation and Validation of a Stereo Camera Underwater SLAM Algorithm.* PhD thesis, Bremen, Universität Bremen, Diss., 2014, 2014.

[30] Marc Hildebrandt and Jens Hilljegerdes. Design of a versatile auv for high precision visual mapping and algorithm evaluation. In *2010 IEEE/OES Autonomous Underwater Vehicles*, pages 1–6. IEEE, 2010.

[31] Paul VC Hough. Method and means for recognizing complex patterns, December 18 1962. US Patent 3,069,654.

[32] Okhtay Ilghami and Dana S Nau. A general approach to synthesize problem-specific planners. Technical report, DTIC Document, 2003.

[33] Sylvain Joyeux. *A software framework for plan management and execution in robotics: application to multirobot systems.* PhD thesis, Ph. D. Thesis, ISAE. http://tel. archivesouvertes. fr/tel-00283086/fr, 2007.

[34] Sylvain Joyeux, Rachid Alami, Simon Lacroix, and Roland Philippsen. A plan manager for multi-robot systems. *The International Journal of Robotics Research*, 28(2):220–240, 2009.

[35] Sylvain Joyeux, Jan Albiez, et al. Robot development: from components to systems. In *6th National Conference on Control Architectures of Robots*, 2011.

[36] Sylvain Joyeux, Frank Kirchner, and Simon Lacroix. Managing plans: Integrating deliberation and reactive execution schemes. *Robotics and Autonomous Systems*, 58(9):1057–1066, 2010.

[37] Balachander Krishnamurthy and David S Rosenblum. Yeast: A general purpose event-action system. *IEEE transactions on Software Engineering*, 21(10):845–857, 1995.

[38] Barbara La Scala and Mark Morelande. An analysis of the single sensor bearings-only tracking problem. In *Information Fusion, 2008 11th International Conference on*, pages 1–6. IEEE, 2008.

[39] Mikael Z. Lagerkvist. *Techniques for Efficient Constraint Propagation.* phdthesis, Royal Institute of Technology, 2008.

[40] Wai Chung Lee, D. Drury, and P. Mellor. Comparison of passive cell balancing and active cell balancing for automotive batteries. In *Vehicle Power and Propulsion Conference (VPPC), 2011 IEEE*, pages 1–7, Sept 2011.

[41] Jun S Liu. Metropolized independent sampling with comparisons to rejection sampling and importance sampling. *Statistics and Computing*, 6(2):113–119, 1996.

[42] Ingo Lütkebohle, Roland Philippsen, Vijay Pradeep, Eitan Marder-Eppstein, and Sven Wachsmuth. Generic middleware support for coordinating robot software components: The task-state-pattern. *Journal of Software Engineering for Robotics*, 2(1):20–39, 2011.

[43] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language. 1998.

[44] Conor McGann, Frederic Py, Kanna Rajan, Hans Thomas, Richard Henthorn, and Rob McEwen. T-rex: A model-based architecture for auv control. In *3rd Workshop on Planning and Plan Execution for Real-World Systems*, volume 2007, 2007.

[45] Conor McGann, Frederic Py, Kanna Rajan, Hans Thomas, Richard Henthorn, and Rob McEwen. A deliberative architecture for auv control. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 1049–1054. IEEE, 2008.

[46] Sheila A McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. *IEEE intelligent systems*, (2):46–53, 2001.

[47] Misc. Iec 60449 - voltage bands for electrical installations of buildings. Technical report, International Electrotechnical Commission (IEC), 1973.

[48] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 968–973. Morgan Kaufmann Publishers Inc., 1999.

[49] Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. Shop2: An htn planning system. *J. Artif. Intell. Res.(JAIR)*, 20:379–404, 2003.

[50] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[51] Gregg Rabideau, Russell Knight, Steve Chien, Alex Fukunaga, and Anita Govindjee. Iterative repair planning for spacecraft operations using the aspen system. In *Artificial Intelligence, Robotics and Automation in Space*, volume 440, page 99, 1999.

[52] David Ribas, Pere Ridao, and José Neira. *Underwater SLAM for structured environments using an imaging sonar*, volume 65. Springer, 2010.

[53] Silvia Richter and Matthias Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Int. Res.*, 39(1):127–177, September 2010.

[54] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[55] Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.

[56] Christian Schulte and Guido Tack. View-based propagator derivation. *Constraints*, 18(1):75–107, 2013.

[57] Steven W Smith et al. The scientist and engineer's guide to digital signal processing. 1997.

[58] Peter Soetens. *A Software Framework for Real-Time and Distributed Robot and Machine Control*. PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2006. `http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf`.

[59] Cyrill Stachniss. *Robotic mapping and exploration*, volume 55. Springer, 2009.

[60] W Stegmüller. Martin davis:" computability and unsolvability". 1962.

[61] Guido Tack, Christian Schulte, and Gert Smolka. Generating propagators for finite set constraints. In Frédéric Benhamou, editor, *Twelfth International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 575–589, Nantes, France, September 2006. Springer-Verlag.

[62] J-M Tarascon and Michel Armand. Issues and challenges facing rechargeable lithium batteries. *Nature*, 414(6861):359–367, 2001.

[63] Moritz Tenorth and Michael Beetz. KnowRob – A Knowledge Processing Infrastructure for Cognition-enabled Robots. Part 1: The KnowRob System. *International Journal of Robotics Research (IJRR)*, 32(5):566 – 590, April 2013.

[64] Moritz Tenorth and Michael Beetz. Knowrob: A knowledge processing infrastructure for cognition-enabled robots. *The International Journal of Robotics Research*, 32(5):566–590, 2013.

[65] Moritz Tenorth, Alexander Clifford Perzylo, Reinhard Lafrenz, and Michael Beetz. The roboearth language: Representing and exchanging knowledge about actions, objects, and environments. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 1284–1289. IEEE, 2012.

[66] S Thrun, W Burgard, and D Fox. *Probabilistic Robotics*, volume 45. MIT Press, 2005.

[67] Sebastian Thrun. Probabilistic robotics. *Communications of the ACM*, 45(3):52–57, 2002.

[68] Markus Waibel, Michael Beetz, Javier Civera, Raffaello d'Andrea, Jos Elfring, Dorian Galvez-Lopez, Kai Haussermann, Rob Janssen, JMM Montiel, Alexander Perzylo, et al. A world wide web for robots. *IEEE Robotics & Automation Magazine*, 18(2):69–82, 2011.

[69] Pierre Willenbrock. Zeitsynchronisierung verschiedener zeitquellen in mobilen robotik experimenten. Master's thesis, Universität Bremen, 2012.

# B Unpublished Journal Article

Die vorliegende Arbeit baut auf einigen gemeinsam mit verschiedenen anderen Wissenschaftern erarbeiteten Ergebnissen auf, die bisher nicht publiziert wurden, für die es aber ein unpubliziertes Manuskript gibt. Ich drucke deswegen hier dieses Manuskript mit Genehmigung der Autoren ab, um aus der eigentlichen Dissertation auf diese gemeinsamen Arbeiten verweisen zu können.

**Noname manuscript No.**
(will be inserted by the editor)

# Avalon - A Reliable Autonomous Underwater Vehicle for Marine Missions

**Matthias Goldhoorn****, Sascha Arnold***, Christian Clausen**, Tim Lehr**, Niklas Pech**, Fabio Zachert**, Malgorzata Goldhoorn**** and Frank Kircher***

Received: date / Accepted: date

**Abstract** Nowadays, autonomous robots like autonomous underwater vehicles (AUVs), that are able to perform tasks over days and months, are becoming increasingly important in the maritime robotics. Such systems could be used for tasks like deep water oilfield inspection or to explore the ocean's landscape. Nevertheless, such an AUV must be able to perform the tasks robustly and be equipped with capabilities like autonomous long-term navigation or dynamic planning under uncertainty. In this paper our AUV *A*utonomous *V*ehicle for *A*quatic *L*earning, *O*peration and *N*avigation (Avalon) as well as the underlying system architecture are presented. We give the overall system description including new algorithms, which we have developed over the last seven years using the robot.

**Keywords** long term-autonomy, underwater SLAM, AUVs, system modelling, system architecture, robots middleware

## 1 Introduction

The system's ability to perform certain tasks autonomously over long periods of time is of great importance in most robotics research areas. Such an intelligent system will also be of great use in the field of marine robotics. AUVs could, for example, perform tasks associated with marine rescue scenarios or offshore industrial missions. The main advantage is that the vehicle can operate in great depths, for long time. Our vision is that AUVs operates for months autonomously without any supporting surface vehicle like motherships. This could reduce the costs of inspection. However, there are still open questions related to the development of such robust autonomous underwater systems for long-term real world operations. These especially include the software design. Regarding the long-term autonomy there are two main aspects: The system must be able to detect significant negative events, and then reason how to react to these given faults. In this paper we present our AUV *Avalon* Figure 1 and the associated project with the same name. The Avalon project was started in 2007 as a students project of the University of Bremen in a cooperation with the DFKI-RIC research institute. Within this project different algorithms and two robotic systems were developed. During the project we have performed several tests in our testing facility and in real-world environments such as the lake near our University. Furthermore, we have presented our skills at the European contests for underwater robotics: Student Autonomous Underwater Vehicle Challenge-Europe (SAUC-E) and Eurathlon. In both of these competitions we have shown our technological capabilities.

The remainder of this paper is structured as follows: First, in section 2 we give recommendations for hardware design and explain those based on our system *Avalon*. This hardware related section is followed by the description of our software design, which is given in section 3. In this section we present our design for the modelling and point out its important parts. In the last part of this section our design for building autonomy is

* Head of the German Research Center for Artificial Intelligence - Robotics Innovation Center (DFKI-RIC) and head of the Robotics Group at the University of Bremen
** Student of University of Bremen
*** Member of the DFKI-RIC
**** PhD Student of Graduate School System Design of the University of Bremen
Adress: Robert-Hooke-Str. 1, D-28359 Bremen, Germany
Phone: +49(421)17845-4100
Fax: +49(421)17845-4150
Email: matthias.goldhoorn@uni-bremen.de

explained in section 3.4. After the software modelling we present our approach for model building of the vehicle motion in section 4.1 and sonar data interpretation in section 4.2. The next section 5 describes the wall detection algorithm which can be used for structure-inspection tasks. This is followed by two localization methods, one based on particles in section 6 and the second which supports the first one based on images in section 7. In the section 8 we explain our control chain which we use for the vehicle motion control. Finally, the evaluation in section 9 and outlook of our future work in section 11 are presented.

