STUTTGARTER BEITRÄGE ZUR PRODUKTIONSFORSCHUNG

Florian Weißhardt

»Automatisiertes Test- und Evaluierungs-framework für die Servicerobotik«

Fraunhofer IPA

Universität Stuttgart

STUTTGARTER BEITRÄGE ZUR PRODUKTIONSFORSCHUNG BAND 166

Florian Weißhardt

»Automatisiertes Test- und Evaluierungsframework für die Servicerobotik«

Herausgeber

Univ.-Prof. Dr.-Ing. Thomas Bauernhansl[1,2]
Univ.-Prof. Dr.-Ing. Dipl.-Kfm. Alexander Sauer[1,3]
Univ.-Prof. Dr.-Ing. Kai Peter Birke[4]
Univ.-Prof. Dr.-Ing. Marco Huber[1,2]

[1]Fraunhofer-Institut für Produktionstechnik und Automatisierung IPA, Stuttgart
[2]Institut für Industrielle Fertigung und Fabrikbetrieb (IFF) der Universität Stuttgart
[3]Institut für Energieeffizienz in der Produktion (EEP) der Universität Stuttgart
[4]Institut für Photovoltaik (*ipv*) der Universität Stuttgart

Kontaktadresse:
Fraunhofer-Institut für Produktionstechnik und Automatisierung IPA
Nobelstr. 12
70569 Stuttgart
Telefon 0711 970-1101
info@ipa.fraunhofer.de
www.ipa.fraunhofer.de

Bibliographische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliografische Daten sind im Internet über http://dnb.de abrufbar.

Zugl.: Stuttgart, Univ., Diss., 2023

D 93

2024

Druck und Weiterverarbeitung:
Fraunhofer Verlag, Mediendienstleistungen, Stuttgart, 2024
Für den Druck des Buches wurde chlor- und säurefreies Papier verwendet.

Automatisiertes Test- und Evaluierungsframework für die Servicerobotik

Von der Fakultät Konstruktions-, Produktions- und Fahrzeugtechnik

der Universität Stuttgart

zur Erlangung der Würde eines Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Abhandlung

Vorgelegt von

Florian Weißhardt

aus Heilbronn

Hauptberichter:	Prof. Dr.-Ing. Dr. h.c. mult. Alexander Verl
Mitberichter:	Prof. Dr.-Ing. Rainer Müller
Tag der mündlichen Prüfung:	7. März 2023

Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen (ISW)

der Universität Stuttgart

2023

Vorwort des Autors

Die vorliegende Arbeit entstand während meiner Tätigkeit in der Abteilung Roboter- und Assistenzsysteme des Fraunhofer Instituts für Produktionstechnik und Automatisierung IPA in Stuttgart sowie bei der Mojin Robotics GmbH.

Mein besonderer Dank gilt Professor Dr.-Ing. Dr. h.c. mult. Alexander Verl für die Förderung meiner wissenschaftlichen Arbeit und die Übernahme des Hauptberichts. Bei Professor Dr.-Ing. Rainer Müller bedanke ich mich für die Unterstützung bei der Durchsicht meiner Arbeit und die Übernahme des Mitberichts. Für die organisatorische Betreuung des Promotionsprozesses und stets helfende Hände bedanke ich mich herzlich bei Heide Kreuzburg.

Bei der Realisierung der vorliegenden Arbeit profitierte ich von zahlreichen Erfahrungswerten, die meine Kollegen und ich in interdisziplinärer Teamarbeit über die vergangenen Jahre in verschiedenen Industrie- und Forschungsprojekten sammeln konnten. Mein Dank gilt daher all meinen Kollegen und Studenten, die durch Wissensaustausch, intensive Diskussionen und ihre fachliche Kompetenz zur Verbesserung unserer mobilen Roboter und der zugehörigen Softwarearchitekturen beigetragen haben. Besonders möchte ich mich bei meinen unmittelbaren Kollegen aus der IPA Arbeitsgruppe 326 sowie dem Team bei Mojin Robotics bedanken. Ihre hervorragende Zusammenarbeit und die stets freundschaftliche Atmosphäre haben meine Arbeit entscheidend beeinflusst. Ein herzliches Dankeschön geht an meine beiden Mojin-Mitgründer und langjährigen Weggefährten Dr.-Ing. Ulrich Reiser und Dr.-Ing. Tim Fröhlich, die mit ihrem Rat und Zuspruch wesentlich zur Fertigstellung dieser Arbeit beigetragen haben. Weiterhin bedanke ich mich besonders bei Felix Meßmer, Benjamin Maidel und Hannes Bachter für die inspirierende Zusammenarbeit und stetige Unterstützung bei der Implementierung unserer Ideen.

Für ihre moralische Unterstützung während der Entstehung dieser Arbeit bedanke ich mich von Herzen bei meinen Eltern, Barbara und Werner Weißhardt, bei meinen Schwiegereltern, Birsen und Orhan Oto, sowie meinem Bruder Dominik Weißhardt mit Familie.

Von ganzem Herzen danke ich meiner Ehefrau Asli für ihr Verständnis und andauernde Unterstützung. Ohne sie wäre die Erstellung dieser Arbeit nicht möglich gewesen. Ihr und meinen Kindern Nuno, Lisa und Bruno ist diese Arbeit gewidmet.

Renningen, im Januar 2024 Florian Weißhardt

Kurzinhalt

Die erfolgreiche Umsetzung von komplexen Serviceroboteranwendungen ist trotz des Fortschritts bei der Auswahl an verfügbaren Hard- und Softwarekomponenten weiterhin eine Herausforderung. Insbesondere im Bereich der Servicerobotik gibt es eine Vielzahl an Anwendungen, die sich durch dynamische und oftmals unbekannte Randbedingungen auszeichnen und daher eine umfangreiche und komplexe Steuerungssoftware benötigen.

Weiterhin zeichnet sich die Servicerobotik durch eine hohe Abhängigkeit der einzelnen Hardware- und Softwarekomponenten aus, sodass die Integration von Komponenten in das Gesamtsystem und insbesondere die Durchführung von einzelnen Komponenten und Systemtests erschwert wird. Bisherige Entwicklungsmethoden, Testframeworks und Werkzeuge zur kontinuierlichen Integration unterstützen den Komponentenentwickler bei der Erstellung und Durchführung von Unittests, jedoch existieren wenige Werkzeuge, die speziell für die Herausforderungen der Servicerobotik umfassende Möglichkeiten zur Unterstützung bei der Systemintegration bieten.

In dieser Arbeit wird die Methode des Test-Driven-Development für die Servicerobotik aufgegriffen und um ein Werkzeug zum Testen und Vergleichen von komponentenbasierter Software erweitert, das sich für Integrationstests und Benchmarkingvergleiche nutzen lässt. Dazu wird ein Test- und Evaluierungsframework, das ***Automated Test Framework*** **(ATF)** konzipiert und umgesetzt.

Das ATF ist ein einfach in bestehende Anwendungen zu integrierendes Framework, das es ermöglicht Servicerobotik-Anwendungen zu testen und verschiedene Varianten miteinander zu vergleichen. Mithilfe von ATF kann der Aufwand für die Erstellung, Durchführung und Auswertung von Tests deutlich gesenkt werden.

Im Rahmen von Anwendungsbeispielen aus verschiedenen Bereichen der Servicerobotik wird der Einsatz von ATF aufgezeigt und dessen Nutzen erfolgreich nachgewiesen und evaluiert. Insbesondere kann in einem Anwendungsbeispiel zu einem Systemtest für die Navigation eine Aufwandsreduktion um den Faktor 10 nachgewiesen werden.

Abstract

The successful implementation of complex service robot applications continues to be a challenge despite the progress made in the selection of available hardware and software components. In the domain of service robotics in particular, there is a multitude of applications that are characterized by dynamic and often unknown boundary conditions and therefore require extensive and complex control software.

Service robotics is also characterized by a high level of dependency on the individual hardware and software components, so that the integration of components into the overall system and, in particular, the implementation of individual components and system tests is made more difficult. Previous development methods, test frameworks and tools for continuous integration support the component developer in the creation and execution of unit tests, but there are few tools that offer comprehensive options to support system integration especially for the challenges of service robotics.

In this thesis, the method of test-driven development for service robotics is taken up and expanded to include a tool for testing and comparing component-based software, which can be used for integration tests and benchmarking comparisons. For this purpose, the **Automated Test Framework (ATF)** is designed and implemented.

The ATF is a framework that can be easily integrated into existing applications, which enables service robotics applications to be tested and different variants to be compared with each other. With the help of ATF, the effort involved in creating, executing and evaluating tests can be significantly reduced.

In the context of application examples from various areas of service robotics, the use of ATF is demonstrated and its benefits proved and evaluated successfully.

In particular, a reduction in effort by a factor of 10 can be demonstrated in an application example of a system test for navigation.

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	**A**pplication **P**rogramming **I**nterface; *Programmierschnittstelle*
ATF	**A**utomated **T**est **F**ramework; *Automatisiertes Test- und Evaluierungsframework*
CI	**C**ontinuous **I**ntegration; *Kontinuierliche Integration*
cob	**C**are-**O**-**b**ot 4®™; *Mobiler Serviceroboter von Mojin Robotics/Fraunhofer IPA*
git	*Software zur verteilten Versionsverwaltung*
Github	*Webbasierter Hostingdienst für git Repositories*
OSS	**O**pen **S**ource **S**oftware; *Quelloffene Software*
ROS	**R**obot **O**perating **S**ystem; *Open Source Roboterbetriebssystem*
ROS Bag	*Dateiformat zum Speichern von ROS Nachrichtendaten*
ROS-I	**ROS-I**ndustrial; *Initiative zur Verbreitung von ROS im industriellen Einsatz*
SW	**S**oft**w**are
TDD	**T**est-**D**riven-**D**evelopment; *Methode zur testgetriebenen Softwareentwicklung*

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

Roboter sind in vielen Lebensbereichen auf dem Vormarsch. Neben ihren traditionellen Einsatzgebieten in der Industrie sind insbesondere Serviceroboter auch außerhalb von Fabrikhallen auf dem Weg zu einer ausgereiften Technologie und kommen in immer mehr Lebensbereichen zum Einsatz.

(a)

(b)

(c)

Abbildung 1.1: *Beispiele für Anwendungsgebiete von Servicerobotern. (a) Mähroboter von Gardena in Garten (Quelle: Gardena), (b) Transportroboter Chuck von 6 River Systems in Warenlager (Quelle: 6 River Systems), (c) Verkaufsassistent Paul von Mojin Robotics in Elektronikfachmarkt (Quelle: Mojin Robotics)*

Jahr	Umsatz in Mrd. US$	Umsatzveränderung im Vergleich zum Vorjahr	verkaufte Einheiten in Tsd.	Veränderung der verkauften Einheiten im Vergleich zum Vorjahr
2018	9,2	+32%	270	+61%
2019	11,2	+32%	173	+32%
2023*			537	+31%

Tabelle 1.1: *Statistik zu jährlichen Umsätzen und verkauften Einheiten von professionellen Service Robotern aus dem World Robotics Bericht 2020 (IFR 2020b). (*=Hochrechnung)*

Das Einsatzgebiet von Servicerobotern erstreckt sich z.B. auf das privaten Umfeld zuhause, den Arbeitsalltag in Büros, den Bereich Gesundheit und Pflege, die öffentlichen Bereiche von Flughäfen, Bahnhöfen, Museen, Hotels und den Einzelhandel. Abbildung 1.1 zeigt Beispiele für verschiedene Anwendungsgebiete von Servicerobotern: im privaten Umfeld zuhause (a), in der Lagerlogistik (b) und im Einzelhandel (c). In den letzten Jahren zeichnet sich laut dem World Robotics Bericht 2020 (IFR 2020b) vor allem der Markt für mobile Serviceroboter mit hohen zweistelligen Wachstumsraten sowohl bei der Anzahl der verkauften Einheiten (2019: +32%), als auch beim Umsatz (2019: +32%) aus. Auch die Wachstumsraten für die nächsten Jahre werden ähnlich hoch prognostiziert, siehe Tabelle 1.1.

Als Serviceroboter werden Roboter bezeichnet, die Dienstleistungen erbringen. Die International Federation of Robotics (IFR) definiert den Begriff Serviceroboter in Anlehnung an die ISO 8373:2012 (ISO 8373:2012) wie folgt:

"*A service robot is a robot that performs useful tasks for humans or equipment excluding industrial automation applications.*"(IFR 2020a)

Serviceroboter weisen weiterhin einen gewissen Grad an Autonomie, von Teleoperation bis hin zu voller Autonomie, auf. Sie habe vielerlei Gestaltungsformen und Anwendungsgebiete, siehe Beispiele in Abbildung 1.2.

Als mobile Serviceroboter werden Serviceroboter bezeichnet, die in der Lage sind sich selbstständig fortzubewegen und ihren Standort wechseln zu können. Dabei kann die Fortbewegung zu Land (Hebert et al. 2012), zu Wasser oder

(a)

(b)

(c)

(d)

Abbildung 1.2: *Beispiele für Serviceroboter. (a) Roboterstaubsauger Roomba von iRobot (Quelle: iRobot), (b) Transportroboter Casero von MLR Systems GmbH (Quelle: Vogel Business Media), (c) Entertainmentroboter Pepper (Quelle: Softbank), (d) Forschungsroboter Care-O-bot 4 (Kittmann et al. 2015; Fraunhofer IPA 2020) (Quelle: Fraunhofer IPA)*

unter Wasser (Yuh 2000) sowie in der Luft (Valavanis 2008) erfolgen. Diese mobilen Serviceroboter agieren in diversen natürlichen, d.h. nicht speziell für Roboter präparierten, unstrukturierten Umgebungen und verwenden deshalb eine Vielzahl an verschiedenen Sensoren, wie z.B. Laserscanner, 2D- und 3D-Kameras. Abbildung 1.3 zeigt typische Sensoren, mit denen Serviceroboter ihre Umgebung wahrnehmen können.

(a) (b) (c)

Abbildung 1.3: *Typische Sensoren für die Erfassung der Einsatzumgebung von Servicerobotern. (a) Laserscanner (Quelle: Sick AG), (b) 2D Farbkamera (Quelle: Stemmer Imaging), (c) 3D Kamera (Quelle: Intel)*

Die ISO 8373:2012 unterteilt Serviceroboter weiterhin auf in *personal service robots* und *professional service robots*. *Personal service robots* kommen in nicht kommerziellen Aufgaben im häuslichen Umfeld zum Einsatz, wie z.B. Haushalts-, Staubsauger- (Jones 2006) und Rasenmähroboter (Hicks & Hall 2000), Entertainmentroboter (Schraft et al. 2001) oder persönliche Mobilitätshilfen (Llarena & Rojas 2016). *Professional service robots* hingegen werden für kommerzielle Aufgaben eingesetzt, z.B. Roboter für die Landwirtschaft (Billingsley et al. 2008), Roboter für professionelle Reinigung (Elkmann et al. 2009; Endres et al. 1998), Inspektions- und Wartungsroboter (Bengel et al. 2009), Roboter als Verkaufsassistenten (Schmitt et al. 2017) oder Transportroboter in öffentlichen Umgebungen wie Krankenhäusern, Pflegeeinrichtungen oder Bürogebäuden (Graf et al. 2012), und vor allem auch in der Warenlagerlogistik (Wurman et al. 2008; Bartels & Beetz 2019).

1.1 Motivation

Immer mehr Serviceroboter erobern immer breiter gefächerte Anwendungen. Aus kommerzieller Sicht wird meist ein autonomer Mehrschicht- oder 24/7-Betrieb angestrebt, sodass ein Roboter ohne direkte Kontrolle oder Fernsteuerung durch einen Menschen auskommt. Die notwendigen Technologien zur Umsetzung der Anwendungen u.a. aus Abbildung 1.1 entwickeln sich immer umfangreicher. Insbesondere die für den Betrieb der Roboter notwendige Software wird immer komplexer. Dies liegt u.a. daran, dass die Roboter in Umgebungen zum Einsatz kommen, die nicht primär für den Betrieb von autonomen Systemen ausgelegt sind, sondern sich nach den Bedürfnissen von Menschen ausrichten. In diesen Umgebungen treffen Serviceroboter auf sich verändernde und unstrukturierte Umgebungsbedingungen, wie z.B. dynamische Hindernisse für die Navigation oder variierende Positionen von Objekten bei der Manipulation, sodass diese ihre Handlungen immer wieder neu anpassen müssen. Dies geschieht typischerweise mithilfe von auf den Servicerobotern mitgeführten Sensoren wie sie in Abbildung 1.3 gezeigt sind. Die Steuerungssoftware des Serviceroboters muss die Umgebung durch die Sensoren wahrnehmen und dann passende Aktionen daraus ableiten wodurch insbesondere weitreichende Fehlerbehandlungen und alternative Lösungsstrategien notwendig werden.

Aufgrund beschränkter Entwicklungszeit und -ressourcen ist es dabei für einzelne Softwareentwickler oder einzelne Entwicklerteams oft nicht möglich alle Funktionalitäten von Grund auf neu zu implementieren. Deshalb wird entweder auf kommerzielle (typischerweise Closed Source) oder frei verfügbare (typischerweise Open Source) Komponenten zurückgegriffen, sodass der eigene Implementierungsanteil handhabbar bleibt. Die eigenen Implementierungen fokussieren sich dabei auf technologische Lücken, für die keine fertigen Komponenten verfügbar sind oder auf eigenentwickelte Komponenten, mit denen sich die Anwendung gegenüber anderen konkurrierenden Anwendungen differenzieren kann.

Kommerzielle Voraussetzungen für den Einsatz von Servicerobotern

Die Verbreitung von Serviceroboteranwendungen scheitert oft nicht nur an der technischen Umsetzung oder der Akzeptanz, sondern an dem Gesamtaufwand, der sich als Preis bzw. dem Preis-/ Leistungsverhältnis niederschlägt. Die vom Fraunhofer IPA durchgeführte Studie zur Wirtschftlichkeitsanalyse von Servicerobotik-Anwendungen (Hägele, Martin et al. 2011) betrachtet für die Wirtschaftlichkeit die Lebenszykluskosten für eine Servicerobotik-Anwendung. Diese setzt sich dabei sowohl aus den Entwicklungskosten für Hard- und Software, den Stückkosten für Material und Montage, den Projektierungs- und Einrichtungskosten sowie den Kosten für den Betrieb und Pflege der Anwendung zusammen. Ziel vieler Anbieter ist es daher die Lebenszykluskosten insgesamt zu senken, um einen wirtschaftlichen Einsatz zu realisieren.

Die Stück- und Montagekosten lassen sich über entsprechende Mengen, andere Materialien und Produktionsverfahren reduzieren. Auch die fortschreitende Kostensenkung und Miniaturisierung von modernen Sensor- und Rechnersystemen sowie Netzwerktechnik machen sich hier bemerkbar. Es gibt auch immer mehr integrierte Produkte, die mehrere Funktionen in einem Gerät bzw. Bauteil zum gleichen Preis vereinen und dadurch die Anzahl der benötigten Hardware-Komponenten reduzieren, was ebenfalls zur einer Kostensenkung führt. Ein Beispiel aus der Computertechnik hierfür sind in CPUs integrierte Grafikkarten, sodass viele Computer heute ohne dedizierte Grafikkarten auskommen.

Trotz des technologischen Fortschritts in Bezug auf Wahrnehmung und kognitive Systeme können die Entwicklungskosten durch die gesteigerte Komplexität und gestiegene Anforderungen an den robusten und störungsfreien Betrieb sowie eine hohe Verfügbarkeit der Serviceroboter nicht wesentlich gesenkt werden. Viele Funktionalitäten können als Hardware- oder Softwareproblem gelöst werden. Oftmals ist eine Hardwarelösung einfacher, aber teurer. Eine äquivalente Softwarelösung erfordert mehr Softwareentwicklungsaufwand. Für eine Bildverarbeitungsaufgabe kann z.B. statt einer hochwertigen, aber auch hochpreisigen, Industrie-Kamera eine günstige Smartphone-Kamera eingesetzt werden falls

entsprechend aufwändigere Kalibrier- und Bildauswerteverfahren zum Einsatz kommen. Dies bringt jedoch eine höhere Softwarekomplexität mit sich. Eine erhöhte Softwarekomplexität wiederum bedeutet eine erhöhte Anfälligkeit gegenüber unvorhergesehenen Störungen (Fernández-Madrigal et al. 2009). Der steigenden Störanfälligkeit muss mit höheren Kosten bei Betrieb und Pflege Rechnung getragen werden, was sich besonders im öffentlichen und industriellen Bereich negativ auf die Wirtschaftlichkeit der Systeme auswirkt. Der Forschungsbericht zum deutschen Innovationssystem No. 11-2016 über Automatisierung und Robotik-Systeme der Expertenkommission Forschung und Innovation belegt:

"*Effizientes Software-Engineering ist entscheidend, damit die Entwicklungskosten für Serviceroboter Anwendungen beherrschbar sind.*"
(Beckert 2016)

Um den Betrieb in diesen Umgebungen zu realisieren kommt eine Vielzahl an softwarebasierten Technologien aus den Bereichen Hardwareansteuerung, Sensordatenverarbeitung, Navigation, Manipulation, Mensch-Maschine-Interaktion, etc. zum Einsatz. Letztere nutzt beispielsweise wiederum Softwaretechnologien wie Spracherkennung und Künstliche Intelligenz. Für viele dieser Technologien sind fertige Funktionalitäten verfügbar, aus denen die Robotik-Anwendung zusammengesetzt werden kann. Hierbei gibt es sowohl kommerziell verfügbare Softwarekomponenten, als auch frei verfügbare Open Source Softwarekomponenten. Die Standardisierung bezüglich Implementierung, Schnittstellen, Kommunikation und Integration von Funktionalitäten ist jedoch noch gering.

ROS als Bindeglied für die Integration

Ein weit verbreitetes Framework als Bindeglied für die Integration verschiedener Funktionalitäten ist das Robot Operating System ROS (Open Robotics 2020b; Open Robotics 2020k), welches ein flexibles Framework für Robotiksoftware darstellt (siehe Kapitel 2.1.4). ROS besteht aus einer Sammlung von Werkzeugen, Bibliotheken und Schnittstellendefinitionen mit dem Ziel das Erstellen von komplexen, robusten, modularen und hardwareübergreifenden Robotersteuerungen

zu vereinfachen. ROS deckt sowohl das Paketmanagement, als auch die Middlewarekommunikation ab (Quigley et al. 2009) und treibt die Weiterentwicklung in einer weltweit agierenden Gemeinschaft voran.

"*Zwei Drittel aller Robotik-Startups nutzten mittlerweile ROS.*"
(Quelle: Christoph Hellmann Santos, Fraunhofer IPA, Manager des ROS-Industrial Konsortiums Europa (Heise 2020))

Abbildung 1.4 zeigt die weltweite Verbreitung von ROS. ROS ist dabei sowohl in der wissenschaftlichen Forschung sehr präsent, als auch im industriellen Einsatz in Verwendung (ROS-Industrial Consortium 2020b). Die Verbreitung in der Forschung zeigt sich u.a. daran, dass sich die Anzahl der Veröffentlichungen, die Quigley et al. (2009) zitieren, auf 9128 summiert (Quelle: Google Schoolar, Stand: 9.12.2021).

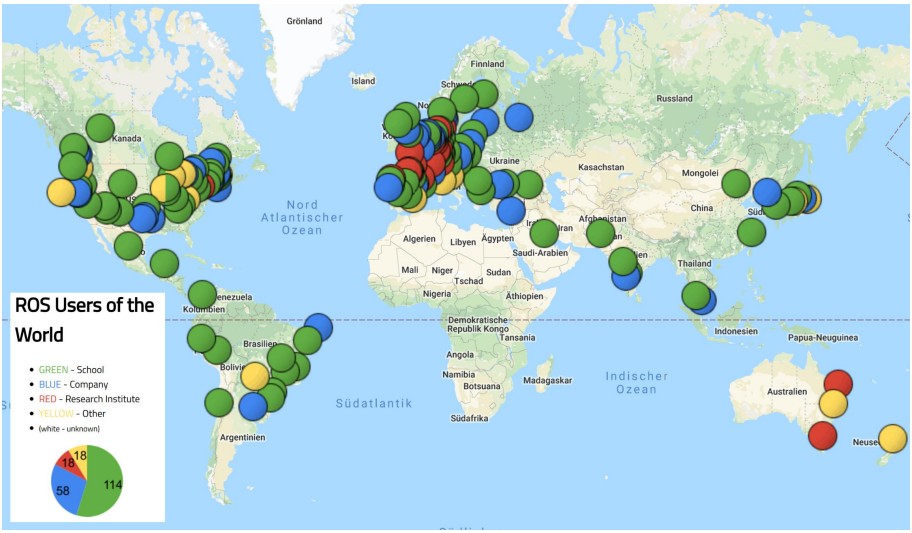


Abbildung 1.4: *Weltweite Mitglieder der ROS Gemeinschaft. (Quelle: David Lu, Stand: 01/2020)*

Hindernisse für die Verbreitung von kommerziellen Servicerobotik-Anwendungen

Ein wesentliches Hindernis für die weitere kommerzielle Verbreitung von Servicerobotern ist deren Zuverlässigkeit und Robustheit in nicht kontrollierten Einsatzbedingungen wie sie durch unstrukturierte Umgebungen und ungeschulte Nutzer vermehrt auftreten. Um dieser Herausforderung zu begegnen, muss die Steuerung von Roboteranwendungen in der Lage sein, unvollständige oder fehlerhafte Wahrnehmung und unvermeidbare Fehler ordnungsgemäß zu beheben. D.h. es gibt während des Entwicklungsprozesses noch unbekannte, jedoch unvermeidbare, Situationen und Umgebungsbedingungen, mit denen der Roboter später konfrontiert wird. Dabei ist es ungewiss, ob der Roboter sich auf diese unbekannten Situationen und verändernde Umgebungsbedingungen einstellen kann.

Verhalten des Gesamtsystems

Das Verhalten des Roboter setzt sich dabei aus dem Verhalten der verwendeten Einzelkomponenten sowie deren Integration in ein Gesamtsystem, in dem diese wechselwirken, zusammen. Neben Fehlern in den Einzelkomponenten kann auch deren Integration in das Gesamtsystem Fehler mit sich bringen. Die Konfigurationen der Einzelkomponenten (z.B. Pfadplaner, Lokalisierung, Fahrwerks- und Odometrieregler) ergeben die Konfiguration des Gesamtsystems und beeinflussen ebenfalls das Verhalten des Gesamtsystems. Voraussetzung für eine erfolgreiche Integration sind daher funktionsfähige Einzelkomponenten sowie eine funktionsfähige Komposition und Kommunikation anhand der passenden Schnittstellen der Einzelkomponenten. Selbst bei Verwendung von Komponenten aus dem ROS Umfeld gibt es viele heterogene und nicht kompatible Schnittstellen. Auch die Implementierungsqualität und das Komponentenverhalten ist breit gefächert, sodass die Komponenten, ohne genaue Kenntnisse der internen Implementierung, oft schwer zu beurteilen sind.

Systemtests zur Verifizierung und Validierung

Verifizierung und Validierung sind jeweils eigenständige Verfahren eines Qualitätsmanagementprozesses, die verwendet werden um zu überprüfen, ob ein Produkt, Service oder System den Anforderungen, der Spezifikation und dem beabsichtigten Zweck entspricht (Hojo 2004). Dabei ist die Aufgabe der Verifizierung das Überprüfen ob die im Pflichtenheft festgelegten Anforderungen implementiert sind. Verifizierung ist demnach kein Nachweis dafür, ob ein Produkt fehlerfrei ist, da nur Fehler die nach der Spezifikation entstanden sind gefunden werden können. Die Validierung hingegen prüft mit objektiven Mitteln das Produkt in seiner Einsatzumgebung auf die Erreichung von zuvor festgelegten Nutzungszielen. Validierung beantwortet somit die Frage „*Bauen wir das Richtige?*“, wohingegen Verifizierung „*Bauen wir es richtig?*“ beantwortet.

Eine Möglichkeit festzustellen, wie der Roboter auf verschiedene Situationen und Einsatzumgebungen reagiert ist, das Gesamtsystem aus Roboter, Umgebung und Situation möglichst ausgiebig und breit zu Testen. Ein Überblick über verschiedene Softwaretests wird in Kapitel 2.2 vorgestellt. Testen ist jedoch nicht immer uneingeschränkt möglich, sodass dies zu folgenden Testbedingungen führen kann:

- Tests können teilweise nur sinnvoll und wirtschaftlich in **Laborumgebungen** statt realen Umgebungen durchgeführt werden.
- Testen mit **realer Hardware** ist **aufwändig und langwierig**.
- Manuelles Bewerten eines Testdurchlaufs bzw. manuelles Auswerten von Tests ist anfällig für eine **subjektive Bewertung der Testergebnisse**, falls keine objektiv messbaren Metriken zugrunde gelegt werden.
- Es gibt keine harten Testkriterien, wann ein Testdurchlauf zufriedenstellend ist, da viele verschiedene Faktoren (z.B. zeitliche Ablauf, Roboterverhalten, Interaktion mit der Umgebung) zur Bewertung des Testergebnisses notwendig sein können und diese wesentlich von der Beschaffenheit der Umgebung und der genauer Situation abhängen.

- Die Testabdeckung ist gering, da eine Vielzahl an Konfigurationen, Umgebungen und Situationen nicht getestet wird.
- Tests können durch Änderungen in der Umgebung (z.B. bewegte Objekte oder veränderliche Lichtverhältnisse), Ungenauigkeiten in der Wahrnehmung (z.B. Sensorrauschen) oder dem Einsatz von probabilistischen Verfahren (z.B. Pfadplaner) andere Ergebnisse liefern.
- Tests können durch Änderungen der Konfiguration der Anwendung oder Änderung der verwendeten Komponenten andere Ergebnisse liefern.
- Tests können für große Parameterräume oft nur für eine eingeschränkte Kombination an Parametern durchgeführt werden.
- Tests können statistischen Schwankungen unterliegen (z.B. probabilistische Pfadplaner), sodass der gleiche Tests mit denselben Startbedingungen und unter den selben Randbedingungen zu unterschiedlichen Ergebnissen führt.

Insbesondere wegen aufwändiger und langwieriger Tests mit realer Hardware und der subjektiven Bewertung von Testergebnissen beim manuellen Testen ist objektives Testen notwendig. Für ein aussagekräftiges, objektives Testergebnis ist es notwendig, dass Tests nach jeder Änderung mit den gleichen Startbedingungen mehrfach durchgeführt werden, sodass statistische Schwankungen erfasst werden können und die Tests anhand von objektiv messbaren Metriken bewertet werden.

1.2 Problemstellung

Ein effizientes Software-Engineering ist entscheidend dafür, eine Servicerobotik-Anwendung wirtschaftlich zu entwickeln und zu betreiben (Beckert 2016). Ein wesentlicher Faktor hierbei sind Komponenten- und Systemtests zur Verifizierung und Validierung des Gesamtsystems bereits während der Entwicklungs- oder Inbetriebnahmephase, sodass durch die Tests entdeckte Fehler möglichst früh und damit kostengünstig behoben werden können. Wie in Kapitel 1.1 dargestellt, müssen Entwickler aus einer Vielzahl an verfügbaren oder zuvor

selbst entwickelten Komponenten sowie deren Konfigurationsvarianten auswählen, um diese zusammen mit eigenen, neu entwickelten Komponenten in ein Gesamtsystem integrieren zu können. Um Sicherstellen zu können, dass die entwickelte Anwendung den Anforderungen in allen unstrukturierten und dynamischen Einsatzumgebungen entspricht, wäre es nötig, die Anwendung in allen Einsatzumgebungen zu validieren. Dies führt in Realität zu vier wesentlichen Problemen:

1. **Geringe Testabdeckung:** Nur eine geringe Anzahl und wenig umfangreichen Tests werden durchgeführt.

2. **Nicht reproduzierbare Testbedingungen:** Testumgebungen sind nicht nachstellbar, nicht mit vertretbarem Aufwand herstellbar oder gar nicht bekannt.

3. **Zeit-, orts- und personenabhängige Testauswertung:** Die Auswertung von Tests ist nicht jederzeit, überall und durch jeden reproduzierbar.

4. **Zu spätes Testen im Entwicklungsprozess:** Bei limitierten Entwicklungskapazitäten werden Tests erst nach der Entwicklungsphase oder nach der Inbetriebnahme der Anwendung durchgeführt.

Umfangreiches Testen fällt meist aufgrund von mangelnden Ressourcen oder zeitlichen Engpässen weg oder der Testumfang wird auf wenige Testfälle reduziert (Fernández-Madrigal et al. 2009). Weiterhin ist manuelles Testen zeitaufwendig und fehleranfällig gegenüber subjektiven Einschätzungen der Person, die den Test durchführt (Santos et al. 2018). Daraus ergeben sich im späteren Betrieb der Anwendung Fehler, die jedoch im Vorfeld durch umfangreicheres objektives Testen hätten vermieden werden können. Die daraus resultierende Forschungsfrage für diese Arbeit ist:

Wie kann während der Entwicklungsphase einer Servicerobotik-Anwendung der Entwicklungs- und Testaufwand reduziert, die Testabdeckung gesteigert sowie die Robustheit des Gesamtsystems im späteren Betrieb erhöht werden?

Ein Lösungsansatz ist das in dieser Arbeit vorgestellte Automatisierte Test- und Evaluierungsframework ATF.

1.3 Zielsetzung und Lösungsansatz

Ziel dieser Arbeit ist es, Servicerobotik-Lösungen robuster und wirtschaftlicher für den Einsatz in realen Umgebungen zu machen, indem deren Entwicklung in einem heterogenen und verteilten Entwicklungsumfeld schneller, kostengünstiger und effizienter wird. Dies wird durch den Einsatz eines automatisierten Test- und Evaluierungsframeworks, welches den Aufwand zur Erstellung von Tests reduziert, die Testabdeckung erhöht und die Testautomatisierung steigert, untersucht.

Diese Arbeit stellt das automatisierte Test- und Evaluierungsframework ATF vor, das dabei helfen soll, die Testabdeckung während der Entwicklungsphase zu erhöhen und den Aufwand zur Erstellung und wiederholten Durchführung von Tests durch die automatisierte Durchführung und Auswertung von Tests zu senken. Durch die Formulierung und den Einsatz von objektiven Kriterien sollen subjektive Verzerrungen der Testergebnisse vermieden und Testergebnisse miteinander verglichen werden können. Das übergeordnete Ziel ist die schnelle und kostengünstige Entwicklung und der robuste Betrieb des Gesamtsystems mit einer hohen Verfügbarkeit.

Das ATF deckt dabei zwei Bereiche ab: Zum Einen kann das ATF als Werkzeug zum entwicklungsbegleitenden Testen (Komponenten- und Systemtests) sowie zum Anderen als Werkzeug für den Vergleich unterschiedlicher Komponenten und Konfigurationen (engl. *Benchmarking*) genutzt werden. Das ATF stellt hierbei ein auf ROS basierendes Framework dar, mit dessen Hilfe Entwickler Tests sowie zu evaluierende Metriken für Anwendungen definieren und ausführen können. Tests werden dabei in vier Phasen zerlegt: (1) In einer sogenannten Testgenerierungsphase werden aus der gegebenen Testkonfiguration ausführbare Tests generiert. (2) Die Tests werden anschließend in einer Aufnahmephase zusammen mit der Anwendung ausgeführt und nehmen entsprechend der defi-

nierten Metriken Daten auf. (3) Nach der Datenaufnahme aller Tests schließt sich eine Analysephase an, in der die Metriken ausgewertet und die Testergebnisse abgeleitet werden. (4) In einer Visualisierungsphase können die generierten Testergebnisse visualisiert, gegenübergestellt und verglichen werden.