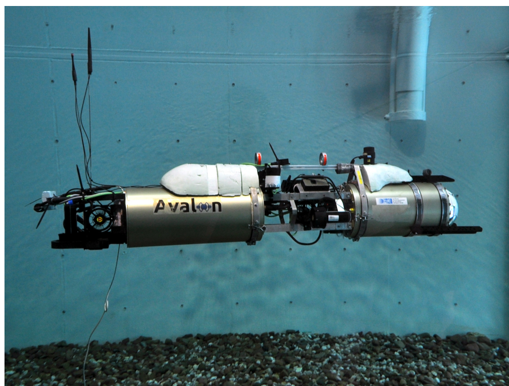



**Fig. 1** Avalon within a test chamber

## 2 Hardware Design

The underlying hardware is one very important part of the overall system. This section outlines the relevant parts of the Avalon AUV and explains the general concept that we propose where a system should follow to become more robust and capable to fulfill long-term missions. From our experience, the most important challenge in hardware designs for vehicles that shall operate long time autonomously is not the question of how to make the hardware as robust as possible (even though this is important, too). The core problems often are to detect and announce failures on the hardware and lower software level and to take the right decisions regarding these problems. It is, in fact, impossible to build a hardware system that will not fail. Therefore, the failures on the lowest level have to be recognized to make it possible to find a proper reaction. One example for such a failure could be a thruster which is blocked due to litter caught in its screw.

### 2.1 Avalon

Avalon was intended to be a modular extendable multi-sensor general-purpose AUV for research applications with the focus on real world domains. It consists of two pressure hulls. Each hull includes batteries and a standard x86 based platform for data processing. The connection between the compartments was made using SubConn underwater connectors. SeaBotix BTD-150 thrusters are used for propulsion. One of the most important sensors for our algorithms is the Tritech Micron Scanning sonar, which is mandatory for our localization method (described in section 6). We were able to navigate the vehicle precisely in different areas, even though Avalon does not have a DVL.

#### *2.1.1 Propulsion*

One of the most common thrusters for small research AUVs from our experience are the Seabotix BTD-150 thrusters. These thrusters are DC-controlled. Unfortunately, the mostly used control variant is a simple RC-craft pulse width modulation (PWM) generator to control the motor. The drawback is that no feedback information is available. At this layer we decided to use an in-house developed H-Bridge PWM controlled electronic board, which also supports current monitoring. The current monitoring gives a sufficient information about the function of the thruster. If the current is too low for the given PWM, the thruster might be completely broken or the connection might be lost. On the other hand currents that are too height indicate a somehow blocked motor. Both error states are indicated and reported to higher software layers.

#### *2.1.2 Energy System*

One very important point for autonomous operation are the batteries, that provide the energy to operate the vehicle. Current lithium cells have one of the best energy available to weight ratios [Tarascon and Armand, 2001]. However, chained cells, cause a disbalance over time and therefore need additional electronics to rebalance the cells. With the intention of autonomous docking and long-time operation, the cells need to be balanced automatically. The problem with most of the currently used battery management systems is that too high voltages of cells are converted to heat energy, by cutting short the cells with resistors [Lee et al., 2011]. It is difficult to transfer heat in the water due to a need of additional seals which would increase the risk of leakage.

Moreover, the balancing procedure wastes energy. Therefore, we found another way using an active charg-

ing and balancing system which also includes the monitoring of the batteries. In our vehicle we are using eight LiFeyPo4 cells with a capacity of 40Ah each.

Our active balancing solution, described in section [Einhorn et al., 2011], is much more efficient. Only 10% (excluded internal lost of the battery itself) of energy is lost during the re-balancing comparing to 100% loss of energy during the passive balancing solution.

### *2.1.3 Security System*

One main challenge in the operation of AUV's is the ability to gain and keep control over the vehicle all the time during a long-term mission. Despite these challenges, the system might operate for long times autonomously, the operator must be able to cancel and recover the vehicle in any condition, even if the complete software stack, or -far worse-, the controlling PC fails. The more the important modules or components fail, the more limited the emergency behavior gets. In some cases the behavior must be a surfacing behavior where the thrusters give a constant force downwards.

The fallback control is mostly used for emergency surfacing commands to abort the current mission. Another advantage of our design improves the usability when in tethered operation mode. In this situation the operator can get a ROV like mode which is independent from the PC's. This is required to abort immediately the high-level PC control, to re-gain the manual control and reconfigure the PC during daily research usage. To allow that, we have found a solution to controlling our motors in a special way. Figure 2 shows the principle. Our motors are connected through a CAN-bus to high-level PC system. On the same bus a micro controller is attached, which is able to takeover the control in special circumstances.

During autonomous routine operation, the controller sends a message to the motor-driver-controller, which enables the listing of lower-prioritized CAN messages. These CAN messages are the only ones send by the PC. The PC, hence, cannot control the motors if the system-supervisor board has not enabled it.

In parallel, we have a second Wireless-LAN serial communication link on the 867MHZ band and this communication does not need handshaking. We implemented a point to multipoint communication on top on this device. We are able to control the motors directly over our system-supervision which disables the control of the PC and sends higher prioritized messages to the motors.

Due to the design of a CAN-bus, these higher prioritized messages are reliable regarding transport. Even if the PC starts to flood the bus with its low-prioritized messages, the supervisor is able to gain control. The same situation occurs if the supervisor receives the emergency-surfacing command via the modem, which is connected directly to it.

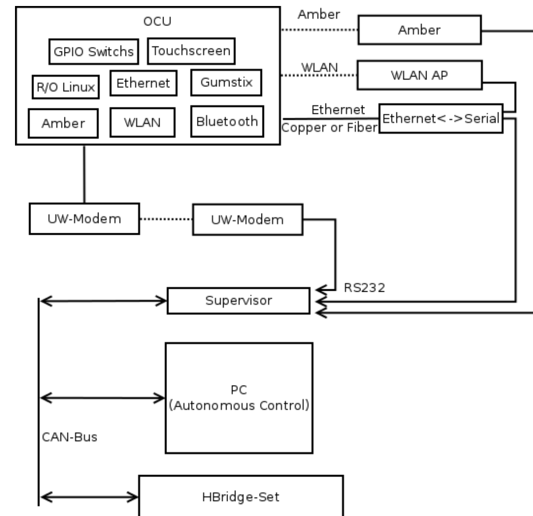



**Fig. 2** The communication bus for our system supervisor board

## 3 Software Design

This section presents the core components of our software design, including the used communication framework and mission modelling tools. First, we give an overview of the communication framework. This is followed by rules that have to be applied to the components and the description of the components themselves. The section ends with the overview of the mission management capabilities.

### 3.1 Overview

The selection of a suitable middle ware software is an important part in modern software development. The framework itself has to be flexible in its skills on one hand, yet stiff in the interface design on the other. The more unclear or unspecified the interfaces of software components become, the more complex the reusability of such components gets. For example, the C++ language allows to define clear and flexible interfaces, but re-using them requires the integration into the own software on C++ level. The interface should also support a mechanism for configuration and control of the components. Another important parts are status information which respond to component's both normal behavior and bahavior witnessed in case of incidents.

This last point is crucial for building autonomous systems. The components should report their *state* in a generic way. The internal healthy state of the component influence the system's stability or the mission.

The framework ROS [Quigley et al., 2009] supports a clear interface for communication, but misses a clear interface for controlling the component and for event status informations. For that reason, we decided to focus on the Robot Construction Kit (ROCK) [roc, 2012], which has a more fine-detailed way of reporting status informations and controlling components.

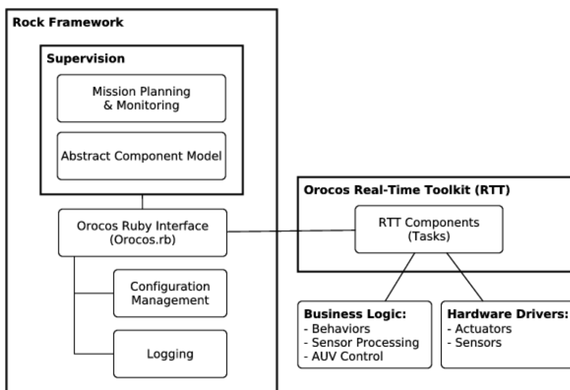



**Fig. 3** Software design of our communication framework

ROCK is similar to ROS, a software framework for the development of robotic systems, based on the Orocos RTT (Real Time Toolkit) in which RTT components (*tasks*), hardware drivers and business logic algorithms (e.g. sensor processing, localization) are written in standard C++. Each task can be configured for several intentions, but the core idea is based on the policy that each task fulfills only one specific purpose. A particular *task* can, therefore, be connected in different ways. One example is a task for image processing, which can either be connected directly to a camera, or to some pre-processing *tasks*. The configuration of the tasks depend on the conditions under which the system should operate. In ROCK all of these changes can be done from an outer interface, without changing the component itself. Figure 3 illustrates the software design principle.

Another advantage of strong modularization is the increasing capability of reusability of such components. By following these principles, a lot of components become automatically independent from the underlying system since the source of the data can stay unknown for the component developer. From our experience we could observe that the amount of time for new workers to understand the system is, thus, reduced. Each new developer has to understand his/her specific part only. Over the time the known range expands in a natural way, iteratively.

At this point, we would like to introduce the role of the *SystemDesigner*, who takes care of the integration of all components in the system. The *SystemDesigner* could be compared to a conductor of an orchestra. He/She defines roles and builds up abstract models for the underlying atomic algorithms that are encapsulated in *tasks*. In the end, all of these running tasks and the way they are connected and configured form the behavior of the system for one point in time (see Fig. 4)

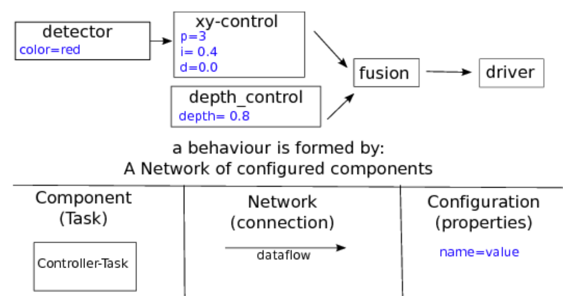



**Fig. 4** Definition of system behavior for one point in time

### 3.2 Component Design

Following the previously described principles, we created some recommendations how software components should be developed to build a maintainable system. These require clear and detailed definitions of interfaces to allow a simple integration into the complete network of components.