Eine Anwendung kann dabei in mehrere Testblöcke aufgeteilt werden, die separat mithilfe von Metriken und Groundtruthdaten ausgewertet werden. ATF enthält eine Auswahl an servicerobotikspezifischen Metriken, die für Anwendungsfälle aus den Bereichen Navigation, Perzeption und Manipulation genutzt werden kann. Das ATF ist ein erweiterbares Framework, welches vorsieht, dass die bestehenden Metriken um weitere anwendungsspezifische Metriken ergänzt werden können. Die Auswahl an durchzuführenden Tests und anzuwendenden Metriken wird in Testsuites definiert, um bei Bedarf die Gesamtzahl an Tests eingrenzen zu können. Weiterhin bietet das ATF Möglichkeiten zur Integration in bestehende automatisierte Infrastrukturen für Tests und kontinuierliche Integration (engl. *Continuous Integration* CI) an und kann somit als CI-Service genutzt werden.

ATF stellt einen Lösungsansatz für die in Kapitel 1.2 aufgelisteten Probleme beim Testen dar. In dieser Arbeit werden die folgenden Hypothesen validiert:

1. **Erhöhte Testabdeckung:** Durch ATF wird der Aufwand zur Umsetzung, Durchführung und Auswertung von Komponenten-, Integrations- und Systemtests verringert. Dies reduziert die Hürde zur Erstellung von Tests und ermöglicht durch die bereitgestellten Metriken, die automatisierte Durchführung und Auswertung sowie die einfache Konfiguration von Testsuites eine hohe Testanzahl mit diversen Konfigurationen und Randbedingungen und damit eine erhöhte Testabdeckung.

2. **Reproduzierbare Testbedingungen:** Durch die Unterstützung von *Simulation-in-the-loop* Tests können reproduzierbare Testbedingungen hergestellt werden. ATF ermöglicht es durch *Hardware-in-the-loop* Tests zudem die Übertragbarkeit von aus Simulationen gewonnenen Testergebnissen mit real durchgeführten Tests zu bewerten.

3. **Zeit-, orts- und personenunabhängige Testauswertung:** Die automatisierte Auswertung anhand von objektiven Metriken ermöglicht es ATF die Ergebnisse von Tests jederzeit, überall und durch jeden reproduzierbar zu machen.

4. **Frühzeitiges Testen:** Durch den geringen Aufwand zur Umsetzung, Durchführung und Auswertung von Tests wird durch ATF ein entwicklungsbegleitendes Testen (CI) ermöglicht. Mithilfe von ATF lassen sich testgetriebene Entwicklungsmethoden umsetzen.

Anmerkung: Die Konzepte und Ergebnisse dieser Arbeit sind grundsätzlich auch ohne ROS anwendbar und auf andere Robotikframeworks übertragbar. Weiterhin ist ATF auf das Feld der Industrierobotik anwendbar, wobei das vorgestellte Framework ATF sein volles Anwendungsspektrum vor allem bei komplexen, sensor-basierten Robotern zeigen kann. Da dies primär im Bereich der Servicerobotik anzutreffen ist, liegt der Fokus der vorliegenden Arbeit auf dem Gebiet der Servicerobotik. Im weiteren Verlauf dieser Arbeit wird der Begriff Roboter daher stellvertretend für Serviceroboter verwendet.

1.4 Gliederung

Der weitere Aufbau dieser Arbeit glieder sich wie folgt in sechs Kapitel. Zunächst werden in Kapitel 2 Grundlagen der Softwareentwicklung mit Bezug auf die Servicerobotik erläutert, verschiedene Aspekte von Softwarequalität und Qualitätssicherung vorgestellt sowie der Stand der Technik zum Thema Softwaretests analysiert. Weiterhin werden Metriken zur Bewertung von Servicerobotik-Anwendungen definiert und Benchmarkingverfahren erörtert. In Kapitel 3 werden Anforderungen an ein automatisiertes Test- und Evaluierungsframework abgeleitet sowie ein Konzept zur Umsetzung des ATF Frameworks erstellt. Die Umsetzung des Konzepts folgt in Kapitel 4, welches sich in vier Schritte gliedert: Schritt 1: Definition und Generierung von Tests, Schritt 2: Ausführen der Anwendung mit Aufnahme von Testdaten, Schritt 3: Analyse der Testdaten und

Schritt 4: Visualisierung und Vergleich der Testergebnisse. Anschließend wird vorgestellt, wie das ATF in einen automatisierten CI-Ablauf eingebettet werden kann. In Kapitel 5 wird das ATF in Bezug auf die in Kapitel 3 aufgestellten Anforderungen anhand von mehreren Anwendungsbeispielen aus den Bereichen Navigation, Perzeption und Manipulation evaluiert. Schließlich wird die Arbeit in Kapitel 6 zusammengefasst und ein Ausblick auf weiterführende Arbeiten gegeben.

Anmerkung: Im weiteren Verlauf dieser Arbeit werden die Ziele und Anforderungen aus den einzelnen Unterkapiteln in Kapitel 2 abgeleitet und jeweils am Ende des Unterkapitels aufgelistet und durch einen Doppelpfeil (⇒) gekennzeichnet. In Kapitel 3 werden die Ziele und Anforderungen in Tabelle 3.1 und Tabelle 3.2 zusammengefasst und nummeriert.

Beispiele:

⇒ **Ziel Z-L-0:** Beispiel für Ziel der Servicerobotik-Lösung

⇒ **Ziel Z-EP-0:** Beispiel für Ziel des Entwicklungsprozess

⇒ **Anforderung AF-0:** Beispiel für Anforderung an ATF

2 Grundlagen und Ausgangssituation

In diesem Kapitel werden Grundlagen der Softwareentwicklung mit Bezug auf die Servicerobotik erläutert. Es werden zunächst gebräuchliche Vorgehensmodelle und Entwicklungsmethoden in Kapitel 2.1 vorgestellt und die besonderen Rahmenbedingungen bei der Softwareentwicklung in der Servicerobotik erörtert. Hierbei wird insbesondere der Einsatz von ROS und die verteilte Entwicklung in der Open Source Gemeinschaft beleuchtet. Anschließend werden in Kapitel 2.2 verschiedene Aspekte von Softwarequalität dargelegt und in Kapitel 2.3 Qualitätssicherungsmethoden zum Messen und Steigern von Softwarequalität vorgestellt. Weiterhin wird der Stand der Technik zum Thema Softwaretests analysiert und vorhandene Testframeworks für die Servicerobotik diskutiert. In Kapitel 2.4 werden Metriken zur Bewertung von Servicerobotik-Anwendungen definiert und anschließend in Kapitel 2.5 Benchmarkingverfahren vorgestellt.

2.1 Softwareentwicklung in der Servicerobotik

Im folgenden Kapitel wird der Stand der Technik für Softwareentwicklung in der Servicerobotik erörtert. In Kapitel 2.1.1 Vorgehensmodelle und Entwicklungsmethoden vorgestellt und deren Verwendung in der Servicerobotik erläutert. Weiter wird in Kapitel 2.1.2 auf den für die Servicerobotik wichtigen Prozess der Systemintegration eingegangen und anschließend zwei Ansätze vorgestellt, mit denen die Systemintegration in der Servicerobotik vereinfacht werden kann. Dies ist zum Einen der Einsatz von Simulationsumgebungen (siehe Kapitel 2.1.3) und zum Anderen der Einsatz von Robotik-Frameworks (siehe Kapitel 2.1.4).

2.1.1 Vorgehensmodelle und Entwicklungsmethoden

In der Softwareentwicklung spricht man von Vorgehensmodellen, die die Erstellung des Softwareprodukts in modellhafte und abstrahierte Beschreibungen von Vorgehensweisen, Richtlinien, Empfehlungen oder Prozessen abbilden. Hierbei wird der Gesamtprozess in verschiedene Phasen unterteilt, die den komplexen Prozess der Softwareentwicklung in überschaubare, zeitlich und inhaltlich abgegrenzte Schritte gliedert. Es werden dabei zwischen sequenziellen, iterativen und inkrementellen sowie agilen Vorgehensmodellen unterschieden. Im Folgenden wird die Klassifizierung nach Reiser (2014) aufgegriffen und es werden einige Vorgehensmodelle, die in der Servicerobotik zu Anwendung kommen, beschrieben.

Sequentielle Phasenmodelle

Unter sequentiellen Phasenmodellen versteht man Modelle, bei denen die einzelnen Prozessschritte sequentiell aufeinander erfolgen und es keine Verzweigungen, Iterationen oder Rücksprünge gibt. Zu den sequentiellen Phasenmodellen werden u.a. das Wasserfallmodell und V-Modell gezählt.

Wasserfallmodell Das Wasserfallmodell ist eine Aufteilung der Projektaktivitäten in sequentiell aufeinander folgende Phasen, wobei jede Phase von den Ergebnissen der vorherigen Phase abhängt. Das Wasserfallmodell findet in diversen Ingenieursdisziplinen Anwendung. In der Softwareentwicklung hingegen gehört es zu den weniger iterativen und weniger flexiblen Ansätzen, da die Abfolge der Phasen z.B. Anforderungsanalyse, Entwurf, Implementierung, Verifikation und Betrieb weitgehend in eine Richtung fließt, wie Wasser über einen Wasserfall (Royce 1987).

V-Modell Das V-Modell (Bröhl 1993) basiert wie das Wasserfallmodell auf sequentiellen Entwicklungsphasen und ergänzt dieses um Phasen zur Qualitätssicherung, indem den einzelnen Entwicklungsphasen Testphasen zur Verifikation

und Validierung zugeordnet werden. Der Name leitet sich von der grafischen Darstellung der Phasen des Entwicklungszyklus ab, siehe Abbildung 2.1.

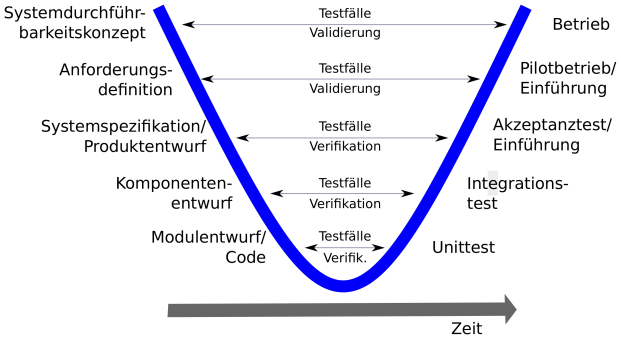


Abbildung 2.1: *Das V-Modell. (Quelle: Reiser 2014)*

Das ursprüngliche V-Modell wurde bereits Ende der 1970er Jahre vorgestellt, mittlerweile existieren Abwandlungen oder Ergänzungen wie z.B. das V-Modell XT (Broy & Rausch 2005), das das V-Modell modularisiert und um Ansätze z.B. aus der agilen Entwicklung ergänzt.

In der Servicerobotik sind sequentielle Modelle wie das Wasserfall oder das ursprüngliche V-Modell durch ihre starre Struktur, die nur beschränkt Rückkopplungen über mehrere Phasen vorsieht, für komplexere Softwareanwendungen ungeeignet, da unentdeckte Fehler insbesondere aus den frühen Phasen nicht korrigiert werden können. Auch für Anwendungen, bei denen zu Beginn noch nicht alle Anforderungen hinreichend definiert sind, sind sequentielle Phasenmodelle nur bedingt geeignet.

Agile Entwicklungsmodelle

Agile Softwareentwicklung umfasst verschiedene Ansätze für die Softwareentwicklung, die auf dem Agilen Manifest von 2001 (Beck et al. 2001) basieren. Ziel von agilen Softwareentwicklungsmodellen ist es, die Transparenz und Flexibilität zu erhöhen, indem Teilprozesse möglichst einfach und somit beweglich (=agil) gehalten werden. Agile Softwareentwicklung beschreibt die Zusammenarbeit von

sich selbst organisierenden und funktionsübergreifenden Teams und deren Kunden und fördert eine adaptive Planung, evolutionäre Entwicklung, frühzeitige Auslieferung von Teilinkrementen und kontinuierliche Verbesserung durch schnelle und flexible Reaktion auf Veränderungen. Die Risiken im Entwicklungsprozess sollen damit minimiert werden.

Entwicklungsmodelle und -methoden von agiler Softwareentwicklung sind z.B. Scrum (Schwaber & Sutherland 2011), Extreme Programming (Beck & Andres 2004) und Test-Driven-Development (TDD) (Beck 2003). Ein speziell auf die Bedürfnisse der Robotik angepasstes Vorgehensmodell ist das Modell *Distributed Development and Test* (DDT) (Reiser 2014).

Test-Driven-Development TDD Testgetriebene Entwicklung (engl. *Test-Driven-Development* TDD) wird den agilen Entwicklungsmethoden zugeordnet. TDD ist eine Methode, bei der im Entwicklungsprozess zuerst Softwaretests erstellt werden, *bevor* die Implementierung von Komponenten beginnt. D.h. die Anforderungen werden in spezifische Testfälle umgewandelt, die vorerst fehlschlagen. Danach wird Zug um Zug die Software so verbessert, dass die Tests erfolgreich bestanden werden. Dies steht im Gegensatz zur Softwareentwicklung, mit der Software hinzugefügt werden kann, die nachweislich nicht den Anforderungen entspricht.

Kent Beck beschreibt den TDD-Prozess in Beck (2003) wie in Abbildung 2.2 gezeigt folgendermaßen:

1. **Füge einen Test hinzu:** Jedes neue Softwarefunktion beginnt mit dem Schreiben eines Tests. Ein Test sollte dabei möglichst präzise sein. Um einen Test schreiben zu können, muss der Entwickler die Spezifikation und die Anforderungen der Softwarefunktion genau kennen. Der Entwickler kann dies z.B. durch Use Cases und User Stories erreichen und kann den Test in einem für die Softwareumgebung geeigneten Testframework schreiben. Diese Vorgehensweise wird sowohl für neue Softwarefeatures, als auch für die Verbesserung bestehender Softwarefeatures angewendet werden.

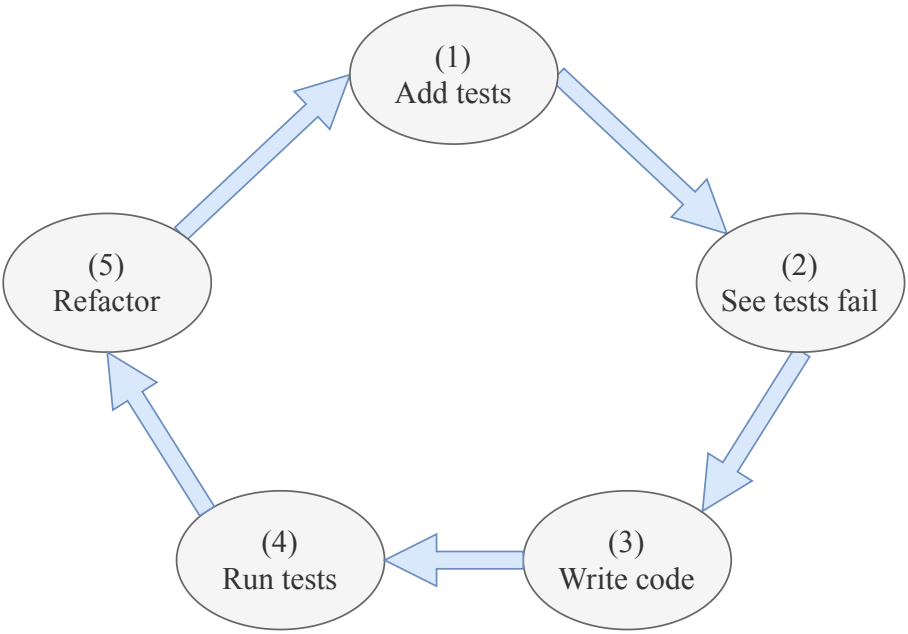


Abbildung 2.2: *Der Test-Driven-Development-Prozess. (1) Test hinzufügen, (2) Tests ausführen und überprüfen, ob neuer Test fehlschlägt, (3) fehlgeschlagene Tests beheben, (4) Tests ausführen und überprüfen, ob alle Tests erfolgreich bestehen und (5) Code Aufräumen.*

2. **Führe alle Tests aus und überprüfe, ob der neue Test fehlschlägt:** Der neue Test wird vom vorhandenen Programmcode erst einmal nicht erfüllt, muss also fehlschlagen bis neuer Programmcode hinzugefügt wird. Das Fehlschlagen des neuen Tests schließt aus, dass der hinzugefügte Test fehlerhaft ist und immer bestanden wird.

3. **Behebe fehlgeschlagene Tests:** Der Entwickler ändert den Programmcode mit möglichst wenig Aufwand und Codeduplizierung ab, bis er alle Tests besteht. Die Änderungen verfolgen den einzigen Zweck, den Test zu bestehen. Der Programmierer darf keinen Code schreiben, der über die Funktionalität hinausgeht, die der Test überprüft.

4. **Führe alle Tests aus und überprüfe, dass alle Tests erfolgreich bestehen:** Dieser Schritt beinhaltet das Ausführen aller Tests, sodass sichergestellt ist, dass der neue Code die Testanforderungen erfüllt ohne vorhandene Tests fehlschlagen zu lassen. Dies vermeidet, dass vorhandene Features nicht mehr funktionieren oder sich verschlechtern.

5. **Räume den Code auf (refactoring):** Die wachsende Codebasis muss während der testgetriebenen Entwicklung regelmäßig bereinigt werden. Neuer Code kann verschoben und Codeduplikate aufgelöst werden. Objekt-, Klassen-, Modul-, Variablen- und Methodennamen können vereinheitlicht und konsistent benannt werden. Die Lese- und Wartbarkeit der Codebasis kann erhöht und Gestaltungsrichtlinien eingehalten werden. Durch die mehrfache Wiederholung der Testdurchläufe während der Refactoringphase kann der Entwickler sicher sein, dass der Prozess keine bestehenden Funktionen verändert.

Dieser Prozess wiederholt sich in jeder Iteration, wobei die Änderungen in jedem Schritt klein sein sollten. Wenn neuer Code einen neuen Test nicht schnell erfüllt oder andere Tests unerwartet fehlschlagen, sollte der Programmierer den Test zurücksetzen, anstatt übermäßig den Fehler zu suchen. Versionskontrolle und Kontinuierliche Integration (engl. *Continuous Integration* CI) hilft durch die Bereitstellung von umkehrbaren Kontrollpunkten. Dieser Prozess wird solange

wiederholt bis bekannte Fehler bereinigt sind und der Code die gewünschte Funktionalität erfüllt und keine weiteren Tests implementiert werden können, die noch scheitern könnten.

Es gibt Studien, die zeigen, dass Entwickler durch die Verwendung von TDD produktiver waren (Erdogmus et al. 2005; Madeyski 2009; Rafique & Mišić 2012) und dass, obwohl bei TDD durch das Schreiben von mehr Tests, eine umfangreichere Codebasis notwendig ist, die Gesamtentwicklungszeit geringer ist (Müller & Padberg 2003). Die frühzeitige und häufige Durchführung der Tests hilft, Fehler früh im Entwicklungszyklus zu erkennen und zu beheben. Die frühzeitige Beseitigung von Fehlern vermeidet in der Regel eine aufwändige und langwierige Fehlersuche gegen Ende des Projekts, was zu einer Steigerung des Kosten-Nutzen-Verhältnisses (engl. *Benefit-Cost-Ratio* BCR) von bis zu 1:4,5 führt (Müller & Padberg 2003).

Im weiteren Verlauf dieser Arbeit wird als Vorgehensmodell der TDD-Ansatz zugrunde gelegt und auf die Softwareentwicklung in der Servicerobotik angewendet.

Aus diesem Kapitel werden die folgenden Ziele und Anforderungen abgeleitet (vergleiche Tabelle 3.1 und Tabelle 3.2):

⇒ **Ziel Z-EP-3:** Verkürzen der Dauer des Entwicklungsprozess

⇒ **Ziel Z-EP-4:** Senkung der Kosten des Entwicklungsprozesses

⇒ **Ziel Z-EP-8:** Erhöhung der Transparenz des Entwicklungsstandes

⇒ **Anforderung AF-9:** Einbindung des Testwerkzeugs in den Entwicklungsprozess

2.1.2 Systemintegration in der Servicerobotik

Als Systemintegration in der Robotik bezeichnet man den Prozess der Zusammenführung von verschiedenen Komponenten, wie z.B. von Aktoren, Sensoren,

Peripheriegeräten, Recheneinheiten oder Softwarekomponenten, zu einem System, das als eine Einheit funktioniert.

"*Systemintegration ist die anspruchsvollste Disziplin in der Robotik, da es ihre Aufgabe ist, Komponenten zusammenzubringen, die nicht dafür gebaut wurden zusammen zu funktionieren.*"
(Quelle: Dr.-Ing. Andreas Pott, Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen Universität Stuttgart, 2015)

Die Systemintegration in der Servicerobotik, speziell für mobile Serviceroboter, kann sich als anspruchsvolle Aufgabe darstellen, da es gilt diverse heterogene Hardware- und Softwarekomponenten unter typischerweise beschränktem Bauraum, beschränkter Energieversorgung und beschränkter Rechenkapazität zusammenzubringen.

Hardwareintegration

Als Hardwarekomponenten kommen dabei für Sensoren nicht nur verschiedene Sensorprinzipien, wie z.B. resistiv, induktiv, kapazitiv, thermisch, optisch (vergleiche Abbildung 1.3), zum Einsatz, sondern auch diverse Kommunikationsschnittstellen und -protokolle wie z.B. USB, RS232, FireWire, Ethernet, EtherCat, CAN, CANOpen u.v.m., bis hin zu proprietären Schnittstellen und Protokollen. Eine ähnliche Vielfalt gibt es bei der Ansteuerung von Aktoren für z.B. Fahrwerk, Räder, Arme, Gelenke und Greifer. Weiterhin haben mobile Serviceroboter häufig eine oder mehrere Recheneinheiten (z.B. PC, Mini-PC, Laptop, Einplatinenrechner) sowie Komponenten für die Kommunikation (z.B. Router, Antennen, UMTS-Modems, Bluetooth) zum Datenaustausch oder die Steuerung von außen mit an Bord.

Softwareintegration

Neben den oben aufgezählten Hardwarekomponenten gibt es auch Softwarekomponenten, die auf verschiedenen Ebenen zum Einsatz kommen. Diese reichen

von Hardwaretreibern zur Ansteuerung von Aktoren oder dem Auslesen von Sensoren bis hin zu Softwarekomponenten, mit robotikspezifischen Fähigkeiten zur Perzeption (Sensordatenverarbeitung, Objekt-, Personen- und Umgebungswahrnehmung), Navigation (Lokalisierung, Pfadplanung), Manipulation (Bewegungsplanung und -regelung, Greifplanung) sowie Softwarekomponenten zur Mensch-Maschine-Interaktion (GUI, Touchinteraktion, Gesteninteraktion, Sprachinteraktion) und Künstlicher Intelligenz. Weiter kommen Softwarekomponenten zum Einsatz, die die Kommunikation zwischen den erwähnten Softwarekomponenten (Middleware) ermöglichen sowie deren Management, Release, Installation, Konfiguration und Deployment übernehmen. Diese Funktionen werden teilweise durch das zugrundeliegende Betriebssystem oder Robotik-Framework bereitgestellt, die jeweils ihre eigenen Randbedingungen oder Softwarearchitekturen voraussetzen (siehe Kapitel 2.1.4).

Systemintegration

Die Systemintegration in ein Gesamtsystem zeichnet sich dadurch aus, dass es dabei nicht nur auf die Integration von Hard- und Softwarekomponenten ankommt, sondern dass insbesondere deren Interaktion mit der Umgebung mit berücksichtigt werden muss. Besonders in der mobilen Servicerobotik existiert typischerweise eine enge Kopplung zwischen Roboter und der Umgebung in der dieser agiert, hervorgerufen durch die gewollte und intensive Interaktion zwischen Roboter und seiner Umgebung. Abbildung 2.3 zeigt schematisch die Kopplung zwischen Roboter und Umgebung als Umgebung-Sensor-Roboter-Aktor-Kreis. Dies hat zur Folge, dass der Roboter mit dessen Sensorik nicht als abgeschlossenes System betrachtet werden darf, sonder die erfassten Sensordaten der Umgebung sich stetig verändern. Dies gilt zum Einen, wenn der Roboter in Bewegung ist, z.B. während einer Navigationsaufgabe. In diesem Fall ändert sich zu jedem Zeitpunkt die Sensorinformation der die Umgebung erfassenden Sensoren in Abhängigkeit von der Position, Orientierung und Geschwindigkeit des Roboters in der Umgebung. Zum Anderen verändert der Roboter die Umgebung indem

er mit dieser interagiert, beispielsweise beim Greifen von Objekten oder beim Öffnen von Türen, was wiederum zu veränderten Sensordaten führt.

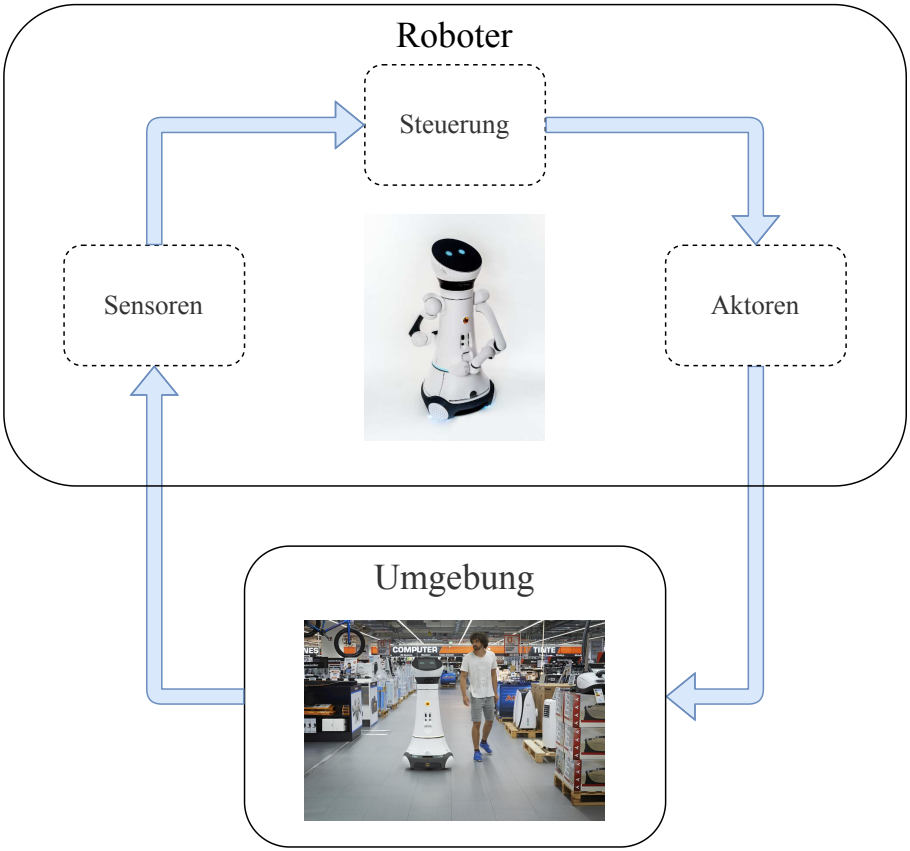


Abbildung 2.3: *Interaktion zwischen Roboter und Umgebung. Umgebung-Sensor-Roboter-Aktor-Kreis: Geschlossener Regelkreis aus Umgebung und Roboter mit Sensoren, Steuerung und Aktoren (Quelle: Fraunhofer IPA (oben), Mojin Robotics (unten)).*

Die Kopplung zwischen Umgebung und Roboter mit einem Regelkreis aus Sensoren, Steuerung und Aktoren hat zur Folge, dass sich das Testen von solchen Systemen oft als schwierig erweist, da interaktive Serviceroboter oftmals durch intensive Interaktion mit der Umgebung gekennzeichnet sind. Steht die Hardware nicht von Beginn des Softwareentwicklungsprozesses an zur Verfügung, muss der

Regelkreis aus Umgebung und Roboter nachgebildet oder simuliert werden, um das Gesamtsystem erfolgreich integrieren zu können.

2.1.3 Simulationsumgebungen

Simulierte Umgebungen bieten häufig die erste und in der Regel die häufigste Testumgebung für Robotersysteme, vor allem aufgrund ihrer Kosten- und Sicherheitsvorteile.

Es gibt Ansätze, die sowohl Roboter, als auch dessen Einsatzumgebung in 2D und 3D-Simulationsumgebungen inklusive der Generierung von Sensordaten nachbilden. Häufig kommen dabei Physik-Engines zum Einsatz, die grundlegende physikalische Prinzipien wie Masse, Trägheit, Reibung und Kollisionen für Festkörper nachbilden (Seugling & Rölin 2006). Je nach Art und Grad der Kopplung von Roboter und Umgebung nach Abbildung 2.3 ist die perfekte Nachbildung oder Simulation jedoch nicht immer möglich. Deshalb liegt der Fokus auf Simulationsumgebungen, die in der Lage sind reale Sensoren und Aktoren ausreichend genau zu simulieren. Ausschlaggebend für einen adäquaten Ersatz eines echten Roboters oder echter Roboterkomponenten durch eine Simulationsumgebung ist dabei eine *ausreichende* Genauigkeit. Ausreichend genau wird dabei in Bihlmaier & Wörn (2014) so definiert, dass durchschnittliche Roboter in durchschnittlichen Umgebungen, die durchschnittlich schwierige Aufgaben ausführen, mit einer Genauigkeit simuliert werden, die es ermöglicht Erfolg oder Misserfolg mit der gleichen durchschnittlichen Wahrscheinlichkeit wie im realen Szenario zu erzielen. Hierbei wird als Wahrscheinlichkeit der Grad des Vertrauens in die erzielte Lösung angesehen, siehe (Hájek et al. 2011). Sollte der echte Roboter eine Aufgabe zuverlässig ausführen können, dann sollte dies auch der simulierte Roboter können und umgekehrt. Es spielt dabei keine Rolle, ob die Simulation genau genug ist, um diesen Test auch für andere Arten von Aufgaben zu bestehen. In Estivill-Castro et al. (2018) wird gezeigt, dass Simulatoren genutzt werden können, um beschleunigte Tests zu durchlaufen, die schneller als Realzeit laufen.

Weiter ist zu beachten, dass sich ändernde Aspekte sowohl der Simulation als auch des realen Roboters sowie der nicht simulierten Systemteile (z.B. Motorsteuerungsalgorithmen) häufig nicht berücksichtigt werden, wenn auf Simulationsergebnisse zurückgegriffen wird. In Lier et al. (2012) wird ein Ansatz vorgestellt, um simulierte Robotermodelle zu verifizieren und zu verbessern, indem diese in einem iterativen CI-Ansatz mit einem realen Roboter verglichen werden.

Beispiele für in der Servicerobotik genutzte Simulationsumgebungen werden in Staranowicz & Mariottini (2011) aufgelistet und verglichen. Beispiele für Simulationsumgebungen sind USARSim (Carpin et al. 2007), VREP (Rohmer et al. 2013), MORSE (Echeverria et al. 2011) und Gazebo (Koenig & Howard 2004). Beim Vergleich der Simulationsumgebungen zeigt sich, dass es Unterschiede bezüglich Anzahl der zu simulierenden Komponenten (Sensoren, Aktoren) und deren physikalischer Interaktion (Dynamik, Trägheit, Reibung, Gelenkarten) sowie der Simulationsgenauigkeit und -geschwindigkeit, gibt. Aufgrund begrenzter Rechenkapazitäten sind Simulationsgenauigkeit und -geschwindigkeit gegenläufige Optimierungsgrößen, die nicht gleichzeitig und unabhängig voneinander erhöht werden können. Bei der Auswahl einer geeigneten Simulationsumgebung gilt es deshalb abzuwägen, welche Genauigkeit und Geschwindigkeit (Echtzeit) für die Simulation der jeweiligen Anwendung mit Kombination von Roboter und Umgebung notwendig ist. Für automatisierte Tests ist meist sogar eine Geschwindigkeit schneller als Echtzeit wünschenswert.

Für den weiteren Verlauf der Arbeit wird Gazebo als Simulator stellvertretend für viele oben genannte Simulationsumgebungen verwendet, weil Gazebo eine ganzheitliche Simulation darstellt und gut in ROS integriert ist. Durch die Vielzahl an Sensoren und aktuierten Gelenken sowie dynamische 3D-Umgebungen können die in Kapitel 5 dieser Arbeit ausgewählten Anwendungsbeispiele ausreichend genau modelliert und simuliert werden. Die Ergebnisse dieser Arbeit sind jedoch nicht auf den Einsatz von Gazebo beschränkt und lassen sich auf andere Simulationsumgebungen anwenden.

Aus diesem Kapitel werden die folgenden Ziele und Anforderungen abgeleitet (vergleiche Tabelle 3.1 und Tabelle 3.2):

⇒ **Anforderung AF-3:** Bereitstellung eines Testwerkzeugs zur Unterstützung von Tests in Simulation als *simulation-in-the-loop* Tests

2.1.4 Robotik-Frameworks

Robotik-Frameworks bieten eine Auswahl an Technologien, Methoden und Werkzeuge an, die Kommunikation, Integration, Deployment, Paketmanagement etc. bei der Entwicklung von komplexen, heterogenen und verteilten Robotik-Anwendungen strukturieren und vereinfachen. Oftmals forcieren die Frameworks eine gewisse Art und Weise, wie Komponenten in einem verteilten System interagieren und kommunizieren. Dieser Aspekt wird als Middleware bezeichnet. Eine Middleware ist eine Abstraktionsschicht, die sich zwischen dem Betriebssystem und den Softwareanwendungen befindet und wird in Schantz & Schmidt (2002) als eine Klasse von Softwaretechnologien definiert, mit deren Hilfe die Komplexität und Heterogenität von verteilten Systemen bewältigt werden kann. Sie wird als Softwareschicht oberhalb des Betriebssystems, aber unterhalb des Anwendungsprogramms angesehen, die eine gemeinsame Programmierabstraktion für ein verteiltes System bereitstellt. Sie wurde entwickelt, um die Heterogenität der Hardware zu verwalten, die Qualität der Softwareanwendungen zu verbessern, das Softwaredesign zu vereinfachen und die Entwicklungskosten zu senken. Der Einsatz von Robotik-Frameworks und Middleware-Frameworks bringt Vorteile in Bezug auf Softwaremodularität, Hardwareabstraktion, Plattformunabhängigkeit, Portabilität und Skalierbarkeit mit sich (Elkady & Sobh 2012).

In Elkady & Sobh (2012) wird eine umfassende Auswahl an Robotik-Frameworks und Middleware-Frameworks wie z.B. OpenRTM (Ando et al. 2005), YARP (Fitzpatrick et al. 2008; Metta et al. 2006), Player/Stage (Gerkey et al. 2003; Collett et al. 2005), OROCOS (Bruyninckx 2001) und ROS (Quigley et al. 2009) aufgelistet. Abbildung 2.4 zeigt eine chronologische Einordnung einiger Frameworks, die in Zug et al. (2013) gegenübergestellt und bezüglich

Hardwareunterstützung, Kommunikation, Programmierung und Debugging miteinander verglichen wurden.