#### *Drivers*

A driver is the lowest access point for the system to the hardware. Therefore, any failure has to be reported by the driver. For instance if a sensor produces data periodically, the driver has to report a failure if data gets missing. The same occurs in the case of a connection port being invalid or a device not being able to be configured as expected. All errors that are reported by the device drivers are announced to higher layers through the state system explained in section 3.3. Another problem that most developers do not take into account is the timestamp problem. Using the current system time, when a data sample arrives in the system, timestamp might not be sufficient. The scheduling in modern systems introduces jitter, hence the sensor fusion might get inaccurate values. Therefore, an algo-

rithm was created inside of ROCK which estimates the correct creation time of these samples [roc, 2012].

### 3.3 System Modeling

To realize autonomy, several points have to be taken into account. Each part from the hardware devices to each software component must cooperate together correctly. This plays an important role in the current software development in robotics, where the amount of algorithms is continuously increasing. An unified component design, as mentioned earlier, is important to keep the system maintainable.

With the *SystemDesigner* the ROCK framework has a tool to support the development on top of the component design, by an abstract layer that allows model-checking. This layer verifies not only the correct matching of the data types, it also verifies that the semantics of the data flow matches.

The developer of the component can define abstract services as described in publication [Joyeux et al., 2011]. We are using the realization called *Syskit* for our system.

*Syskit* is the software that our *SystemDesigner* uses to design the overall component network. It allows the *SystemDesigner* to build a hierarchical and capsuled network. All requirements that are needed to run a system are defined as *InstanceRequirements* (IR) and these IR are resolved to one or multiple tasks. One IR, for example, could be a sensor fusion component as shown in Fig. 5. In the end, the *DataService IMUSensors* is resolved to an actual provider of *IMUSensors*, like a real IMU, such as the XSens::MTi. This modeling principle stands out from the real devices and makes the system design independent from the underlying hardware.

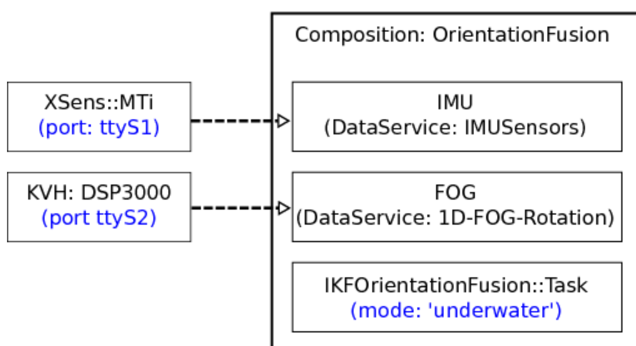



**Fig. 5** Abstract composition of sensor fusion with associated devices

At one point of the design, the association between DataServices and devices *could* be made, but is by no means a necessity. As long as there is only one provider of this information available, there is no need to define this and the resolution between the services could be done automatically. Furthermore, the association between DataServices and its providers is not limited only to the devices.

As mentioned, an IR could be a real task that runs on a system, but it could also be a *composition* which is a collection of IRs. These *compositions* encapsulate several IRs-like tasks or DataServices and provide by themselves DataServices that are used in other compositions in the end. This points out that all kind of representations are automatly IRs. This kind of modeling enables a quite flexible and portable way of algorithms and enables the direct reusability and integration into other ones.

### 3.4 Autonomy

To achieve autonomy, additional information is required. The system needs to change its behavior. This could be done by changing the internal state of components as explained in [Lütkebohle et al., 2011]. The drawback here is, that in the end the system control depends on the components. Each component makes decisions how the system shall behave. This spreads the decision making into the components and can result in undesirable states if a component does not behave as expected or the interaction between the components ends up in unwanted behavior. We therefore choose an approach as explained in [Joyeux et al., 2009] and [Goldhoorn and Joyeux, 2014]:

The idea that each component reports only its own state defines a good basis for interaction. Each task reports its state, this could be in a format of an error from a device or an information stating that a controlling behavior ended successfully. The complete system mission design is based on such state changes. These changes are monitored by the *PlanManager*. The *PlanManager* we are using is *Roby* [Joyeux et al., 2009]. The *PlanManager* executes the plan which was designed by the *SystemDesigner*. Depending on the progress of the mission, the *SystemManager* has to react properly in order to achieve the overall mission goal: to succeed in as many missions as possible while operating safely and not exceeding the time limit. The *SystemManager* has the capability to abort missions if they do not progress as desired, for example, if the pipeline following task sends the signal that it has lost the pipeline. In that case, the *SystemManager* passes the vehicle control over to the navigation task in order to drive to the last known pipeline position. Then, if there is enough time left, the pipeline following behaviour can start over.

As an example of an autonomous mission, we would like to present a practice mission we created for demonstration purposes as illustrated in Figure 6. The goal is to find the pipeline, survey it (Fig: 7), navigating to a given coordinate, following the wall for two corners and stop in the third one. This is done in an inifinity loop. This mission worked for more than ten iterations for over two hours without any human interaction, even sometimes the system got out of its requested task. See Figure 8 for an overview of the complete mission.

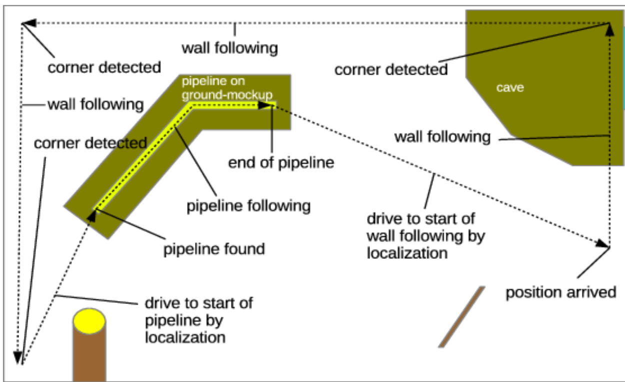



**Fig. 6** Long-time autonomous demonstration run in the maritime test hall at DFKI Bremen ( 23m x 19m x 8.0m )

**Fig. 7** Avalon is inspecting the pipeline in the maritime test hall at DFKI Bremen ( 23m x 19m x 8.0m ) (picture: Annemarie Hirth / DFKI GmbH)

## 4 Building models of Sensors and the System

### 4.1 Underwater Velocity Estimation

We use a dynamic motion model for predicting the vehicle movements in an abstract representation. This

```
state_machine "demo"
    starting_state = state submerge
    search_pipe = state find_pipe_with_localization
    follow_pipeline = state pipeline
    rescue = state move_to_pipe_end
    window = state to_window
    wall = state wall_servoing_right

    start(starting_state)
    transition(starting_state.success,search_state)
    ...
    transition(follow_pipeline.success,window)
    transition(follow_pipeline.lost_pipe,rescue)
    ...
end
```

**Fig. 8** Simplified design of our autonomous demo mission

model allows to estimate the current velocity and direction of Avalon from the actuator commands of our thruster controllers. Formally, the model simulates state transitions described by the common notation for motion of a maritime vehicle:

$$\mathbf{v} = \left[v_x \; v_y \; v_z \; v_\Phi \; v_\theta \; v_\psi\right]^T \tag{1}$$

For the Avalon system the linear and angular velocities are represented by the body-fixed frame illustrated in Figure 9. The rotation around the $x, y, z$ axis is defined as $\Phi, \theta, \psi$, or roll, pitch and yaw.

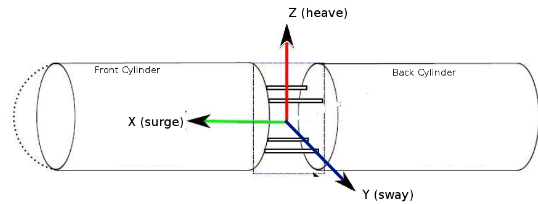



**Fig. 9** Sketch of the underwater vehicle Avalon

The basic mathematical theory behind our dynamic underwater model is formed by the general differential equation introduced by [Antonelli and Antonelli, 2014]:

$$M_v\dot{v} + C_v(v)v + D_{RB}(v)v + g_{RB}(\Theta) = \tau_v \tag{2}$$

$M_v$ being the mass matrix, $C_v$ the coriolis and centripetal matrix, $D_{RB}$ being the damping matrix, $g_{RB}$ being the gravity and buoyancy matrix and $\tau$ being the force and torque matrix representing the thrusters.

The mass matrix $M_v$ is given by:

$$M_v = \begin{bmatrix} m \cdot I_3 & 0 \\ 0 & I_O \end{bmatrix} + M_A \tag{3}$$

where $I_3 \in \mathbb{R}^{3\times3}$ is the identity matrix and $I_0$ specifies the inertia tensor expressed in the body-fixed frame. For

Avalon the inertia tensor matrix is basically described by the moments of inertia for three cylinders with a radius $r_i$, length $l_i$, mass $m_i$ and a distance vector $c_i = \begin{pmatrix} x_c & y_c & z_c \end{pmatrix}$ to the centre point of mass given by:

$$I_O = I_{c1} + I_{c2} + I_{c3} \tag{4}$$

with

$$I_{ci} = I_{mi} + I_{ri} \tag{5}$$

and

$$I_{mi} = \begin{bmatrix} \frac{1}{2}m_i r_i^2 & 0 & 0 \\ 0 & \frac{1}{12}(m_i(3r_i^2 + l_i^2)) & 0 \\ 0 & 0 & \frac{1}{12}(m_i(3r_i^2 + l_i^2)) \end{bmatrix} \tag{6}$$

$$I_{ri} = \begin{bmatrix} m_i(y_c^2 + z_c^2) & 0 & 0 \\ 0 & m_i(x_c^2 + z_c^2) & 0 \\ 0 & 0 & m_i(x_c^2 + z_c^2) \end{bmatrix} \tag{7}$$

The added mass matrix considers the surrounding fluid to the vehicle and its acceleration. It is specified for Avalon by:

$$M_{Ai} = \begin{bmatrix} -0.1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\pi\rho r_i^2 l_i^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\pi\rho r_i^2 l_i^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{12}\pi r_i^2 l_i^3 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{12}\pi r_i^2 l_i^3 \end{bmatrix} \tag{8}$$

We are able to measure the thruster force for a given voltage. For our model we approximate a linear relation between force and voltage, which allows us to estimate the current force of the thrusters at runtime.

With this done, we can identify the mass matrix and the damping matrix experimentally with a simplifying assumption $v \longrightarrow 0 \Longrightarrow C_v(v) = 0$ due to the relatively low speed of maximum $1/\frac{m}{s}$ of our underwater vehicle. Additionally, the gravity and buoyancy matrix can also be expected to be very small and to affect mostly the depth-control of Avalon. We have sensors to get much more accurate information about the current depth and the current velocity along the z-axis (heave) is not needed for any task we want to solve. Because of these aspects we also assume $g_{RB} = 0$.

The dynamic model could estimate the current speed in all three directions and all rotational axes. But we have sufficient sensors to get samples for depth and yaw. Avalon cannot control its roll, and optimally its pitch is zero. Hence, the model is basically used to estimate the speed on the x- (surge) and the y-axis (sway). This model is especially interesting for our localization algorithm as an implementation of the dynamic state transition in a temporal filter.

So, in the end, we can use the following, simplified equation, where all forces are set to 0, except of the forces in x- and y-direction.

$$M_v\dot{v} + D_{RB}(v)v = \tau_v \tag{9}$$

## 4.2 Sonar Processing

Avalon is equipped with an 360° active scanning sonar. The sonar sends out a signal at a certain angle and records all returning signals at the expected frequency for a limited period of time. This signal is called *sonar beam* and is affected by natural and artificial noise, ambiguities and double reflections. The main reason for the ambiguities is the fact that scanning sonars are made for obstacle avoidance and therefore have a high vertical opening angle (e.g. 35° in case of the Tritech Micron sonar). The goal for the sonar processing is to filter the sonar signal and extract the actual obstacles that can be processed by higher software components, like the particle localization (6) or the wall detection (5).

The sonar feature extraction locates for each sonar beam a set of possible candidates. Every candidate is supposed to be an obstacle, e.g. a wall. All candidates are weighted by applying a measurement for their probability.

### *4.2.1 Candidate extraction from a sonar signal*

For extracting the candidates from a sonar signal, the signal is filtered and then the derivative of the signal is used to identify and weight the candidates. To filter the discrete signal $\mathbf{S_k}(i)$ a moving average filter [Smith et al., 1997] is used to smooth the signal:

$$\bar{\mathbf{S}}_\mathbf{k}(i) = \frac{1}{m} \sum_{j=-(m-1)/2}^{(m-1)/2} (\mathbf{S_k}(i+j)) \tag{10}$$

$k$ identifies the current sonar beam with a bearing $angle(k)$. $m$ describes the window size of the filter. Smoothing the signal allows to compute a more distinguishable derivative later on.

Afterwards we remove the noise caused by known sources. This is for instance the noise from the sonar device itself and reflections from the ground and the water surface. We define a model for each source represented by a function. We discovered that an exponential decay function ($e^{-x}$) is suitable to represent the sonar device noise $M_D$. The reflections from the ground $M_B$

and the surface $M_S$ are represented by Gaussian distributions. Here we make the assumption of having flat ground and surface planes. We subtract all models from the signal $\bar{\mathbf{S}}_k$ in order to compute the signal $\bar{\bar{\mathbf{S}}}_\mathbf{k}$:

$$\bar{\bar{\mathbf{S}}}_\mathbf{k} = \bar{\mathbf{S}}_\mathbf{k} - (a_D \cdot M_D + a_B \cdot M_B + a_S \cdot M_S) \tag{11}$$

$0 \leq a_{D,B,S} \leq 1$ are factors to weight the impact of each model to the signal. The model offsets for the two Gaussian distributions can be computed by the current orientation of the AUV, the distance to the ground (sensed by an echo sounder) and the distance to the surface (sensed by a pressure sensor). The scaling of each model is updated by the local minimum and maximum values of the signal around the model offset. The example in

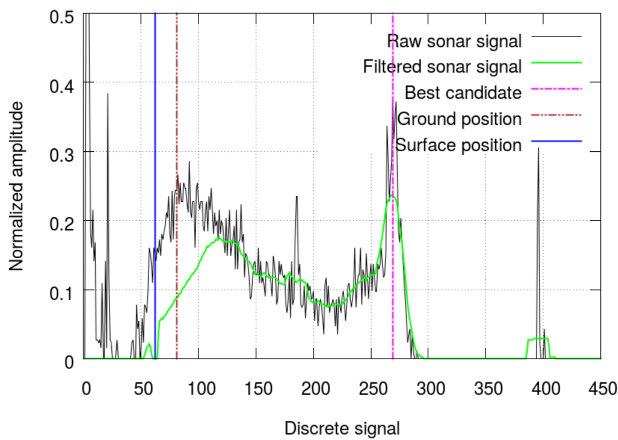



**Fig. 10** Thin black line: The raw sonar signal $\mathbf{S_k}$. Thick green line: The filtered signal $\bar{\bar{\mathbf{S}}}_\mathbf{k}$. Solid blue vertical line: Estimated first contact with the surface. Double dotted brown vertical line: Estimated first contact with the ground. Single dotted purple line: Estimated position of a wall (best candidate)

Figure 10 shows the impact of the models in the filtered signal $\bar{\bar{\mathbf{S}}}_\mathbf{k}$. The noise from the sonar device is completely subtracted by the exponential decay function. The first contact with the ground is correctly estimated and the impact of the subtraction of the corresponding Gaussian distribution is observable in the resulting filtered signal $\bar{\bar{\mathbf{S}}}_\mathbf{k}$. In this example, the surface doesn't have a distinguishable echo. Reflections from the ground depend highly on its consistency, its structure and any existing objects. Reflection from the surface depends highly on the shape of waves. Our experience has shown that especially in open sea environments the predictions for the ground and the surface are mostly too inaccurate in the situation of distinguishable reflections from both directions. Therefore, we are usually not using $M_B$ and $M_S$ in the current state.

In the next step we compute the discrete derivative $\mathbf{S}'_\mathbf{k}(i)$ of the filtered signal $\bar{\bar{\mathbf{S}}}_\mathbf{k}$, which is used in the following to identify and rate the local maxima in the signal:

$$\mathbf{S}'_\mathbf{k}(i) = \frac{\mathrm{d}}{\mathrm{d}i}\bar{\bar{\mathbf{S}}}_\mathbf{k}(i) = \frac{\bar{\bar{\mathbf{S}}}_\mathbf{k}(i+1) - \bar{\bar{\mathbf{S}}}_\mathbf{k}(i-1)}{2} \tag{12}$$

The candidate offsets in the current signal are represented by a set $P_k$, which contains the indices of all local maxima in the signal $\bar{\bar{\mathbf{S}}}_\mathbf{k}$. Those local maxima can be extracted from the derivative $\mathbf{S}'_\mathbf{k}$:

$$P_k = \left\{0 \leq i \leq N \mid \mathbf{S}'_\mathbf{k}(i) \geq 0 \wedge \mathbf{S}'_\mathbf{k}(i+1) < 0\right\} \tag{13}$$

In which $N$ is the number of samples in the discrete signal. For all elements of the set $P_k$ we compute the average of the positive and absolute negative slope responsible for the current local maximum (Fig. 11) using the function $M : P \to \mathbb{R}$.

$$\begin{aligned} \forall n \in P_k : \; & m \leq n, \mathbf{S}'_\mathbf{k}(m) \geq 0 \wedge \mathbf{S}'_\mathbf{k}(m-1) < 0, \\ & n < o, \mathbf{S}'_\mathbf{k}(o) < 0 \wedge \mathbf{S}'_\mathbf{k}(o+1) \geq 0, \\ M(n) &= \frac{1}{o-m}\left(\sum_{i=m}^{n} \mathbf{S}'_\mathbf{k}(i) - \sum_{i=n+1}^{o} \mathbf{S}'_\mathbf{k}(i)\right) \end{aligned} \tag{14}$$

$M_k$ is the set of all slopes from $M(n)$ for the current

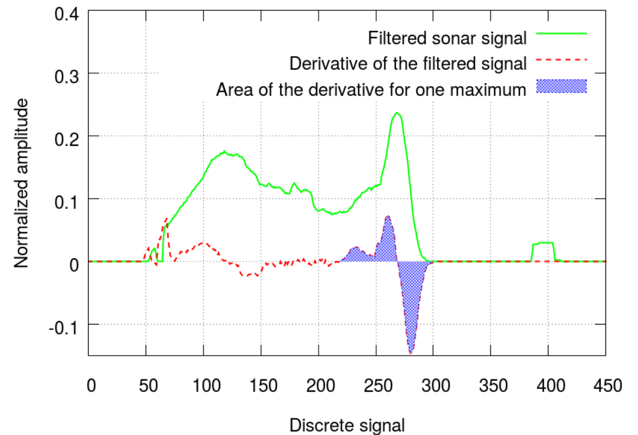



**Fig. 11** Example for the area used in $M(n)$ for one local maximum. Solid green line: The filtered signal $\bar{\bar{\mathbf{S}}}_\mathbf{k}$. Dotted red line: The derivative $\mathbf{S}'_\mathbf{k}$.

signal $\mathbf{S_k}$:

$$M_k = \left\{M(n) \mid n \in P_k\right\} \tag{15}$$

Where $P_k$ defines the indices of all candidates, $M_k$ gives a measurement for their corresponding strength in the signal. We use the strongest candidates of the last sonar signals to compute a threshold:

$$m_{threshold} = \frac{1}{b}\sum_{i=0}^{b} max(M_{(k-b)+i}) \tag{16}$$

$m_{threshold}$ is the average of the last $b$ maximum values of $M_k$. This threshold is used to reject weak candidates. Now we can define a function $W : P \rightarrow [0,1]$ to weight all candidates in relation to their value $M(n)$:

$$\forall n \in P_k,\ W(n) = \begin{cases} \frac{M(n)}{max(M_k)} & \text{if} M(n) > a \cdot m_{thresh.} \\ 0 & \text{else} \end{cases} \tag{17}$$

With the parameter $0 \leq a < 1$ the threshold can be adjusted in relation to the average maximum. A candidate is a vector $c_n$:

$$c_n \in C_k : c_n = \begin{pmatrix} d \\ \psi \\ w \end{pmatrix} \tag{18}$$

Here the weight $w$ of the sonar feature candidate will be extended by the polar coordinates of the feature. $d$ is the distance in meters and $\psi$ is the angle of the current signal. The result of the candidate extraction is now defined as a set $C_k$:

$$C_k = \left\{ \begin{pmatrix} dist(n) \\ angle(k) \\ W(n) \end{pmatrix} \mid n \in P_k \right\} \tag{19}$$

$dist(n)$ computes the distance for an index $n$ based on the spatial resolution of the signal.