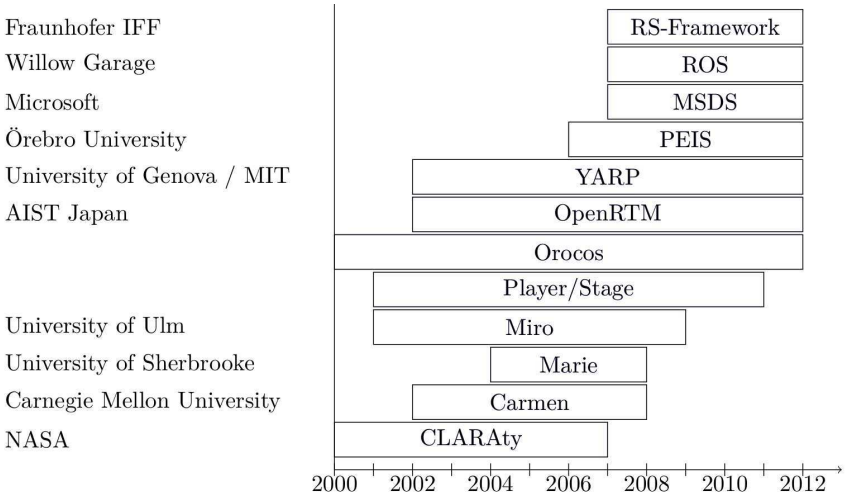


Abbildung 2.4: *Übersicht Robotik-Frameworks und deren chronologische Einordnung. (Quelle: Zug et al. 2013)*

Open Source Robot Operating System ROS

Ein in der Servicerobotik weit verbreitetes Framework ist das Open Source Robot Operating System ROS (Open Robotics 2020b), welches ein flexibles Framework für Robotiksoftware darstellt. ROS besteht aus einer Sammlung von Werkzeugen, Bibliotheken und Schnittstellendefinitionen mit dem Ziel das Erstellen von komplexen, robusten und hardwareübergreifenden Robotersteuerungen zu vereinfachen. ROS deckt sowohl das Paketmanagement, als auch die Middlewarekommunikation ab (Quigley et al. 2009) und treibt die Weiterentwicklung in einer weltweit agierenden Gemeinschaft voran. Die Abbildungen 1.4 und 2.5 zeigen die weltweite Verbreitung von ROS anhand der weltweiten Mitgliederverteilung und Zugriffsstatistik auf das ROS Wiki (Open Robotics 2020k).

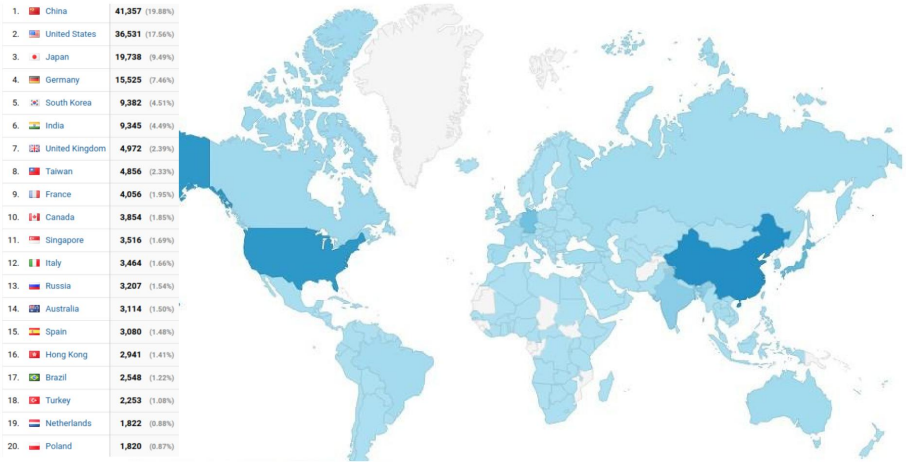


Abbildung 2.5: *Downloadstatistik ROS Wiki. (Quelle: ROS Community Metrics Report, Stand: 07/2019)*

ROS verwaltet Funktionalitäten in Paketen (*Packages*). Pakete können dabei Message-Definitionen, eine Anbindung an eine vorhandene Softwarebibliothek oder die Implementierung einer Funktionalität als ausführbares Programm (*Node*) darstellen. In einem laufenden ROS Netzwerk kommunizieren die Nodes über XML-RPC und TCP oder UDP via *Topics*, *Services* und *Actions* miteinander. Der *Rosmaster* oder *Roscore* stellt eine zentrale Instanz dar, die die Kommunikationskanäle zwischen den Nodes vermittelt und als *Parameterserver* auch eine zentrale Konfigurationsmöglichkeit darstellt. In *ROS Bags* (Open Robotics 2020d) können ROS Nachrichtendaten für die spätere Verwendung aufgenommen werden.

Für Servicerobotik-Anwendungen kommt typischerweise eine große Zahl an Nodes aus verschiedenen Paketen zum Einsatz. Tabelle 2.1 zeigt die Anzahl der verwendeten ROS Pakete für drei verschiedene Anwendungen (Museumsguide, Verkaufsassistent und Logistik). Der Museumsguide interagiert mit Besuchern über einfache Kommandos, führt diese durch das Museum und erklärt Exponate. Der Museumsguide baut daher vor allem auf einfacher Sprachinteraktion und Navigation auf. Der Verkaufsassistent bietet im Vergleich zum Museumsguide

Anwendung	Gesamtanzahl der eingesetzten Pakete	Direkte Abhängigkeiten zu eigenen Pakete	Einsatz fremder Pakete (inklusive rekursiver Abhängigkeiten)
Museumsguide	253	31 (12%)	147 (58%)
Verkaufsassistent Elektronikfachmarkt	341	40 (12%)	202 (59%)
Logistik	369	42 (11%)	228 (62%)

Tabelle 2.1: *Anzahl der eingesetzten ROS Pakete am Beispiel von Care-O-bot 4 Servicerobotik-Anwendungen als Maß für deren Komplexität. (Quelle: Mojin Robotics)*

komplexere Dialoge zur Produktsuche an und integriert dazu weitere Services wie z.B. eine Produktdatenbank. Im Bereich der Logistik kommen zu den genannten Fähigkeiten noch die Manipulation hinzu. Zu erkennen ist, dass mit steigender Komplexität der Anwendung auch die Anzahl der verwendeten ROS Pakete steigt und daher als Maß für die Komplexität einer Anwendung herangezogen werden kann. Der Anteil an direkten Abhängigkeiten (d.h. ohne rekursive Abhängigkeiten) liegt bei ca. 12% und der Gesamtanteil an fremden ROS Paketen bei ca. 60%, was als Maß für die Verflechtung und Wiederverwendbarkeit der eingesetzten Pakete interpretiert werden kann.

Die Wahl für diese Arbeit fällt auf ROS, da es ein beliebtes und weit verbreitetes Open Source Framework in der Servicerobotikforschung ist, das aber auch zunehmend in kommerziellen Anwendungen eingesetzt wird. Weiterhin ist bereits eine Vielzahl an Funktionalitäten und Werkzeugen (z.B. für Simulation, Visualisierung, Datenlogging etc.) integriert.

Aus diesem Kapitel werden die folgenden Ziele und Anforderungen abgeleitet (vergleiche Tabelle 3.1 und Tabelle 3.2):

⇒ **Anforderung AF-12:** Integration in ROS zur einfache Nutzung im ROS Umfeld

2.2 Softwarequalität in der Servicerobotik

In der Softwareentwicklung wird Softwarequalität in der ISO 25010:2011 (ISO 25010:2011) als der Grad definiert, in dem ein Softwareprodukt die angegebenen und implizierten Anforderungen erfüllt, wenn es bestimmungsgemäß verwendet wird. Dabei werden acht Kategorien mit jeweils mehreren Unterkategorien unterschieden:

- **Funktionalität:** funktionale Vollständigkeit, funktionale Korrektheit, funktionale Angemessenheit
- **Effizienz:** Zeitverhalten, Ressourcenverbrauch, Leistungsfähigkeit
- **Kompatibilität:** CO-Existenz mit weiterer Software, Interoperabilität
- **Benutzbarkeit:** angemessene Erkennbarkeit, Erlernbarkeit, Bedienbarkeit, Schutz vor Fehlbedienung durch den Nutzer, Ästhetik der Benutzerschnittstelle, Zugänglichkeit
- **Zuverlässigkeit:** Reifegrad, Verfügbarkeit, Fehlertoleranz, Wiederherstellbarkeit
- **Sicherheit:** Vertraulichkeit, Integrität, Nicht-Ablehnbarkeit, Verantwortlichkeit, Authentizität
- **Änderbarkeit:** Modularität, Wiederverwendbarkeit, Analysierbarkeit, Änderbarkeit, Testbarkeit
- **Übertragbarkeit:** Adaptierbarkeit, Installierbarkeit, Ersetzbarkeit

Während bei den nicht-funktionalen Eigenschaften zur Zuverlässigkeit, Benutzbarkeit und Effizienz die Anforderungen einzuordnen sind, die das Softwareprodukt bei seinem Betrieb erfüllen soll, sind die Qualitätskriterien zu Änderbarkeit und Übertragbarkeit auf die interne Beschaffenheit der Software, des Quellcodes, ausgerichtet.

2.2.1 Softwarequalität in der Open Source Softwareentwicklung

Die Entwicklung von Open Source Software (im Folgenden als Open Source bezeichnet) basiert auf einer relativ einfachen Idee: Die Software wird von einem einzelnen Programmierer oder einem Team von Programmierern entwickelt und bereits während der Entwicklungsphase wird der Quellcode veröffentlicht, den andere Programmierer frei lesen und je nach Lizenzmodell ändern und weitergeben können. Die Entwicklung des Systems vollzieht sich typischerweise schneller als bei nicht offenen Entwicklungsprojekten. Die Open Source Gemeinschaft hat es geschafft, einige weit verbreitete Produkte wie z.B. das Linux- und Android-Betriebssystem, den Apache-Webserver und diverse Browser zu entwickeln.

Befürworter der Open Source Softwareentwicklung behaupten, dass mit diesem Modell bessere Software als mit dem traditionellen geschlossenen Modell hergestellt wird, wofür es jedoch nur wenige empirische Belege gibt. In Stamelos et al. (2002) wird in einer Fallstudie untersucht, wie Open Source Entwicklung die strukturelle Qualität beeinflusst und wie die Vorteile der Open Source Entwicklung in einer strukturellen Qualitätsanalyse genutzt werden kann.

Ein Problem ist, dass der Open Source Entwicklungsprozess nicht genau definiert ist (McConnell 1999). Das Projekt wird normalerweise vom ursprünglichen Entwickler geleitet, der für alle Verwaltungsaktivitäten wie z.B. Freigabe neuer Versionen oder Konfigurationsverwaltung des sich schnell ändernden neuen Systems verantwortlich ist. In der Praxis kann es durch den nicht fest definierten Entwicklungsprozess (vergleiche Kapitel 2.1.1) vorkommen, dass Systemtests und Dokumentationen vernachlässigt werden und sich die Entwickler nur auf die Umsetzung von funktionalen Anforderungen fokussieren.

2.2.2 Softwarequalität in ROS

Die in Kapitel 2.2.1 genannten Probleme treffen auch auf Open Source Entwicklungen in der Servicerobotik u.a. im ROS Umfeld zu. Im ROS Umfeld sind die Herausforderungen trotz einiger Bestrebungen die Qualität von ROS Paketen zu kontrollieren, z.B. durch die Open Source Robotics Foundation OSRF für die

Ausführungsqualitäten (*Execution Qualities*)	Evolutionsqualitäten (*Evolution Qualities*)
Performance efficiency	Maintainability
Responsiveness (latency)	Reusability
Safety and Reliability	Portability
Robustness and Adaptability	Flexibility
Recoverability	Extensibility
Scalability	Modularity
Usability and Predictability	Changeability
Functional Correctness	Integrability
Interoperability	Testability

Tabelle 2.2: *Relevante Qualitätsmerkmale für Servicerobotiksoftware nach Mari & Eila (2003) unterteilt in Ausführungsqualitäten (engl. Execution Qualities) und Evolutionsqualitäten (engl. Evolution Qualities).*

grundlegenden ROS Funktionalitäten oder durch die ROS-Industrial Initiative für den Bereich industrielle Robotik, weiterhin von Bedeutung. In Reichardt et al. (2013) werden für die Servicerobotik relevante Softwarequalitätskriterien aufgeführt und untersucht, inwieweit diese Qualitätskriterien von dem verwendeten Robotikframework beeinflusst werden. Die Softwarequalität ist stark von nicht funktionalen Anforderungen abhängig, da diese „die Softwarequalität charakterisieren und die Wiederverwendung von Software ermöglichen“ (Brugali & Scandurra 2009). In Mari & Eila (2003) wird zwischen Ausführungsqualitäten (engl. *Execution Qualities*) und Evolutionsqualitäten (engl. *Evolution Qualities*) unterscheiden, siehe Tabelle 2.2. Die in Tabelle 2.2 zusammengefassten Qualitätsmerkmale sind laut Reichardt et al. (2013) für eine Vielzahl komplexer Steuerungssysteme für Serviceroboter besonders relevant.

Verschiedene Veröffentlichungen zu Robotersoftware befassen sich mit der Notwendigkeit und den Schwierigkeiten, diese Qualitätsmerkmale zu erreichen, z.B. Wiederverwendung von Software (Vaughan & Gerkey 2007; Brugali & Scandurra 2009; Baker et al. 2011), Robustheit und Zuverlässigkeit (Smart 2007), Skalierbarkeit (Shakhimardanov et al. 2011) oder Interoperabilität und Integration (Côté et al. 2007; Smits & Bruyninckx 2011). Offensichtlich ist es für einen Softwareentwickler eine Herausforderung, all diese Attribute beim Entwurf

einer Roboteranwendung zu berücksichtigen. Daher ist es wichtig, den Aufwand zur Einhaltung der einzelnen Qualitätsmerkmale, wie z.B. die Erstellung und Durchführung von Tests, so gering wie möglich zu halten.

Aus diesem Kapitel werden die folgenden Ziele und Anforderungen abgeleitet (vergleiche Tabelle 3.1 und Tabelle 3.2):

⇒ **Ziel Z-EP-5:** Reduktion des Einsatzes von Entwicklerressourcen für das Testen

⇒ **Ziel Z-EP-6:** Minimierung des Aufwands zur Erstellung von Tests

⇒ **Ziel Z-EP-7:** Minimierung des Aufwands zur Durchführung von Tests

⇒ **Anforderung AF-1:** Bereitstellung vordefinierter Testbausteine

2.3 Qualitätssicherung in der Servicerobotik-Softwareentwicklung

Zunächst werden in Kapitel 2.3.1 Qualitätssicherungsmechanismen und Werkzeuge für die Servicerobotik-Softwareentwicklung vorgestellt, mit denen die Softwarequalität gemessen und gesteigert werden kann. Weiterhin wird in Kapitel 2.3.2 der Einsatz von Softwaretests genauer analysiert und deren Einfluss auf die Softwarequalität in der Servicerobotik erörtert und abschließend in Kapitel 2.3.3 Testframeworks in der Servicerobotik analysiert und gegenübergestellt.

2.3.1 Methoden und Werkzeuge zum Messen und Steigern von Softwarequalität

Es existieren viele unterschiedliche Methoden und Werkzeuge zum Messen und Steigern von Softwarequalität in der allgemeinen Softwareentwicklung. Viele davon kommen in der Servicerobotik-Softwareentwicklung regelmäßig zum

Einsatz. Diese Methoden und Werkzeuge sind neben der Verwendung eines geeigneten Robotik-Frameworks geeignete Maßnahmen zur Unterstützung bestimmter Softwarequalitätsattribute. Durch die Förderung guter Softwareentwicklungspraktiken wie z.B. die Trennung von Framework-unabhängigem von Framework-abhängigem Code werden Wiederverwendbarkeit und Portabilität von Softwarekomponenten erhöht. Im Folgenden werden einige der in der Servicerobotik, vor allem aus dem ROS Umfeld, gebräuchlichen Methoden und Werkzeuge vorgestellt.

Gestaltungsrichtlinien für Quellcodeerstellung (*Style Guides*)

Gestaltungsrichtlinien (engl. *Style Guides*) decken typischerweise Teilbereiche der Entwicklung und Verwaltung von Software ab, sodass für ein Entwicklungsprojekt mehrere Gestaltungsrichtlinien auf unterschiedlichen Ebenen zum Einsatz kommen: z.B. für die Benamung von Verzeichnissen, Dateien, Variablen, Methoden und Klassen, die Art und Weise wie der Quellcode verwaltet wird, wie häufig Releases erstellt werden und wie und wo die Dokumentation erfolgt. Motivation für den Einsatz von Gestaltungsrichtlinien ist die Erhöhung der Konsistenz und damit vor allem die Erhöhung des Softwarequalitätskriteriums Änderbarkeit. Beispiele für Programmierstandards aus der Automobilindustrie sind AUTOSAR (Staron 2017) sowie MISRA-C bzw. MISRA-C++ (MISRA 2008). Diese Programmierstandards definieren eine Untermenge des Sprachumfangs von C bzw. C++, d.h. sie umfassen Richtlinien, die zu einer Qualitätssteigerung, insbesondere der Softwarequalitätsaspekte der Zuverlässigkeit und Wartbarkeit, in der Softwareentwicklung führen sollen. Zur Überprüfung der Konformität existieren kommerzielle Werkzeuge, wie z.B. Axivion Bauhaus Suite (Axivion 2020), Astrée (Absint 2020a) oder RuleChecker (Absint 2020b).

Im ROS Umfeld kommen Gestaltungsrichtlinien z.B. für die Paketverwaltung (Open Robotics 2020g) und die verschiedenen Programmiersprachen (Open Robotics 2020h; Open Robotics 2020i) zum Einsatz.