*4.2.2 Reinforcing candidates on straight structures*

The former extraction of sonar feature candidates was, besides of the computation of the threshold, only considering the current signal. Most detectors are in particular interested in returns from straight continuous structures, like walls. To enforce such features we take the neighborhood of a signal into account to re-weight the candidates. The procedure can be outlined as follows:

1. Select candidates in a continuous set of sonar beams
2. Perform a non-discrete Hough transformation
3. Compute intersection points
4. Compute an intersection density for every intersection point
5. Use intersection points and density to re-weight the candidates

The set $E$ is the union of the best-weighted $m$ candidates of the last $l$ sets $C_i$ resulting from the previous candidate extraction:

$$E = \bigcup_{i=k-l}^{k} selectBest(C_i, m) \tag{20}$$

The method *selectBest* returns just the $m$ best candidates of a set $C_i$ in relation to their weight $w$.

In the next step we create for all elements of the set $E$ the corresponding representation a Hough space $(\alpha, r)$, which is a curve $r(\alpha) = x \cdot cos(\alpha) + y \cdot sin(\alpha)$. This procedure is well known in literature and therefore we will not discuss it here in detail [Hough, 1962] [Ballard, 1981]. Usually we select for $m$ and $l$ values between 3 and 5. This constrains the number of candidates transformed to the Hough space and is necessary for keeping the process real-time capable.

We compute all intersection points between the curves in the range of $r \geq 0, (angle(k) - \pi) < \alpha \leq (angle(k) + \pi)$. The range on the $\alpha$ axis can be limited, since the curve $r(\alpha)$ is a trigonometric function. The result is a set of intersection points $g \in G : g = \begin{pmatrix} \alpha \\ r \end{pmatrix}$.

Because we are not using a discrete world as Hough space, we use a two-dimensional Gaussian function to find the areas with the highest intersection density:

$$p(g_i) = \sum_{g \in G/g_i} \exp\left(-(\frac{(\alpha(g) - \alpha(g_i))^2}{2 \cdot \sigma_\alpha^2} + \frac{(r(g) - r(g_i))^2}{2 \cdot \sigma_r^2})\right) \tag{21}$$

$P_g = \{p(g) \mid g \in G\}$ is the set of all probabilities for all intersections in $G$. Every intersection point in $G$ represents a line in Hesse normal form in the image space. In the next step we calculate the intersection point of each of those lines for the current sonar signal.

$$dist(g_i) = \frac{r(g_i)}{cos(angle(k) - \alpha(g_i))} \tag{22}$$

We apply now the new weight given by $p(g_i)$ with one-dimensional Gaussian functions, while the distances given by $dist(g_i)$ are their offsets:

$$H(c) = \frac{1}{|G|} \sum_{g \in G} \frac{p(g)}{max(P_g)} \cdot e^{-\frac{d(c) - dist(g)}{2 \cdot \sigma_d^2}} \tag{23}$$

$$\forall c \in C_k,\ W_{\text{new}}(c) = a \cdot H(c) + (1 - a) \cdot W(c) \tag{24}$$

How much impact this new weight has can be selected by the parameter $0 \leq a \leq 1$. We use then again a dynamic threshold to filter the new weighted candidates similar to the previous extraction approach.

This candidate reinforcement method helps to remove isolated features and favor features on straight continuous structures.

## 5 Wall detection

The challenge when trying to detect a structure, like a wall, in the sonar scan is to deal with outliers, uncertainties, a slow actualization rate and a limited field of view. The sonar field of view is mostly limited to an area of interest (angular window with limited distance) to increase the actualization rate of the sensor. Since a quick rotation of the AUV could move the structure out of the field of view, the detector should also avoid to produce false positives.

As basis for the wall detector the sonar feature estimation process with the reinforcement of candidates on straight structures is used, as explained in section 4.2.

As a first iteration we were using the RANSAC algorithm to estimate lines in the sonar features. But the RANSAC algorithm had the tendency to produce false positives on this kind of data and was therefore to unusable to keep track of a structure. An approach that has been shown to be suitable for us, is the use of weighted center points. For this purpose the complete sonar scan, with a limited opening angle ($< 180°$), is divided into a left and a right area $(A_l, A_r)$. For each of this areas the center point of the best feature of every sonar beam inside of this area is computed. Additionally the features are weighted by their age using a fading out factor $0 < \gamma_{fading} < 1$:

$$p_l = \frac{1}{\sum_{S_i \in A_l} (t-i) \cdot \gamma_{fading}} \sum_{S_i \in A_l} (t-i) \cdot \gamma_{fading} \cdot ES(selectBest(C_i)) \tag{25}$$

$$p_r = \frac{1}{\sum_{S_i \in A_r} (t-i) \cdot \gamma_{fading}} \sum_{S_i \in A_r} (t-i) \cdot \gamma_{fading} \cdot ES(selectBest(C_i)) \tag{26}$$

The function $ES$ transforms the polar coordinates to euclidean space. The detected wall is a line defined through the center points $p_l$ and $p_r$. Figure 12 illustrates the partition of the sonar feature candidates into two sets.

If it is hard to determine one concrete line in the set of candidates, the results of the RANSAC algorithm will jump when trying to detect the most likely line. The center point approach on the other hand has shown to compute stable results under this circumstances and is therefor suitable to work with a lot of outliers and high uncertainties. This might increase the overall uncertainty of the results but has less false positives and jumps. The slow actualization rate of the sensor is also taken into account by weighting the features regarding their age.

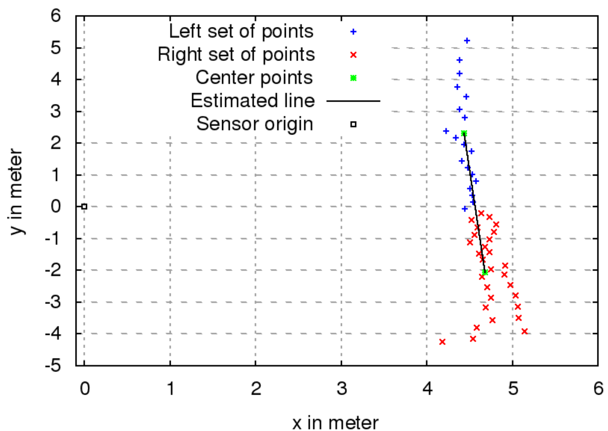



**Fig. 12** Left and right set of sonar feature candidates, center points of the sets and the line defined by the center points. The line represents the most likely pose of the wall.

Additionally, the estimated walls are checked for their stability regarding the distance and the angle of the wall. If the values stay within a boundary for a defined time span, the detected walls are considered as valid. In numerous experiments and competitions it was shown that this comparatively simple detector is suitable and very robust for the inspection of walls and even corners can be handled appropriately.

## 6 Localization

For estimating a global position a probabilistic localization approach using a particle filter has been implemented based on the theories of [Thrun, 2002]. Each particle represents a weighted state hypothesis and is described by the vector:

$$\mathbf{X_t} = \begin{bmatrix} \mathbf{p_t} \\ \mathbf{v_t} \end{bmatrix}' = \begin{bmatrix} p_{x,t} \; p_{y,t} \; v_{x,t} \; v_{y,t} \end{bmatrix} \tag{27}$$

The orientation $\mathbf{\Theta_t} = (\mathbf{\Phi_t}, \theta_\mathbf{t}, \psi_\mathbf{t})$ is not part of the state hypothesis because it is already estimated by an Extended-Kalman-Filter (EKF) from IMU and FOG sensors. This means we apply each orientation sample from this EKF directly to all particles in the model and separate it from the state estimation in the particle filter. The same also holds for incoming depth samples $p_{z,t}$ from the pressure sensor.

A particle filter requires specific models for weighting the particle set based on current measurement samples and processing a state transition for each particle within the dynamic update. These models are described as follows:

6.1 Transition Model

The particle filter requires a state transition model that draws a new position sample for each particle from the following distribution:

$$\mathbf{x}^{[i]} \sim P(\mathbf{X}_t^{[i]} | \mathbf{X}_{t-1}^{[i]}, \mathbf{U}_t) \tag{28}$$

The dynamic update receives an approximated speed sample $\mathbf{v_t}$ and a relating covariance matrix $\Sigma_{v,t}$ from the motion model, described in section 4.1. For the localization component we use a simplified, kinematic transition model that linearizes the motion derivation from the speed samples $\mathbf{v_{t-1}}, \mathbf{v_t}$. We draw new motion samples from the distribution $\mathbf{v_{uvw}} \sim N(\mathbf{v_t}, \Sigma_{v,t})$ and calculate the average velocity $\mathbf{v_{avg}}$. A state transition for a single particle can then be computed by:

$$\mathbf{v_{avg}} = \frac{\mathbf{v_{t-1}} + \mathbf{v_{uvw}}}{2} \tag{29}$$

$$\mathbf{X_t^{[i]}} = \begin{bmatrix} \mathbf{p_{t-1}^{[i]}} + rotate(\mathbf{v_{avg}} \cdot \Delta t, \Theta_{t-1}) \\ \mathbf{v_{uvw}} \end{bmatrix} \tag{30}$$

6.2 Perception Model

The perception model evaluates the current particle set for a given measurement sample and a static map representation. As measurement samples the candidates of the sonar candidate estimation, explained in section 4.2, are used. Formally, we calculate for each particle, the weights from the following distribution:

$$w_t^{[i]} = P(\mathbf{X_t^{[i]}} | \mathbf{Z_t}, \mathbf{M_t}) \tag{31}$$

For computing reasonable weights, the perception update depends on an appropriate map representation. The map itself is organized in a hierarchical tree of landmarks and line-based obstacles (specified by $\mu_L \in \mathbb{R}^3, \Sigma_L \in \mathbb{R}^{3\times3}$ and by two points $a, b \in \mathbb{R}^3$). Each wall surface is modelled by a line and for the sonar-based perception update we determine the expected obstacle point $\mathbf{z_t^*}$ on a wall for a given measurement $\mathbf{Z_t}$ with a ray-tracing computation (line intersection). The distance between the expected and real measurement influences the probability of the particles that is formed by a Gaussian function parametrized with $\mu = 0.0$ and a configurable variance $\sigma_{sonar}$.