Manuelle Codeprüfung (*Code Reviews*)

Manuelle Codeprüfung (engl. *Code Review*), manchmal auch als *Peer Review* bezeichnet, ist eine Aktivität zur Qualitätssicherung von Software, bei der eine oder mehrere Personen ein Programm überprüfen, indem sie hauptsächlich Teile des Quellcodes anzeigen und lesen. Mindestens eine der Personen darf dabei nicht der Autor des Codes sein. Die Personen, die die Überprüfung durchführen, mit Ausnahme des Autors, werden als Gutachter bezeichnet. Die Durchführung von Code Review dienen den folgenden Zielen: der Verbesserung der internen Codequalität (Lesbarkeit, Änderbarkeit), dem Auffinden von Fehlern (Funktionalität, Korrektheit, Sicherheit) und dem Einhalten von Qualitäts-Gestaltungsrichtlinien. Code Reviews sind häufig an die Source Code Versionsverwaltung geknüpft (z.B. bei Github in Form von Pull Requests) und können entweder manuell oder (teil-) automatisiert durchgeführt werden (Github 2020b).

Zertifizierung und Klassifizierung

Als Zertifizierung bezeichnet man ein Verfahren, mit dessen Hilfe die Einhaltung bestimmter Anforderungen nachgewiesen wird. Die Zertifizierung bezieht sich auf die Bestätigung bestimmter (Qualitäts-)Merkmale einer Software. Diese Bestätigung erfolgt häufig durch eine externe Überprüfung oder Bewertung.

Die ROS-Industrial Initiative (Edwards & Lewis 2012; ROS-Industrial Consortium 2020b) hat eine Zertifizierung für ROS Komponenten eingeführt (ROS-Industrial Consortium 2020c). Ziel ist die Identifizierung von ROS Paketen, die zuverlässig genug sind, um im produktiven Einsatz verwendet werden zu können. Dieses Ziel steht häufig im Widerspruch zu den agilen Softwareentwicklungsmethoden der Open Source ROS Gemeinschaft. Oft wird von ROS Entwicklern Software in unterschiedlichen Reifegraden bereitgestellt (siehe Kapitel 2.2.2). Um diese Software als solche zu identifizieren, hat ROS-Industrial eine dreistufige Skala zur Klassifizierung des Reifegrads eingeführt.

- **Experimental:** Dieser Status zeigt an, dass es sich bei dieser Software bestenfalls um experimentellen Code handelt. Es gibt bekannte Probleme

und fehlende Funktionen. Die Programmierschnittstellen (engl. *Application Programming Interface* API) sind instabil und können sich ändern. Der Einsatz in Produktivsystemen wird nicht empfohlen. Jedes neue Paket beginnt auf dieser Ebene.

- **Developmental:** Dieser Status zeigt an, dass diese Software noch nicht produktivreif ist. Die Software hat ein gewisses Maß an Unittests. Es gibt bekannte Probleme und fehlende Funktionen. Die APIs sind instabil, es ist jedoch unwahrscheinlich, dass sie sich drastisch ändern. Die Verwendung in Produktivsystemen erfordert wahrscheinlich Änderungen, einschließlich Verbesserungen und/oder Fehlerkorrekturen.

- **Production:** Dieser Status zeigt an, dass diese Software produktivreif ist. Es erfüllt qualitative Kriterien für den Einsatz in Produktivsystemen. Es sind nur wenige oder keine Probleme bekannt und keine davon sind kritisch. Die APIs sind stabil und es ist unwahrscheinlich, dass sie sich ändern. Die Software verfügt über eine umfassende Testsuite und Dokumentation und wurde in der Vergangenheit in Produktivsystemen verwendet. Der Code wurde einer begrenzten Überprüfung unterzogen, um diesen Status zu erhalten.

ROS2 definiert eine 5-stufige Qualitätseinstufung von Paketen (Open Robotics 2020m). Die Einstufung basiert auf den unterschiedlichen Erfüllungsgraden der folgenden Kriterien: Versionsrichtlinien, Änderungsprozess, Dokumentation, Test, Abhängigkeiten, Zielplattformen und IT-Sicherheit. Insbesondere sind System-, Integrations-, Unit- und Leistungstests sowie deren Testabdeckung und statische Codeanalyse vorgesehen.

- **Quality Level 1:** Höchste Qualitätsstufe, enthält Pakete die im Produktivbetrieb eingesetzt werden und hohen Anforderungen unterliegen.

- **Quality Level 2:** Hohe Qualitätsstufe, enthält Pakete die auf dem Weg zu Stufe 1 sind oder allgemein eine weite Verbreitung haben, jedoch nur manchmal im Produktivbetrieb eingesetzt werden.

- **Quality Level 3:** Qualitätsstufe für Tools zur Unterstützung der Entwicklung und Fehlersuche. Diese Pakete werden typischerweise nicht im Produktivbetrieb eingesetzt.
- **Quality Level 4:** Qualitätsstufe für Demo-, Tutorial- oder experimentelle Pakete, die nicht für eine direkte Wiederverwendung gedacht sind und keinen strengen Anforderungen unterliegen. Diese Pakete können jedoch z.B. eine ausführliche Dokumentation enthalten und als Implementierungsvorlage dienen.
- **Quality Level 5:** Qualitätsstufe für Pakete, die nicht einmal die Mindestanforderungen für Stufe 4 erfüllen und daher nicht verwendet werden sollten. Stufe 5 selbst stellt keine Anforderungen.

Sowohl der ROS-Industrial Status, als auch die ROS2 Qualitätseinstufung basieren nicht auf einer objektiven und quantitativen Bewertung. Der Status ist lediglich die subjektive und qualitative Meinung des Prüfers, Entwicklers oder Maintainers darüber, ob der Code produktivreif ist. Diese Indikatoren werden verwendet, um Benutzern und Entwicklern von ROS die Zuverlässigkeit und Verwendbarkeit von ROS Paketen zu visualisieren, stellt dabei aber keine objektive und quantitative Methode dar.

Kontinuierliche Integration (CI)

In der Softwareentwicklung ist Kontinuierliche Integration (engl. *Continuous Integration* CI) eine Methode, bei der der Quellcode fortlaufend kompiliert und getestet wird. Oftmals geschieht dies mehrmals täglich oder bei jeder Codeänderung. Ursprünglich bei Extreme Programming Entwicklungsprozessen eingeführt, wird das Konzept von CI heute in nahezu allen Softwareentwicklungsbereichen eingesetzt (Duvall et al. 2007; Smart 2011; Meyer 2014).

CI kann in Kombination mit automatisierten Komponententests verwendet werden, die z.B. auf der Grundlage von Test-Driven-Development erstellt wurden. Die Automatisierung wird mithilfe eines Build-Servers erreicht, der den Quellcode kompiliert und die Ergebnisse an die Entwickler zurückmeldet. In den

meisten Fällen werden Tests auch von einem Build-Server ausgeführt. Zusätzlich zum Ausführen der Unit- und Integrationstests führen solche Build-Server zusätzliche statische Analysen durch (siehe Kapitel 2.3.2), messen und extrahieren Leistungswerte, extrahieren und formatieren Dokumentation aus dem Quellcode und erleichtern damit manuelle Qualitätssicherungsprozesse. Es gelten dabei u.a. folgende Grundsätze:

- **Pflegen einer Hauptversion für alle Nutzer:** Dieser Grundsatz befürwortet die Verwendung eines Versionskontrollsystems für den Quellcode des Projekts. Alle zum Erstellen des Projekts erforderlichen Artefakte sollten im Repository abgelegt werden. Statt mehrere Kopien für jeden Entwickler zu verwalten, ist es vorzuziehen dass Änderungen in den Hauptzweig integriert werden. Der Hauptzweig sollte der Ort für eine stets lauffähige Version der Software sein.

- **Den Prozess Automatisieren:** Der Prozess der Kontinuierlichen Integration sollte keinen manuellen Mehraufwand bedeuten, d.h. der Prozess des Kompilierens und Testens sollte automatisiert und nicht-interaktiv ablaufen.

- **Testen ist Teil des Prozesses:** Sobald der Quellcode kompiliert ist, sollten alle Tests ausgeführt werden, um zu verifizieren, dass er sich so verhält, wie die Entwickler es erwarten.

- **Häufige Integration und kurze Testzyklen:** Der Prozess sollte schnell abgeschlossen werden können, damit ein Integrationsproblem schnell erkannt werden kann und schnelle Rückmeldung an die Entwickler erfolgen kann.

- **Testergebnisse transparent zur Verfügung stellen:** Nach einem Testdurchlauf sollten die detaillierten Ergebnisse des Tests für alle Entwickler abgefragt werden können, um so den Nachbearbeitungsaufwand gering zu halten, der beim Neuerstellen eines Features anfällt, das die Anforderungen nicht erfüllt. Durch frühzeitiges Testen wird außerdem die Wahrscheinlichkeit verringert, dass Fehler bis zur Bereitstellung bestehen bleiben. Ein

früheres Auffinden von Fehlern kann den Arbeitsaufwand verringern, der zu deren Behebung erforderlich ist.

CI wird häufig mit *Continuous Delivery* und *Continuous Deployment* verknüpft. Continuous Delivery steht dabei für das fortlaufende und inkrementelle Bereitstellen von lauffähigen Anwendungen, welches meist die Ergebnisse eines erfolgreichen CI-Durchlaufs sind. Wird die lauffähige Anwendung dann direkt in den produktiven Einsatz gebracht, spricht man von Continuous Deployment. Abbildung 2.6 zeigt den Zusammenhang von Continuous Integration, Continuous Delivery und Continuous Deployment im Entwicklungsprozess.

Abbildung 2.6: *CI und CD im Entwicklungsprozess. Einordnung von Continuous Integration, Continuous Delivery und Continuous Deployment im Entwicklungsprozess. (Quelle: Redhat)*

Neben Werkzeugen wie Jenkins (Smart 2011), gibt es auch gehostete Services, die meist direkt an gehostete Versionskontrollsysteme wie z.B. Github integriert sind. Zwei verbreitet genutzte gehostete Services für Open Source Projekte sind Travis CI (Travis CI 2020) und Github Actions (Github 2020a), welche auch in der ROS Gemeinschaft viel genutzt werden. Allerdings führen einer Auswertung von Open Source Projekten auf Travis CI zufolge nur 58,64% der CI-Jobs Tests durch (Durieux et al. 2019). Demnach gibt es Bedarf an der weiteren Adoption von Test-getriebener Entwicklung in Open Source Projekten (Duvall et al. 2007). Weitere im ROS Umfeld genutzte CI-Services sind die offizielle ROS Buildfarm (basierend auf Jenkins) (Open Robotics 2020e), Buildbot (Ferguson 2014), Shadow Robots Build Tools (Shadow Robotics 2020), ROS Gitlab CI (Lamoine 2020) und das auf Travis CI und Github Actions basierende ROS-Industrial CI (ROS-Industrial Consortium 2020a).

Der Schwerpunkt der aufgeführten Werkzeuge liegt auf der Verwendung als automatisierte Continuous Integration Server, die das Kompilieren von ROS Code für verschiedene Architekturen und Betriebssysteme, die Durchführung von Unittests und die Erstellung von Releases ermöglichen. Allerdings bleibt der Aufwand für die Erstellung und Konfiguration von Tests weiterhin beim Entwickler. Da ROS-Industrial CI einen umfangreichen und aktiv entwickelten sowie in vielen ROS Paketen genutzten Funktionsumfang bietet und in Kombination mit dem weit verbreitenden Travis CI und Github Actions bestehende Infrastruktur nutzt, wird im weiteren Verlauf dieser Arbeit auf diesen Werkzeugen aufgebaut.

Aus diesem Kapitel werden die folgenden Ziele und Anforderungen abgeleitet (vergleiche Tabelle 3.1 und Tabelle 3.2):

⇒ **Anforderung AF-8:** Integration in Infrastruktur zur Testautomatisierung

2.3.2 Softwaretests

Ein Softwaretest prüft einen Softwarebaustein, durch einen jederzeit wiederholbaren Nachweis, auf die Erfüllung von zuvor festgelegten Anforderungen und misst dessen Qualität. Der damit einhergehende Vorgang wird als Testen bezeichnet.

"*Testing and programming is faster than just programming.*"
(Eckel & Allison 2016)

In Chemuturi (2006) wird zwischen Testen für ein einzelnes Projekt und Testen für ein Produkt unterschieden. Wenn Software für einen einzelnen Kunden bzw. Anwendungszweck entwickelt wird, kommen nach Chemuturi (2006) neben manuellen Sichtprüfungen (Code Reviews), insbesondere für komponentenbasierte Systeme die folgenden aufeinander aufbauenden Teststufen zum Einsatz (Grechenig 2010):

- **Statische Codeanalyse:** Als statische Codeanalyse wird ein statisches Softwaretestverfahren bezeichnet, das rein auf der Analyse des Quellcodes beruht, bevor dieser kompiliert ist.

- **Unittests:** Als Unittest wird ein Softwaretest bezeichnet, der überprüft, ob ein Teil einer Anwendung (engl. *Unit*) dem Entwurf entspricht und sich wie beabsichtigt verhält.

- **Integrationstests:** Integrationstests sind eine aufeinander abgestimmte Reihe von Einzeltests, die dazu dienen, verschiedene voneinander abhängige Komponenten eines komplexen Systems im Zusammenspiel miteinander zu testen. Die gemeinsam zu testenden Komponenten sollten jeweils eigene Unittests erfolgreich bestanden haben und für sich isoliert fehlerfrei funktionsfähig sein.

- **Systemtests:** Der Systemtest dient dazu, die Erfüllung der Systemanforderungen zu prüfen. Der Zweck des Systemtestprozesses ist die Sicherstellung, dass die Implementierung aller Systemanforderungen auf deren Erfüllung getestet wird, und dass das System zur Auslieferung bereit ist.

- **Akzeptanz- bzw. Abnahmetests:** Ein Akzeptanztest oder Abnahmetest überprüft, ob eine Software aus Sicht des Benutzers wie beabsichtigt funktioniert und dieser die Software akzeptiert.

Die genannte Teststufen bauen dabei aufeinander auf, sodass spätere Teststufen jeweils davon ausgehen, dass die vorgelagerten Tests erfolgreich abgeschlossen werden. Dabei steigt der Testaufwand von einem einzelnen Unittest zu einem kompletten System- oder Akzeptanztest stark an, weshalb eine Testautomatisierung anzustreben ist. Während Unittests häufig vom Entwickler selbst geschrieben werden, fordert der allgemeine Grundsatz der Qualitätssicherung, dass Systemtests von einem vom Entwickler unabhängigen Tester durchgeführt werden sollte (Vier-Augen-Prinzip). Die unabhängige Bewertung kann auch durch den Einsatz automatisierter Testauswertung anhand objektiver Metriken erreicht werden.

Aus diesem Kapitel werden die folgenden Ziele und Anforderungen abgeleitet (vergleiche Tabelle 3.1 und Tabelle 3.2):

⇒ **Anforderung AF-6:** Unterstützung zur automatisierten Auswertung der Tests anhand von objektiven Metriken

Wird die Software darüber hinaus in verschiedenen Anwendungsfällen bzw. bei verschiedenen Kunden eingesetzt, kommen neben den oben genannten Testarten noch weitere Tests zur Anwendung, davon sind für die vorliegende Arbeit insbesondere die folgenden Tests relevant:

- **Funktionale Tests:** Funktionale Tests unterziehen der zu testenden Anwendung Tests, bei denen die funktionalen Anforderungen überprüft werden.
- **Deployment Tests:** Ziel von Deployment Tests ist es sicherzustellen, dass die Anwendung auch auf dem Zielsystem (z.B. Zielhardware) ordnungsgemäß installiert und gestartet werden kann.
- **Sanity Tests:** Sanity Tests umfassen z.B. Plausibilitätsprüfungen, einfache modellbasierte Prüfungen und Konsistenzprüfungen von Anforderungen.
- **Regression Tests:** Unter Regressionstests versteht man die Wiederholung von Testfällen, um sicherzustellen, dass Modifikationen in bereits getesteten Teilen der Software keine neuen Fehler (Regressionen) verursachen.
- **Performance Tests:** Performance Tests werden durchgeführt, um zu bestimmen, wie ein System in Bezug auf Reaktionsfähigkeit und Stabilität unter einer bestimmten Arbeitslast funktioniert.

Üblicherweise werden nicht alle aufgelisteten Testarten in einem Projekt verwendet, jedoch oft viele davon.

White-Box-Test

Der Begriff White-Box-Test bezeichnet eine Methode des Softwaretests, bei der die Tests mit Kenntnissen über die innere Funktionsweise des zu testenden

Systems entwickelt werden (Spillner & Linz 2012). Für White-Box-Tests muss also zwingend Zugriff auf den Quellcode bestehen. Deshalb ist die Durchführung von White-Box-Test bei kommerzieller Software meist nicht möglich, prinzipiell bei Open Source Software jedoch immer möglich, da hier definitionsgemäß Zugriff auf den Quellcode existiert. Zu den White-Box-Tests gehört vor allem die statische Codeanalyse.

Black-Box-Test

Black-Box-Test bezeichnet eine Methode des Softwaretests, bei der die Tests ohne Kenntnisse über die innere Funktionsweise des zu testenden Systems entwickelt werden. Er beschränkt sich auf funktionsorientiertes Testen, d. h. für die Ermittlung der Testfälle werden nur die Anforderungen, aber nicht die Implementierung des Testobjekts herangezogen. Die genaue Beschaffenheit des Programms wird nicht betrachtet, sondern vielmehr als Black-Box behandelt. Nur nach außen sichtbares Verhalten fließt in den Test ein (Spillner & Linz 2012).

Black-Box-Tests werden meist für kommerzielle Software verwendet, jedoch auch bei Open Source Software werden Komponenten oft als Black-Box betrachtet, da die Ressourcen nicht verfügbar bzw. das Wissen nicht vorhanden ist, tiefer in einzelne fremde Komponenten einzutauchen. Als Black-Box-Tests können z.B. Integrations-, System- und Benutzerakzeptanztests ausgeführt werden.

Statische Codeanalyse

Mithilfe statischer Codeanalyse können eine Reihe von Fehlern entdeckt werden, noch bevor ein Programm ausgeführt wird. Damit gehört dieses Verfahren zu den White-Box-Testverfahren, da zwingend Zugriff auf den Quellcode vorhanden sein muss. Statische Codeanalyse gehört zu den falsifizierenden Verfahren, d.h. eine erfolgreiche statische Analyse ist keine Garantie dafür, dass das Programm später fehlerfrei ausgeführt werden kann. Als statische Analyse zählen Verfahren wie z.B. manuelles und automatisiertes Code Review (siehe Kapitel 2.3.1 *Code Reviews*) sowie die ebenfalls manuelle oder automatisierte Überprüfung von

Gestaltungsrichtlinien (siehe Kapitel 2.3.1 *Style Guides*), die meist mithilfe eines CI-Servers in den Entwicklungsprozess integriert werden. In Pitzer (2012) wird untersucht, inwieweit durch statische Codeanalyse Qualitätskriterien für ROS Pakete abgeleitet werden. Es ist festzustellen, dass es Metriken gibt, die Hinweise auf die Codequalität liefern, jedoch ist eine Aussage über die Funktionalität bzw. den Funktionsumfang der ROS Pakete nicht möglich.

Statische Codeanalyse kann nur begrenzt eine Aussage über funktionale Korrektheit treffen, kann jedoch unabhängig mit anderen Testtypen kombiniert werden. Statische Codeanalyse wird daher im weiteren Verlauf dieser Arbeit nicht speziell untersucht.

Hardware-in-the-loop und *simulation-in-the-loop* Tests

Hardware-in-the-loop bezeichnet ein Verfahren, bei dem ein eingebettetes System (z.B. Motorsteuergerät oder mechatronische Komponente) über seine Ein- und Ausgänge an ein Testsystem angeschlossen wird, welches die reale Umgebung nachbildet. Dieses Verfahren wird verwendet, um während der Entwicklung frühzeitig z.B. mit dem realen Steuergerät testen zu können, bevor das Gesamtsystem zum Testen bereitsteht. Eine weitere Anwendung des Verfahrens ist die frühzeitige Inbetriebnahme.

Bei *Software-in-the-loop* bzw. *Simulation-in-the-loop* wird das Verfahren auf Systeme angewandt, bei denen im Gegensatz zu *Hardware-in-the-loop* keine besondere Hardware eingesetzt wird, sondern ein Software- bzw. Simulationsmodell davon. Das Modell wird auf dem Entwicklungsrechner zusammen mit der zu testenden Software ausgeführt, anstatt wie bei *Hardware-in-the-loop* auf der Zielhardware zu laufen.

Beide Verfahren sind gut geeignet, um damit Systemtests umzusetzen. In der Servicerobotik werden dabei häufig die in Kapitel 2.1.3 vorgestellten Simulationsumgebungen eingesetzt. Im weiteren Verlauf dieser Arbeit werden *hardware-in-the-loop* und *simulation-in-the-loop* Verfahren angewendet, um automatisierte Systemtests zu realisieren.

Aus diesem Kapitel werden die folgenden Ziele und Anforderungen abgeleitet (vergleiche Tabelle 3.1 und Tabelle 3.2):

⇒ **Anforderung AF-2:** Bereitstellung eines Testwerkzeugs zur Unterstützung von Tests mit realer Hardware als *hardware-in-the-loop* Tests

⇒ **Anforderung AF-3:** Bereitstellung eines Testwerkzeugs zur Unterstützung von Tests in Simulation als *simulation-in-the-loop* Tests

Testabdeckung und statistische Versuchsplanung (*Design of Experiments*)

Die Testabdeckung ist ein Maß, mit dem beschrieben wird, zu welchem Anteil der Quellcode eines Programms ausgeführt wird, wenn eine bestimmte Testsuite ausgeführt wird. Bei einem Programm mit hoher Testabdeckung, wurde während des Tests mehr Quellcode ausgeführt, was darauf hindeutet, dass die Wahrscheinlichkeit, unerkannte Softwarefehler zu enthalten, geringer ist als bei einem Programm mit geringer Testabdeckung. Zur Berechnung der Testabdeckung können viele verschiedene Metriken verwendet werden. Einige der grundlegendsten sind der Prozentsatz der Programmanweisungen, die während der Ausführung der Testsuite aufgerufen werden.

Eine vollständige Testabdeckung bedeutet in der Praxis oft, dass die Anzahl möglicher Testfälle sehr schnell groß wird (z.B. durch kombinatorische Explosion der Parameter). Stattdessen beschränkt man sich auf eine Auswahl sinnvoll erscheinender Tests für Grenzfälle. Diese Auswahl kann mithilfe von statistischer Versuchsplanung (engl. *Design of Experiments*) ermittelt werden (Kuhn & Reilly 2002). Statistische Versuchsplanung umfasst statistische Verfahren, die vor der Durchführung von Tests angewendet werden. Beispiele hierfür sind: die Bestimmung des minimal erforderlichen Versuchsumfanges zur Einhaltung von Genauigkeitsvorgaben, das Aufstellen von faktoriellen Testplänen und sequentielle Testplanung und -auswertung. Da die Durchführung von Tests Ressourcen benötigen (Personal, Zeit, Geräte usw.), ergibt sich ein Zwiespalt zwischen einerseits der Genauigkeit und Zuverlässigkeit der zu erwarteten Er-

gebnisse und andererseits dem dazu notwendigen Aufwand. Mit der statistischen Versuchsplanung wird mit möglichst wenigen Tests der Wirkzusammenhang zwischen Einflussfaktoren (= unabhängige Variablen) und Zielgrößen (= abhängige Variable) möglichst genau ermittelt.

Aus diesem Kapitel werden die folgenden Ziele und Anforderungen abgeleitet (vergleiche Tabelle 3.1 und Tabelle 3.2):

⇒ **Ziel Z-EP-1:** Erreichen einer hohen Testabdeckung

⇒ **Ziel Z-EP-5:** Reduktion des Einsatzes von Entwicklerressourcen für das Testen

⇒ **Anforderung AF-5:** Bereitstellung eines Werkzeugs zur automatischen Generierung von Testsuiten

Motivation für (automatisiertes) Testen im Entwicklungsprozess

Beim Schreiben von Software kommt es vor allem darauf an, die gestellten Anforderungen zu erfüllen. Allerdings ist es schwierig, diese Anforderungen zu Beginn eines Projekts lückenlos aufzustellen, zumal diese sich während des Entwicklungszeitraums auch verändern können (Eckel & Allison 2016). Um diesen Umstand zu berücksichtigen haben sich Entwicklungsmethoden durchgesetzt, die einen iterativen Entwicklungsstiel verfolgen (siehe Kapitel 2.1.1). Hier wird in kurzen Zyklen zwischen Anforderungsanalyse, Entwurf, Implementierung und Test gewechselt. Wird hierbei der Testaufwand des Entwicklers durch automatisiertes Testen reduziert, bleibt mehr Zeit für die eigentliche Programmieraufgabe übrig.

Automatisierte Tests vereinfachen die Wartung, da diese das zu testende Softwarepaket automatisch auf aktualisierte Abhängigkeiten und aktualisierte APIs testen und dabei sehr schnell festgestellt werden kann, ob das Paket noch funktioniert oder angepasst werden muss.

Automatisiertes Testen hilft anderen Entwicklern Änderungen beizutragen und abzuschätzen, ob diese bestehende Funktionalitäten beeinflussen.

Eine weitere Motivation für automatisiertes Testen im Entwicklungsprozess ist, dass die kreative Seite eines Programmierers stets bestrebt ist, die bestehende Software besser zu strukturieren, wobei dies oft in Konflikt mit dem Erhalt einer stabilen lauffähigen Softwareversion steht. Auch hier kann automatisiertes Testen helfen, die nötige Gewissheit zu schaffen, dass nach einer Neustrukturierung der Software der Funktionsumfang weiterhin garantiert werden kann.

Gleich aus welchem Grund Software angepasst, erweitert oder neu strukturiert wird, gilt es durch Werkzeuge sicherzustellen, dass die Software diese Veränderungen übersteht und über die Zeit hinweg kontinuierlich verbessert werden kann.

Aus diesem Kapitel werden die folgenden Ziele und Anforderungen abgeleitet (vergleiche Tabelle 3.1 und Tabelle 3.2):

⇒ **Ziel Z-EP-2:** Steigerung der Testautomatisierung

⇒ **Ziel Z-EP-5:** Reduktion des Einsatzes von Entwicklerressourcen für das Testen

⇒ **Ziel Z-EP-6:** Minimierung des Aufwands zur Erstellung von Tests

⇒ **Ziel Z-EP-7:** Minimierung des Aufwands zur Durchführung von Tests

⇒ **Anforderung AF-8:** Integration in Infrastruktur zur Testautomatisierung

⇒ **Anforderung AF-9:** Einbindung des Testwerkzeugs in den Entwicklungsprozess

Softwaretests in ROS

Es gibt Bestrebungen, die Qualität und Nutzbarkeit von ROS zu erhöhen. U.a. zielt das ROSIN Projekt (ROSIN Consortium 2020) darauf ab und liefert einen Überblick über den Stand der Technik zu Softwarequalität und Qualitätssicherungsmechanismen in ROS (ROSIN Consortium 2017).

Viele der in Kapitel 2.3.2 vorgestellten Tests werden im ROS Umfeld genutzt. Bezogen auf ROS Pakete werden daraus drei Level an Tests abgeleitet (Open Robotics 2020j):

- **Level 1 (Unittests für Bibliotheken):** Falls der ROS Node eine ROS unabhängige Bibliothek anbindet, sollten für diese ROS unabhängige Unittests erstellt werden, die die eigentliche Kernfunktionalität der Bibliothek überprüfen. Hierbei kommen die bekannten Unittests der genutzten Programmiersprachen wie z.B. Unittest für Python (Open Robotics 2020c) oder Gtest für C++ (Open Robotics 2020a) zum Einsatz.
- **Level 2 (ROS Node Tests):** Tests für ROS Nodes starten und Konfigurieren einen zu testenden ROS Node und kommunizieren während des Tests über seine externe API, z.B. Topics, Services und Actions. Hierbei kommen Rostests (Open Robotics 2020f) zum Einsatz.
- **Level 3 (Integrations-/Regressionstests für mehrere ROS Nodes):** In diesem Level wird ein Netzwerk aus mehreren ROS Nodes konfiguriert und gestartet, um zu überprüfen, ob diese fehlerfrei zusammen wirken. Da es sich bei mehreren Nodes um mehrere Prozesse, ggf. auf einem verteilten System, handelt, gleicht es der Fehlersuche in nebenläufigen Prozessen mit Deadlocks, Race Conditions etc.

Um aussagekräftige und vergleichbare Ergebnisse zu erhalten, ist zu beachten, dass die Ausführung der Anwendung nicht durch das Testsystem beeinflusst wird. Dies gilt zum Einen für das Zeitverhalten der Anwendung, d.h. die Beobachtung durch das Testsystem darf zu keinen Wartezeiten führen, die die zeitliche Ausführung verlangsamen bzw. anderweitig verändern. Zum Anderen

muss sichergestellt sein, dass ausreichend Systemressourcen, wie z.B. CPU- und Netzwerklast, für die Anwendung bereit stehen und diese nicht durch das Testsystem beeinflusst wird.

Die Definition von Level 1 deckt sich mit den in Kapitel 2.3.2 vorgestellten Testarten in der allgemeinen Softwareentwicklung. Der weitere Verlauf der Arbeit fokussiert sich auf Tests in Level 2 und Level 3, wobei die eigentliche Funktionalität der ROS Nodes durch den Einsatz eines Testframeworks nicht beeinflusst werden darf.

Aus diesem Kapitel werden die folgenden Ziele und Anforderungen abgeleitet (vergleiche Tabelle 3.1 und Tabelle 3.2):

⇒ **Anforderung AF-11:** Umsetzung des Testframeworks mit minimaler Beeinflussung des zu testenden Systems durch die Ausführung und Auswertung von Tests

2.3.3 Testframeworks in der Servicerobotik

Für die Servicerobotik existieren verschiedene Testframeworks, die sich auf verschiedene Aspekte der Qualitätssicherung spezialisiert haben. Diese sind u.a. Robot Unit Testing (Bihlmaier & Wörn 2014), Rostest (Open Robotics 2020f), ROS Buildfarm (Open Robotics 2020e), Buildbot ROS (Ferguson 2014) und ROS-Industrial CI (ROS-Industrial Consortium 2020a). Weiterhin kommen Methoden aus der allgemeinen Softwareentwicklung zum Einsatz wie z.B. Style Guides und Code Reviews. Tabelle 2.3 zeigt eine Übersicht und Abgrenzung von existierenden Testframeworks, die im ROS Umfeld genutzt werden. Viele der existierenden Testframeworks setzen auf eine qualitative Analyse, bei der lediglich ein binäres Testergebnis (bestanden oder nicht bestanden) an den Entwickler gemeldet wird. Viele der Testframeworks unterstützen *simulation-in-the-loop* Tests, wobei nur wenige Testframeworks Unterstützung für *hardware-in-the-loop* Tests anbieten. Insbesondere können nur durch *hardware-in-the-loop* Tests unerwartete Testfälle abgedeckt und die Robustheit des Gesamtsystems gesteigert werden.

In Tabelle 2.3 ist zu beobachten, dass kein existierendes Testframeworks quantitative Analyse und Benchmarking mit Unterstützung von *hardware-in-the-loop* Tests anbietet (siehe rote Markierung in Tabelle 2.3). Wie in Kapitel 1.2 erläutert ist dies jedoch ein wichtiger Teil für die Beantwortung der Forschungsfrage dieser Arbeit:

Wie kann während der Entwicklungsphase einer Servicerobotik-Anwendung der Entwicklungs- und Testaufwand reduziert, die Testabdeckung gesteigert sowie die Robustheit des Gesamtsystems im späteren Betrieb erhöht werden?

Aus diesem Kapitel werden die folgenden Ziele und Anforderungen abgeleitet (vergleiche Tabelle 3.1 und Tabelle 3.2):

⇒ **Anforderung AF-2:** Bereitstellung eines Testwerkzeugs zur Unterstützung von Tests mit realer Hardware als *hardware-in-the-loop* Tests

⇒ **Anforderung AF-7:** Bereitstellung einer graphischen Visualisierung der quantitativen Testergebnisse

⇒ **Anforderung AF-10:** Unterstützung zum Vergleich von alternativen Komponenten bzw. Systemen (Benchmarking)

Testframework	Methode / Prozess	Statische Codeanalyse	Unittests	Integrationstests	Systemtests	*Hardware-in-the-loop* Unterstützung	*Simulation-in-the-loop* Unterstützung	Qualitative Analyse	Quantitative Analyse	Benchmarking Unterstützung	Funktionale Tests	Deployment Tests	Regression Tests	Vordefinierte Testbausteine	CI Integration	ROS Integration
Robot Unit Testing (Bihlmaier & Wörn 2014)	✓					✗			✗	✗						
ROS Style Guide (Open Robotics 2020g)	✓					✗			✗	✗						✓
Code Review	✓	✓				✗		✓	✗	✗						
ROS-I Rating (ROS-Industrial Consortium 2020c)	✓					✗			✗	✗						✓
ROS2 Quality Rating (Open Robotics 2020m)	✓					✗			✗	✗						✓
Linter (catkin_lint, ros_lint)		✓				✗		✓	✓	✓				✓	✓	✓
ROS Code Quality (Pitzer 2012)		✓				✗		✓	✓	✓					✓	✓
Rostest (Open Robotics 2020f)			✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓		✓	✓
ROS Buildfarm (Open Robotics 2020e)		✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓		✓	✓
Buildbot ROS (Ferguson 2014)		✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓		✓	✓
ROS-Industrial CI (ROS-Industrial Consortium 2020a)		✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓		✓	✓
ROS Gitlab CI (Lamoine 2020)		✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓		✓	✓
SR Buildtools (Shadow Robotics 2020)		✓	✓			✗		✓	✗	✗						✓
DDT (Reiser 2014)	✓			✓	✓	✓			✗	✗		✓				✓

Tabelle 2.3: *Gegenüberstellung existierender Testframeworks für die Servicerobotik.*

2.4 Qualitative und quantitative Metriken zur Bestimmung der Softwarequalität in der Servicerobotik

Im folgenden Kapitel werden Softwaremetriken vorgestellt, die für die Qualifizierung und/oder Quantifizierung von Testergebnissen verwendet werden können.

Eine Softwaremetrik ist nach IEEE Standard 1061 wie folgt definiert: „Eine Softwarequalitätsmetrik ist eine Funktion, die eine Softwareeinheit in einen Zahlenwert abbildet, welcher als Erfüllungsgrad einer Qualitätseigenschaft der Softwareeinheit interpretierbar ist.“ (IEEE 1061:1998). Diese Definition wird für den Kontext dieser Arbeit so übernommen.

Metriken können im Allgemeinen nach verschiedenen Eigenschaften klassifiziert werden, z.B. in qualitativ oder quantitativ, absolut oder relativ sowie direkt gemessen oder indirekt berechnet. Während sich für Bereiche wie z.B. in der Produktion weithin geltende Metriken wie z.B. Prozesszeit, Durchsatz, Energieverbrauch etc. etabliert haben, werden in der Servicerobotik kontinuierlich neue oder verfeinerte Metriken angewendet. Dieses Phänomen ist am deutlichsten bei Roboterwettbewerben wie RoboCup (Wisspeintner et al. 2009), DARPA Robotics Challenge (Krotkov et al. 2017) oder Amazon Picking Challenge (Hernandez et al. 2016) zu beobachten, bei denen die Regeln und damit implementierten Metriken einer ständigen Änderung unterliegen, um den Fortschritten auf diesem Gebiet Rechnung zu tragen.