This perception model for sonar- and map-based measurements is described in detail by the following algorithm:

**Function** sonarupdate($\mathbf{X_t^{[i]}}, \mathbf{Z_t}, \mathbf{M_t}, \epsilon$) $\rightarrow w_t^{[i]}$

```
if not isPointInWorld(X_t^[i], M_t) then
  return 0.0
else if z_min > range(Z_t) or range(Z_t) > z_max then
  φ_t^[i] = 1/N
else
  z_rel,t = point(Z_t)
  z_abs,t = rotate(z_rel,t, Θ_ψ) + p_t^[i]
  wall = getNearestWall(z_abs,t, M_t)
  beam = line(p_t^[i], z_abs,t)
  z_t^* = intersection(wall, beam))
  d = sqrt((z_t^* − z_abs,t)(z_t^* − z_abs,t))
  φ_t^[i] = N(μ = 0.0, σ_sonar, d)
end if
return φ_t^[i]
```

A primary design issue for the perception model is the fusion of sonar measurement and the results from our visual detectors (buoy, pipeline) in order to increase the accuracy for our estimated positions. The perception model for sensing a buoy and pipeline uses the landmark representation in $\mathbf{M_t}$ and computes the Mahalanobis distance to the nearest, visual feature in our map with:

$$\sqrt{(\mu_\mathbf{L} - \mathbf{z_{abs,t}}) \mathbf{\Sigma_L}^T (\mu_\mathbf{L} - \mathbf{z_{abs,t}})}. \tag{32}$$

6.3 Selective resampling and state estimation

Resampling is an important task for particle filters and it replaces particles with low importance weights $w_t^{[i]}$ by samples with high weights. This step is necessary in order to improve the approximation of the current state space (particles) due to the characteristics of using only a finite number of particles. On the downside it can delete good estimations from the particle set, which is a great problem if the particle-distribution is nearly uniform. To prevent unnecessary resampling-steps, we use a measure for the dispersion over the weights of all particles, introduced by Liu and Stachniss [Liu, 1996, Stachniss, 2009], to trigger the resampling. The *effective number of particles* is computed with normalized weights $w_t^{[i]}$ by:

$$N_{eff} = \frac{1}{\sum_{i=1}^{N} (w_t^{[i]})^2} \tag{33}$$

and attains values in the range $1 \leq N_{eff} \leq N$. If the weights of the particles are uniformly distributed, $N_{eff}$ will be N.

$$\forall i \in N : w_t^{[i]} = \frac{1}{N} \Longrightarrow N_{eff} = N \tag{34}$$

If the probability is concentrated completely in a single particle s, $N_{eff}$ results in:

$$\exists s \in N : w_t^{[s]} = 1.0 \Longrightarrow N_{eff} = 1.0 \quad (35)$$

Our implementation performs resampling automatically iff the computed $N_{eff}$ does not exceed a specific, configurable threshold. This means the number of resampling steps can be reduced depending on the threshold for $N_{eff}$ and this takes place if the dispersion of the weights is too high.

The vehicle state is estimated by calculating the weighted average of all particles. This procedure is more stable and robust to outliers than picking the best particle. Because the weights are already normalized, we only need to calculated the weighted sum.

$$X_t = \sum_{i=0}^{n} \phi_t^{[i]} \cdot X_t^{[i]} \quad (36)$$

One drawback of the average calculating is the case if we have no convergence or multiple converge centers. So it is important to calculate the variance of the particles. If the variance exceeds a given threshold, we can assume that the position estimation is invalid, and we cannot navigate, based on this estimation.

### 6.4 Mapping

As an extension, a map of sonar and depth-features is built, in which detectable landmarks (e.g. pingerbox, buoys, pipe-structures) can also be mapped. So the landmarks are identified as single sonar features from our feature extraction algorithm and no visual detection is used. Because the landmarks are not distinguishable from each other in the sonar and we have no unique landmarks, the DP-SLAM algorithm [Eliazar and Parr, 2003] is used.

This algorithm extends the particle filter with an occupancy grid map, in which in every cell, a tree structure of observed features for every particle is saved. In this way, there exists a feature map for every particle, but by using a hierarchical tree structure in each gridcell the storage can be reduced. This is based on the assumption that each map of the particles also contains the features of the parent particle.

By storing multiple maps instead of landmarks, the data association problem is avoided. The association problem is substituted by a localization problem. By using a sufficient number of particles, there is no need for an explicit loop-closure algorithm.

During our tests at the NURC, the algorithm was able to create an online depth-map of an area of the size $50m \times 30m$ with an cell-size of 0.5m and using 500 particles.

## 7 Alternative Localization Approach

Additionally to the introduced pose estimation based on particle filters, an alternative approach for the localization of the AUV has been implemented. This method performs computer vision algorithms on sonar samples retrieved from the Micron sonar which is mounted on top of Avalon. The general aim is to buffer the sonar data of a full 360° scan, detect walls of the basin and estimate the position of the AUV in respect to these walls. The key idea of this approach was taken from [Ribas et al., 2010]. To avoid unnecessary conversions, the algorithm works directly on the polar space frame from the sonar sensor. This also simplifies the compensation of dynamic rotations from the vehicle during the measurement to apply an offset to each angle of a sonar beam. Nevertheless, the images shown in this section are converted into cartesian space for better comprehension.

Since the processing would cause a high peak in CPU usage after each 360° scan, the filtering and the wall detection algorithm have been designed for continuously processing the incoming data and thus relieving the CPU from high peak loads.

The sonar candidate estimation process, explained in section 4.2, is not used in this approach. The intention behind this was to be independent, not only from the particle filter based localization, but also from the previous sonar processing.

The localization is basically divided into four parts:

1. Filtering of the raw sonar data
2. Performing a Hough transform
3. Selecting the most likely lines in the parameter space
4. Computing the position in relation to the previous selection

The incoming sonar data is a stream of subsequent beams. Each beam contains (amongst other things) an angle and a field of bins. Each bin contains the accumulated strength of an acoustic echo for a specific distance.

### 7.1 Filtering

Figure 13 shows a 360° scan of the NURC basin. It is apparent that strong echoes are present within the whole range of the basin. Outside of the basin the echoes become significantly weaker, but noisy peaks resulting

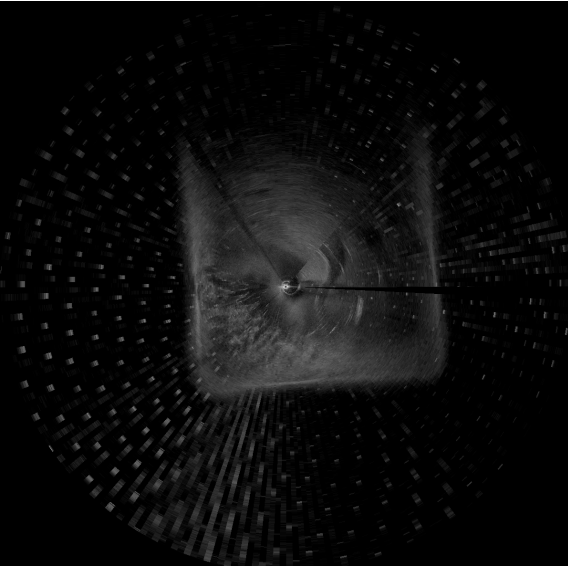


**Fig. 13** Raw sonar data recorded at SAUC-E 2011 in a harbour site, brighter spots represent stronger echoes

from reflections of the sonar beam on waves also exist. The goal of the filtering step is to keep only the echoes that represent the walls of the basin in order to have a good starting point for the Hough transform.

The filtering itself consists of three steps. The first step is the application of a minimum filter with a $5 \times 5$ convolution kernel. With this step the noisy sonar data can be smoothed. In a swimming pool of the University of Bremen the minimum filter is not necessary because of the smaller dimensions of the basin. Hence this is designed as an optional processing step.

The next phase is a Canny filter for edge detection (e.g. the walls of the basin). In this case it is a combined Gaussian and Sobel filter. Since there are different Sobel operators for both distance and angular direction, the input beams must be filtered twice. For an optimization, the linear time-invariant characteristics of convolution is used to separate a filter kernel $H_i \in \mathbb{R}^{5 \times 5}$ into two smaller ones $H_1, H_2^T \in \mathbb{R}^{5 \times 1}$. With this separation two additional filters running across the image are necessary but the number of calculations for each bin is reduced from 25 to 10.

For the edge detection in distance and angular direction the $H_D, H_A \in \mathbb{R}^{5 \times 5}$ kernels are split up into:

$$H_D = \left[ -\tfrac{1}{4} \; -\tfrac{1}{2} \; 0 \; \tfrac{1}{2} \; \tfrac{1}{4} \right] * \begin{bmatrix} \frac{1}{4} \\ 1 \\ \frac{3}{2} \\ 1 \\ \frac{1}{4} \end{bmatrix} \tag{37}$$

$$H_A = \left[ \tfrac{1}{4} \; 1 \; \tfrac{3}{4} \; 1 \; \tfrac{1}{4} \right] * \begin{bmatrix} -\frac{1}{4} \\ -\frac{1}{2} \\ 0 \\ \frac{1}{2} \\ \frac{1}{4} \end{bmatrix} \tag{38}$$

The filters always keep the last 5 sonar beams in memory to allow continuous filtering in the direction of the beam's angle.

The peaks (for instance the local maxima of each filtered beam) that exceed a given threshold are passed to the next step.

Since the motion of the vehicle can lead to distortion of the full sonar scan, the motion model, described in section 4.1, is used to compensate the motion. The estimated motion is used to transform each single peak in the world frame by the relative difference in the position. This leads to more accurate sonar scans and a more stable algorithm, even at high vehicle velocities.

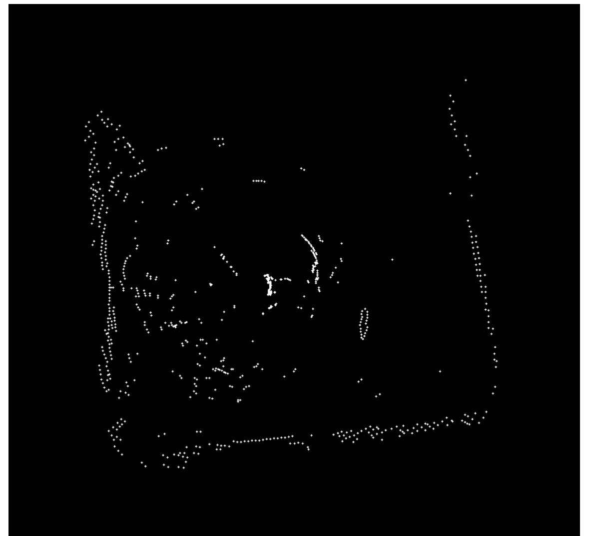


**Fig. 14** Sonar peaks of the basin after filtering the data from Fig. 13

7.2 Hough transform for line extraction

The Hough lines transform is a voting algorithm for detecting lines. The classical implementation runs pixel wise on a Sobel- or Canny-filtered image. In this approach the transform runs on the peaks retrieved from the filtering step. Each peak is represented by a 2-tupel $p \in \mathbb{R}^2$, described by an angle $\theta_p$ and a distance $d_p$ to the AUV. The lines to be detected are also represented as a 2-tupel $(\theta_l, d_l)^T, \in \mathbb{R}^2$ being the angle of the normal

and the shortest distance to the AUV. The Hough line transform provides a 2D field the *Hough space* with the dimensions being the parameters of the lines. Then all possible lines that touch the considered pixel result in a high accumulation of the equivalent cell in the Hough space and can be evaluated in the next phase.

In our approach we consider the fact that a sonar pulse hitting a wall in a too obtuse angle is not reflected back to the emitter. So only those lines are taken into account, for which the given echo does not exceed a certain angle threshold. Additionally, the sonar peaks are not very accurate in the distance. Knowing this, the values of all lines are not increased by one. Instead a function is used to accumulate the values of surrounding lines with respect to their distance from the peak and their angle. This is basically a combination of two Gaussian curves:

$$Acc(p,l) = 0.01 \cdot E(p) \cdot Acc_{max} \cdot e^{-0.5 \cdot \frac{(\Delta\alpha)^2}{\sigma_\alpha^2}} \cdot e^{-0.5 \cdot \frac{(\Delta d)^2}{\sigma_d^2}} \tag{39}$$

where $E(p)$ is the intensity of the peak, $Acc_{max}$ is the maximum value to increase, $\Delta\alpha = \theta_p - \theta_l$, $\Delta d = d_p - d_l$ are the differences for the angles and distances of a peak $p$ and line $l$. $\sigma_{alpha}$, $\sigma_d$ are constant scaling values.

Figure 15 shows a part of the Hough space after handling a single sonar peak.

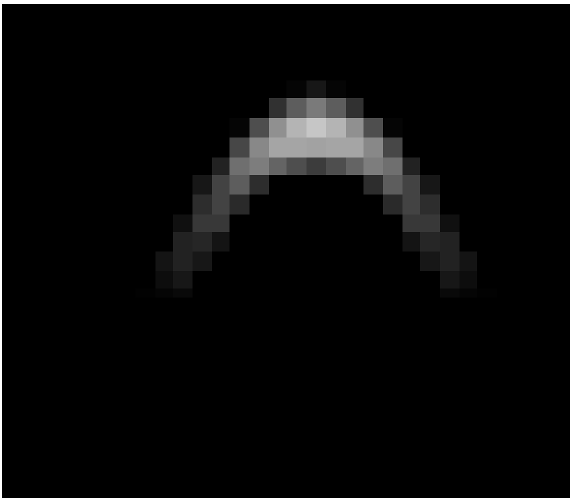


**Fig. 15** Part of Hough space after handling a single sonar peak

Whenever a full 360° scan has been performed, local maxima are searched within the Hough space. These are the possible basin walls for the next step.

### 7.3 Line selection

For the selection of the correct lines the actual heading of the AUV is used. If the orientation of Avalon with respect to the basin walls is known at the beginning of any mission, the actual orientation of the basin walls with respect to the AUV can be computed. Thus, the lines running approximately length- or crosswise to the basin can be separated. The other lines are dropped. For each line a partner line is being assigned that has approximately the correct distance (known from basin size). If no line meets this requirements, a partner line is guessed. All these line pairs now get a score based on the actual distance and the votes of the Hough space. For both length- and crosswise running lines the best pair is selected. These four lines represent the detected basin (see Fig. 16).

The basin of the NURC contains only three walls. Nevertheless a fourth wall is assumed to run the same detector on different basins. Our line selection-algorithm is implemented for lines in a rectangular geometry, since this geometry is usual for basins. But it would be possible to extend the algorithm for non-rectangular environments.

With the knowledge of the basin walls with respect to the vehicle the position of the AUV itself is computable. For the position along the x-axis, the distances of the two crosswise running lines is stretched to such an extend that the distance between them matches the width of the basin $w_B$. Then the x-coordinate can be calculated as the difference of the distance to the line that is in direction of positive x and half of the basin width. The y- coordinate is calculated analogously in respect to the height $h_B$:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{-(C_L+C_R)}{2\cdot|C_L-C_R|} \cdot w_B \\ \frac{-(L_L+L_U)}{2\cdot|L_L-L_U|} \cdot h_B \end{pmatrix} \tag{40}$$

with $C_L$, $C_R$ being the distance value of the crosswise running lines; $L_L$, $L_U$ for the lengthwise running lines analogously.

### 7.4 Usage of the alternative Approach

The explained approach provides quite accurate but slowly updating localizations. The speed results from the need of a full 360° scan that takes approx. 5 to 10 seconds for a mechanical scanning sonar. Additionally, the position becomes inaccurate when Avalon is moving during the scan. Thus this method is not qualified for a stand-alone localization.

But the approach can be used as an initialization for the particle filter (6) based localization at the startup

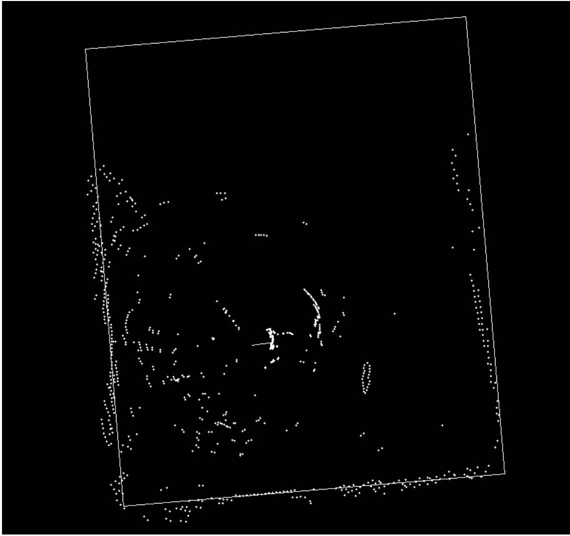


**Fig. 16** The detected basin from the sonar peaks in Fig. 14

or an interspersal update that generates new particles with a computed position to particle set. A higher layer could also force a reinitialization, when both localization approaches differ a lot during a longer period of time. Additionally, the heading of the AUV measured by the IMU drifts over the time. With the knowledge of the actual orientation of the walls this drift can be measured and reported back to the system, where a correction can be performed.

## 8 Control and Navigation

The primary purpose for developing an operative localization is the resulting feasibility for the system to navigate autonomously in the environment. It forms for us the fundamental capability for a robotic system to follow well designed plans and especially to react reasonably in failing states of a mission. We developed a new complete generic control chain for AUVs. The new design is illustrated in figure 17.

The control chain is designed to control any kind of underwater vehicle with up to six degrees of freedom. Due to the generic design of this component, not all axis need to be actuated. The underlying design of the control chain is a chain itself, on the lowest level only force control is done. The upper layers build a chain up to global world position controlling. Each axis is separate controllable from each others.

As shown in figure 3, in the lower part, every link of the chain receives a pose sample and only the lowest *AlignedToBody*-controller is responsible for the actual control of the thruster.

One key idea is that the robot control on higher layers should not be done using forces. The *AlignedToBody*-controller gets commands from the *AlignedVelocity* -controller. This PID-based controller controls the velocity of the robot, a kind of information humans can think with. Therefore, from this level on, it is possible to give concrete 6DOF commands to control the robot.

To control the vehicle in a way to target a position five meter in front of it, the *AlignedPosition*-controller can be used. The *Aligned* frame is a body-fixed frame, where the z-axis point to the equal direction than the z-world-axis. This layer was introduced because most of the current AUVs operate locally in a plane-like mode. Therefore, this simplified layer was developed to reduce the influence of pitch and roll which is mostly caused by a common swinging system-control. However, the generated control information from the *AlignedPosition*-controller are chained to the velocity based *AlignedToBody*-controller.

The last link of the actual control-chain is the *WorldToAligned*-controller. This controller receives global positioning requests where the robot should move to. Similar to the lower layers this is a 6DOF control request including heading, pitch and roll, even the system cannot actively control these axes due to limited thrusters. The control information again are transformed and chained to the lower layer, the *AlignedPosition*-controller.

Due to the fact of the modular design we also introduced a *GroundAvoidance* task which actively overrides the desired depth to a save-depth to keep a defined minimal distance to the ground. This override is currently done by injecting the *GroundAvoidance* in front of the world-z input of the existing chain. At this point, the algorithmic implementation is straight forward.

$$z(target) = \begin{cases} z_r, & \text{if}(z_c - a + d_m) < z_c \\ z_c - a + d_m, & \text{otherwise} \end{cases} \tag{41}$$

Where $z_t$ defines the target depth
$z_c$ defines the current depth
$z_r$ is the requested depth from higher layers
$z_g$ is the depth of the ocean floor
$d_m$ defines the minimal ground distance
$a$ defines reading from the altimeter

It is possible to combine all of the commands to be able to control the vehicle in the following way: "drive 50 meters forward, constantly spinning in a depth of 4 meters", which results in a chain where the *WorldPosition* task does the depth control, the *AlignedPosition* moves for a distance of 50 meters forward, and the *AlignedVelocity* task does a constant spinning move.

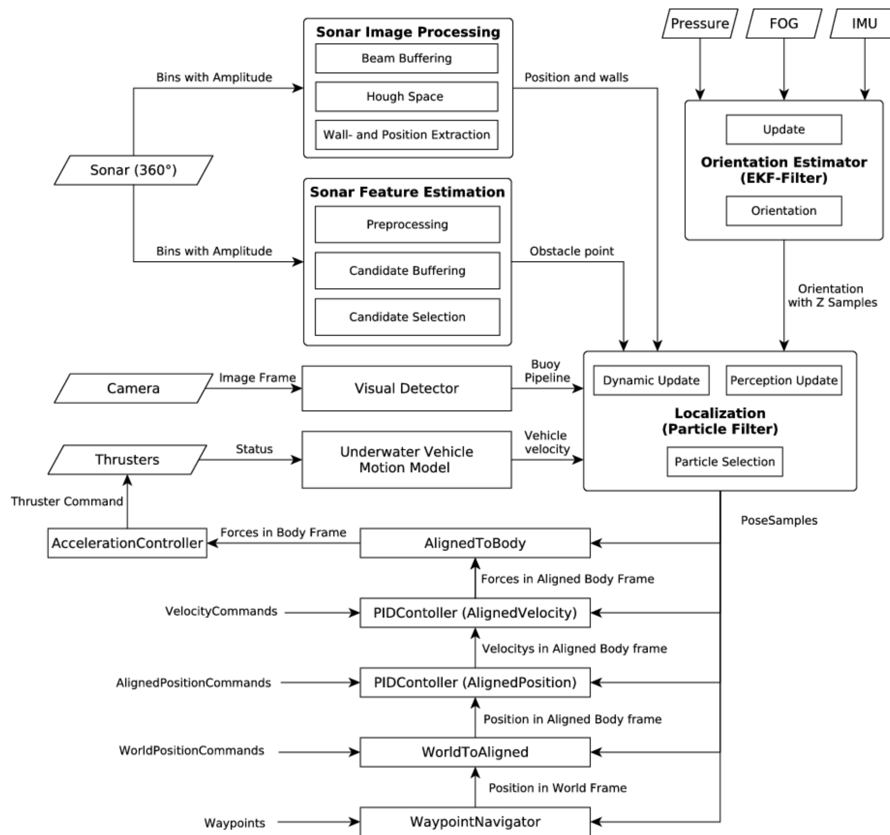



**Fig. 17** Avalons component structure for navigation

## 9 Evaluation

We presented two approaches for estimating the global position within a known environment. To evaluate both approaches some experiments have been performed in a swimming pool provided by the University of Bremen. To obtain the true reference position of the robot, a self-developed laser-based tracking system was used. The system uses a Hokuyo laser range finder, statically mounted outside of the pool, to measure permanently the distance to a reflector. The refelctor was attached onto the compartment of the robot and was sticking out through the water surface. Via triangulation and multiple aligned laser beams in different angles, it is possible to extract the real position of the vehicle within the pool, that provides a reference measurements with low uncertainties for the pose estimation approaches in this environment. The system is shown in Figure 18 and is explicitly described in [Goldhoorn, 2010].

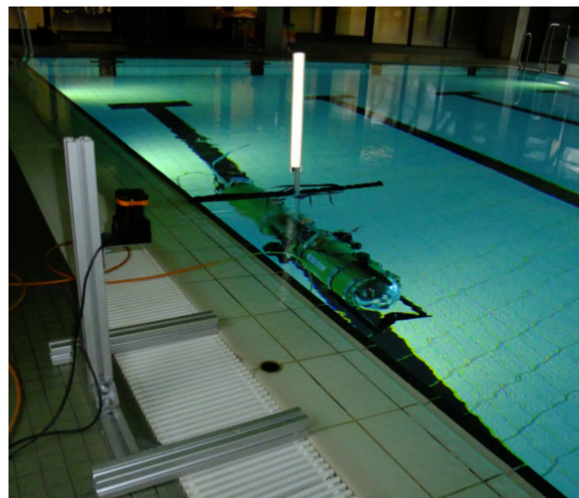


**Fig. 18** Avalon equipped with a Laser-based tracking system ontop of it[Goldhoorn, 2010]

For a comparison of the two localization-approaches, the particle filter and the Hough-transformation-based localization, Avalon followed one trajectory equipped with the tracking device in the $17.3 \times 10.0 \times 2.20$ sized pool and measured the distances to the walls with a 360° configured, mechanical sonar, continuously. For the evaluation the particle filter implementation was not connected with the velocity motion model because the development of this component was still in progress. This is replaced by a simple Gaussian noise with $\mu =$

$\begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}$ and $\sigma = \begin{bmatrix} 4.0 & 0.0 \\ 0.0 & 4.0 \end{bmatrix}$ in each dynamic step. Figure 9 lists the current, statistical results from estimated positions attained.

| | Hough Localization | Particle Filter |
|---|---|---|
| average error | 0.85m | 1.16m |
| max error | 5.15m | 4.64m |
| min error | 0.01m | 0.07m |
| variance | 0.95m$^2$ | 0.56m$^2$ |
| standard dev | 0.97m | 0.75m |

**Fig. 19** Statistical characteristics for Hough- and particle localization

The experiments pointed out that the Hough localization provides a higher average accuracy in comparison to the particle filter. But it is affected by a small number of outliers with high peaks caused by the dynamics of the vehicle and the resulting motion artifacts in the sonar image. This can also be observed in Figure 20 that shows the euclidean error between the given position from the laser tracking system and the hough localization.

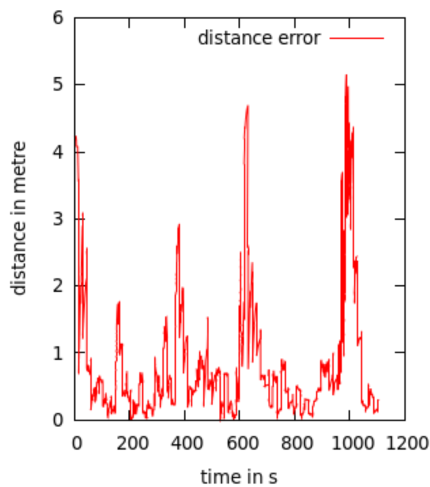



**Fig. 20** Euclidean error of the Hough localization

## 10 Conclusion

In this paper we present the hardware, the sensor processing algorithms and the control system of our reliable autonomous underwater vehicle Avalon. The proposed system has been successfully tested during two European challenges SAUC-E and Eurathlon over four years. In this both international competitions Avalon has gathered following prices, respectively:

- 2009 3'th place at SAUC-E
- 2011 3'th place at SAUC-E
- 2012 *best qualifier* award at SAUC-E
- 2014 1'th place at SAUC-E
- 2014 3'th place for *Combined scenario* at Eurathlon
- 2014 2'th place for *Leak localisation and structure inspection* at Eurathlon
- 2014 2'th place for *Interaction with underwater structures* at Eurathlon
- 2014 1'th place for *Environmental survey of an accident area* at Eurathlon

These results show the capabilities of the system, which are continuously improved. Through the regular Avalon's participation in those competitions we had an opportunity to test our vehicle in real challenging environments. This enables us to improve our algorithms and, thus, the overall system robustness, continuously.

Furthermore, we present a new approach for localization which is based on two separated detectors which has shown it's functionality in different areas. We developed a new approach for six DoF hovering AUV, in which a generic motion control chain is used. Additionally, based on the modular system design and the abstraction, the knowledge about the system can be easily passed on to the new collaborators and users.

All the software mentioned within this publication is a open source software and can be accessed at github: `https://github.com/auv-avalon`.

## 11 Outlook

In our future work we are going to develop further model-based approaches for system control, and data processing. In the next years, we are going to present additional improved algorithms for navigation and mapping. These algorithms should cover unstructured environments. Furthermore, we would like to develop another system which includes a DVL for better motion-estimation and a higher operation accuracy. We are planning to extend the system with underwater-docking capabilities to train long-time autonomous operation and perform tests in real environments. Our in 3.3 described plan manger will be completely revised. We figured out that a linear-constraint solver like Roby is not succifent for complex system's like Avalon is. Therefore we currently reseaching for other more artificial intelligence (AI) based solutions for the management of component model based systems.

## 12 Acknowledgement

This work has been supported by the Graduate School SyDe, funded by the German Excellence Initiative within the University of Bremen's institutional strategy. We also would like to thank the previous *Avalon* student teams AUV07, AUV09, AUV10, AUV12 for their excellent work during the project. Special thanks to Hennig Kost for his previous work on the alternative localization approach as explained in 7.

## References

roc, 2012. (2012). Rock, the Robot Construction Kit. http://www.rock-robotics.org.

Antonelli and Antonelli, 2014. Antonelli, G. and Antonelli, G. (2014). *Underwater robots*. Springer.

Ballard, 1981. Ballard, D. H. (1981). Generalizing the hough transform to detect arbitrary shapes. *Pattern recognition*, 13(2):111–122.

Einhorn et al., 2011. Einhorn, M., Roessler, W., and Fleig, J. (2011). Improved performance of serially connected li-ion batteries with active cell balancing in electric vehicles. *Vehicular Technology, IEEE Transactions on*, 60(6):2448–2457.

Eliazar and Parr, 2003. Eliazar, A. and Parr, R. (2003). Dp-slam: Fast, robust simultaneous localization and mapping without predetermined landmarks. In *IJCAI*, volume 3, pages 1135–1142.

Goldhoorn, 2010. Goldhoorn, M. (2010). Unterwasser slam (underwater slam). diploma thesis, University of Bremen, Faculty of Computer Science.

Goldhoorn and Joyeux, 2014. Goldhoorn, M. and Joyeux, S. (2014). Extension of a plan-based component manager for real time adaptation. In *ISR/Robotik 2014; 41st International Symposium on Robotics; Proceedings of*, pages 1–6. VDE.

Hough, 1962. Hough, P. V. (1962). Method and means for recognizing complex patterns. US Patent 3,069,654.

Joyeux et al., 2009. Joyeux, S., Alami, R., Lacroix, S., and Philippsen, R. (2009). A plan manager for multi-robot systems. *The International Journal of Robotics Research*, 28(2):220–240.

Joyeux et al., 2011. Joyeux, S., Albiez, J., et al. (2011). Robot development: from components to systems. In *6th National Conference on Control Architectures of Robots*.

Lee et al., 2011. Lee, W. C., Drury, D., and Mellor, P. (2011). Comparison of passive cell balancing and active cell balancing for automotive batteries. In *Vehicle Power and Propulsion Conference (VPPC), 2011 IEEE*, pages 1–7.

Liu, 1996. Liu, J. S. (1996). Metropolized independent sampling with comparisons to rejection sampling and importance sampling. *Statistics and Computing*, 6(2):113–119.

Lütkebohle et al., 2011. Lütkebohle, I., Philippsen, R., Pradeep, V., Marder-Eppstein, E., and Wachsmuth, S. (2011). Generic middleware support for coordinating robot software components: The task-state-pattern. *Journal of Software Engineering for Robotics*, 2(1):20–39.

Quigley et al., 2009. Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*.

Ribas et al., 2010. Ribas, D., Ridao, P., and Neira, J. (2010). *Underwater SLAM for structured environments using an imaging sonar*, volume 65. Springer.

Smith et al., 1997. Smith, S. W. et al. (1997). The scientist and engineer's guide to digital signal processing.

Stachniss, 2009. Stachniss, C. (2009). *Robotic mapping and exploration*, volume 55. Springer.

Tarascon and Armand, 2001. Tarascon, J.-M. and Armand, M. (2001). Issues and challenges facing rechargeable lithium batteries. *Nature*, 414(6861):359–367.

Thrun, 2002. Thrun, S. (2002). Probabilistic robotics. *Communications of the ACM*, 45(3):52–57.