Eine Liste an, für die Servicerobotik relevanten, Metriken findet sich in Nowak et al. (2010). Die Metriken werden dort in drei Kategorien unterteilt:

- *Cost Metrics* (Metriken, die die Kosten für die Zielerreichung beschreiben): z.B. Rechenzeit, Ausführungszeit, Reaktionszeit, Ressourcenaufwand
- *Utility Metrics* (Metriken, die die Art und Weise der Zielerreichung beschreiben): z.B. Weglänge, Abstand zu Hindernissen, Trajektorienglätte, Zielgenauigkeit

- *Reliability Metrics* (Metriken, die den Grad der Zielerreichung beschreiben): z.B. Erfolgs-/Fehlerquote, Anzahl an Kollisionen, Robustheit in engen Passagen

Die Klassifizierung einer Metrik kann auch von ihrem Kontext und ihrer konkreten Definition abhängen. Metriken werden häufig kombiniert, um übergeordnete Eigenschaften wie Optimalität, Sicherheit oder Robustheit zu bewerten (Iossifidis et al. 2005). Mehrere mathematische Gleichungen zur Messung von Trajektorien sind in Iossifidis et al. (2005) dargestellt. In Aguirre (2007) und Geraerts (2006) werden Metriken für Bewegungsplaner untersucht. Muñoz et al. (2007) definiert Metriken für die Navigation mobiler Roboter. In der Multi-Anual-Roadmap 2020 (EuRobotics aisbl 2015) werden in Kapitel 5 übergeordnete Metriken für Teiltechnologien wie z.B. Systementwicklung, Mensch-Roboter-Interaktion, Mechatronik sowie den Bereichen Perzeption, Navigation und Kognition definiert.

Oft sind die Metriken oberflächlich definiert und nur beschreibend erläutert. Die Metriken müssen daher für ihre Messbarkeit auf die technische Ebene heruntergebrochen werden (siehe Kapitel 3.2).

Aus diesem Kapitel werden die folgenden Ziele und Anforderungen abgeleitet (vergleiche Tabelle 3.1 und Tabelle 3.2):

⇒ **Anforderung AF-6:** Unterstützung zur automatisierten Auswertung der Tests anhand von objektiven Metriken

2.5 Benchmarking in der Servicerobotik

(Technologie-) Benchmarking bezeichnet die vergleichende Analyse von Ergebnissen oder Prozessen mit einem festgelegten Vergleichsmaßstab (engl. *benchmark*). Bei einem Technologie-Benchmarking können u.a. die folgenden Gesichtspunkte zum Vergleich herangezogen werden: Ziel der Technologie oder des Prozesses, Funktionsweise, Leistungsfähigkeit, Stabilität, Material, Erfahrungswerte, Kosten, Kapazitäten, u.v.m. Für diese Arbeit bezieht sich Benchmarking auf den

Vergleich der Funktionsweise mehrerer Lösungen, welcher anhand der Auswertung von objektiven Metriken bewertet wird.

In Zillich (2012) wird festgestellt, dass es schwierig ist, Roboter bzw. Robotersoftware miteinander zu vergleichen, da viele Roboterdemonstratoren beeindruckende, aber oft sehr eingeschränkte bzw. spezialisierte Fähigkeiten zeigen und formuliert deshalb einen Turing-ähnlichen Test für Roboter, bei dem ein Roboter zum Einen autonom agiert, zum Anderen durch einen menschlichen Teleoperator gesteuert wird, der dieselben sensorischen Eingaben hat wie der Roboter. Diese Art von Tests setzen neben qualitativer Metriken vor allem auf einen Vergleich anhand von quantitativen Metriken voraus.

In der Servicerobotik wurde die Notwendigkeit wiederholbarer quantitativer Benchmarks erkannt (EuRobotics aisbl 2015; Dillmann 2004; Iossifidis et al. 2005; Madhavan et al. 2009a; Madhavan et al. 2009b) und führte z.B. zu den Leistungsmetriken für intelligente Systeme (PerMIS) (Madhavan & Messina 2006). Einige Benchmarks konzentrieren sich jeweils nur auf eine Funktion wie z.B. Pfadplanung (Calisi et al. 2008; Baltes 2000), Hindernisvermeidung (Jimenez et al. 2006), Navigation und Kartierung (Balaguer et al. 2007; Ceriani et al. 2009), visual Servoing (Cervera 2006), Manipulation (Grunwald et al. 2008; Moll et al. 2015) oder soziale Interaktion (Tsui et al. 2012; Henkel et al. 2012).

Eine Übersicht über existierende Benchmarks für mobile Roboter aus Hägele & Pfeiffer (2007) zeigt Abbildung 2.7. Für die Einordnung der Benchmarks werden diese anhand der Kriterien Genauigkeit (engl. *precision of benchmark*) und Komplexität (engl. *aggregation of functions*) bewertet. Bei Genauigkeit wird dabei nicht immer anhand objektiver Kriterien ausgewertet. Bei manchen Tests wird der Vergleich auf Basis rein subjektiver Einschätzungen über Vergleichswettkämpfe erstellt. Bei der Komplexität werden Tests auf Komponentenebene, Systemebene und Tests, bei denen das Robotersystem mit seiner Umgebung (z.B. Menschen) interagiert unterschieden. Es zeigt sich, dass vor allem auf dem Gebiet von hoher Genauigkeit und hoher Komplexität Anwendungsgebiete für Benchmarks fehlen.

Bei allen Benchmarkingansätzen ist zu beobachten, dass Benchmarkingvergleiche auf Gesamtsystemebene schwer untereinander vergleichbar sind, vor allem

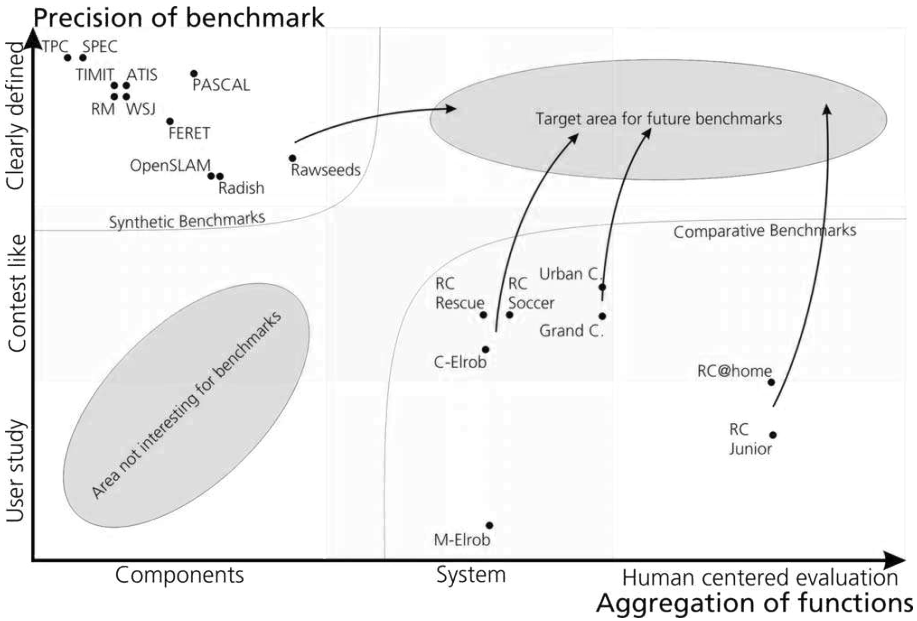


Abbildung 2.7: *Übersicht existierender Benchmarks für mobile Roboter. (Quelle: Hägele & Pfeiffer 2007)*

weil unterschiedliche Sensoren, Aktoren, Architekturen und Implementierungen vorliegen. Aus diesem Grund wird das Benchmarking meist auf Teiltechnologien bzw. Komponentenebene wie z.B. Navigation, Manipulation oder Perzeption zerlegt. Jede Komponente kann dabei wiederum anhand mehrerer Metriken wie z.B. Zeit, Pfadlänge oder Genauigkeit für einen Vergleich messbar gemacht werden.

Aus diesem Kapitel werden die folgenden Ziele und Anforderungen abgeleitet (vergleiche Tabelle 3.1 und Tabelle 3.2):

⇒ **Anforderung AF-10:** Unterstützung zum Vergleich von alternativen Komponenten bzw. Systemen (Benchmarking)

3 Ziele und Anforderungen an ein automatisiertes Test- und Evaluierungsframework für die Servicerobotik

In diesem Kapitel wird ein Testframework für die Softwareentwicklung in der Servicerobotik entworfen, welches den in Kapitel 2 aufgezeigten Stand der Technik erweitert. In Kapitel 3.1 werden Ziele und Anforderungen an ein automatisiertes Test- und Evaluierungsframework abgeleitet und anschließend in Kapitel 3.2 Metriken für Servicerobotik-Anwendungen definiert.

3.1 Ziele und daraus abgeleitete Anforderungen

Das in dieser Arbeit entworfene Testframework soll von den in Kapitel 2.2 genannten Softwarequalitätseigenschaften vor allem die Funktionalität, d.h. funktionale Korrektheit, sicherstellen. Das für die vorliegende Arbeit in Kapitel 1.3 formulierte Ziel lässt sich weiter in Ziele für die Servicerobotik-Lösung und Ziele für den Entwicklungsprozess unterteilen. Tabelle 3.1 formuliert Ziele für ein Testframework. Die Ziele lassen sich in Ziele für die Servicerobotik-Lösung (Z-L) und Ziele für den Entwicklungsprozess (Z-EP) gliedern. Aus diesen Zielen lassen sich die in Tabelle 3.2 dargestellten konkreten Anforderungen an ein automatisiertes Testframework ableiten. Die Umsetzung des in Kapitel 4 vorgestellten Testframework leitet sich aus diesen Anforderungen ab. Weiterhin

Ziel	Bezeichnung	Beschreibung
Z-L-1	Robustheit	Sicherstellung eines robusten Einsatzes in realen Umgebungen
Z-L-2	Wirtschaftlichkeit	Ermöglichen eines wirtschaftlicher Einsatzes in realen Use Cases
Z-EP-1	Testabdeckung	Erreichen einer hohen Testabdeckung
Z-EP-2	Test-automatisierung	Steigerung der Testautomatisierung
Z-EP-3	Dauer	Verkürzen der Dauer des Entwicklungsprozess
Z-EP-4	Kosten	Senkung der Kosten des Entwicklungsprozesses
Z-EP-5	Ressourceneinsatz	Reduktion des Einsatzes von Entwicklerressourcen für das Testen
Z-EP-6	Testerstellungs-aufwand	Minimierung des Aufwands zur Erstellung von Tests
Z-EP-7	Testdurchführungs-aufwand	Minimierung des Aufwands zur Durchführung von Tests
Z-EP-8	Transparenz	Erhöhung der Transparenz des Entwicklungsstandes

Tabelle 3.1: *Ziele für ein Testframework mit Zielen für die Servicerobotik-Lösung (Z-L) und Zielen für den Entwicklungsprozess (Z-EP).*

wird das vorgestellte Testframework in Kapitel 5 anhand dieser Anforderungen validiert.

Neben den in Tabelle 3.2 gelisteten Anforderungen gibt es weitere weiche Anforderungen, die ein Testframework erfüllen sollte:

Ein Testframework sollte unterschiedliche Hierarchien von Tests (Unittests, Integrationstests, Systemtests) unterstützen, sodass die komplette Testbandbreite nach Kapitel 2.3.2 damit abgedeckt werden kann. Weiterhin ist zu beachten, dass das Testframework von unterschiedlichen Entwicklern verwendet werden kann, d.h. sowohl Tests für die Komponentenentwicklung (Komponentenentwickler), als auch Tests für die Integration mehrere Komponenten in ein Gesamtsystem (Systemintegratoren). Dazu ist es notwendig, dass Entwickler das Testframework individuell konfigurieren und gegebenenfalls für ihre Bedürfnisse erweitern können.

Anforderung	Bezeichnung	Beschreibung
AF-1	Testbausteine	Bereitstellung vordefinierter Testbausteine
AF-2	*Hardware-in-the-loop*	Bereitstellung eines Testwerkzeugs zur Unterstützung von Tests mit realer Hardware als *hardware-in-the-loop* Tests
AF-3	*Simulation-in-the-loop*	Bereitstellung eines Testwerkzeugs zur Unterstützung von Tests in Simulation als *simulation-in-the-loop* Tests
AF-4	Teststufen	Unterstützung verschiedener Teststufen (Unit-, Integrations- und Systemtests)
AF-5	Testgenerierung	Bereitstellung eines Werkzeugs zur automatischen Generierung von Testsuiten
AF-6	Testauswertung	Unterstützung zur automatisierten Auswertung der Tests anhand von objektiven Metriken
AF-7	Ergebnisvisualisierung	Bereitstellung einer graphischen Visualisierung der quantitativen Testergebnisse
AF-8	Testautomatisierung	Integration in Infrastruktur zur Testautomatisierung
AF-9	TDD-Integration	Einbindung des Testwerkzeugs in den Entwicklungsprozess
AF-10	Benchmarking	Unterstützung zum Vergleich von alternativen Komponenten bzw. Systemen (Benchmarking)
AF-11	Minimale Beeinflussung	Umsetzung des Testframeworks mit minimaler Beeinflussung des zu testenden Systems durch die Ausführung und Auswertung von Tests
AF-12	ROS Integration	Integration in ROS zur einfache Nutzung im ROS Umfeld

Tabelle 3.2: *Anforderungen an ein Testframework.*

3.2 Metriken für ein Servicerobotik-Testframework

In diesem Kapitel werden Metriken aus Kapitel 2.4 für das Test- und Evaluierungsframework übernommen. Dabei werden die beschriebenen Metriken in konkrete Berechnungsvorschriften für den Kontext dieser Arbeit definiert. Die Metriken werden dabei in die Kategorien *Cost Metrics*, *Utility Metrics* und *Reliability Metrics* strukturiert (siehe Kapitel 2.4).

Weiterhin werden die Metriken nach der Art der Auswertung kategorisiert. Hier wird zwischen den folgenden Arten unterschieden:

- **Zeitpunkt (engl. *snap*):** Metriken werden zu einem konkreten Zeitpunkt angewendet und werten den Systemzustand zu diesem Zeitpunkt aus. Z.B. Bei Ende einer Aufgabe.
- **Zeitspanne (engl. *span*):** Metriken werden auf eine Zeitspanne angewendet, die durch zwei Zeitpunkte definiert ist. Z.B. von Start bis Ende einer Aufgabe.

Es gibt Metriken die sowohl als Snap-Metrik, als auch als Span-Metrik benutzt werden können. Beispiele für die konkrete Anwendung der Metriken finden sich in den verschiedenen Anwendungsbeispielen in Kapitel 5.

3.2.1 Metriken auf Betriebssystemebene

Metriken, die auf Betriebssystemebene erfasst werden sind in Tabelle 3.3 aufgelistet. Diese gelten jeweils nur für ein Referenzsystem, d.h. beim Vergleich dieser Metriken wird vorausgesetzt, dass die Anwendung auf dem gleichen Zielsystem ausgeführt wurden. Mithilfe dieser Metriken kann abgeschätzt werden, wie sich die Systemauslastung auf eine umgesetzte Anwendung auswirkt, also ob ab einer gewissen Systemauslastung die Funktionalität der Komponenten beeinflusst wird. Dies kann z.B. dann interessant sein, wenn die Hardwareressourcen beschränkt sind und sich mehrere Komponenten dieselben Hardwareressourcen teilen müssen. Ist dies der Fall könnte in einem mobilen Roboter z.B. die zusätz-

liche Ressourcenbelastung durch eine Perzeptionskomponente die Lokalisierung beeinflussen und damit einen Lokalisierungsverlust herbeiführen.

Metrik	Bezeichnung	Maß-einheit	Beschreibung	Berechnungs-vorschrift	Art (*snap, span*)	Kategorie (*cost, utility, reliability*)
M-OS-1	Maximale Prozessor-auslastung	[%]	Maximale Prozessorauslastung pro ROS Node	$\eta_{CPU,max}$	span	cost
M-OS-2	Durchschnittliche Prozessorauslastung	[%]	Durchschnittliche Prozessorauslastung pro ROS Node	$\eta_{CPU,avg}$	span	cost
M-OS-3	Maximale Arbeitsspeicher-auslastung	[kb]	Maximale Arbeitsspeicherbelegung pro ROS Node	$\eta_{RAM,max}$	span	cost
M-OS-4	Durchschnittliche Arbeitsspeicher-auslastung	[kb]	Durchschnittliche Arbeitsspeicher-belegung pro ROS Node	$\eta_{RAM,avg}$	span	cost
M-OS-5	Maximale Festplatten-zugriffsrate	[kb/s]	Maximale Festplattenzugriffsrate pro ROS Node	$\eta_{diskIO,max}$	span	cost
M-OS-6	Durchschnittliche Festplattenzugriffsrate	[kb/s]	Durchschnittliche Festplattenzugriffs-rate pro ROS Node	$\eta_{diskIO,avg}$	span	cost
M-OS-7	Maximale Netzwerk-zugriffsrate	[kb/s]	Maximale Netzwerkzugriffsrate pro ROS Node	$\eta_{ethIO,max}$	span	cost
M-OS-8	Durchschnittliche Netzwerkzugriffsrate	[kb/s]	Durchschnittliche Netzwerkzugriffs-rate pro ROS Node	$\eta_{ethIO,avg}$	span	cost

Tabelle 3.3: *Betriebssystemspezifische Metriken.*

3.2.2 Robotikspezifische Metriken

In Tabelle 3.4 ist eine Auswahl an robotikspezifischen Metriken aus Kapitel 2.4 aufgelistet. Die Metriken beobachten dabei die Ausführung der Anwendung "von außen", z.B. über Informationen die über ROS Topics oder ROS TF abgefragt werden können, d.h. es werden keine komponenten-internen Informationen benötigt. Dies hat den Vorteil, dass die gleichen Metriken auf verschiedene Komponenten bzw. verschieden parametrisierte Systeme angewendet werden können und setzen damit eine wichtige Voraussetzung für eine spätere objektive Vergleichbarkeit im Rahmen von Benchmarkingtests um. Eine Ausnahme davon bildet die Metrik Benutzerergebnis, welche die Möglichkeit bietet benutzerspezifische Werte auszuwerten, die direkt über die Testapplikation befüllt werden können.

Die Auswahl der Metriken deckt die Auswertungen, der im Rahmen dieser Arbeit verwendeten Anwendungsbeispiele aus Kapitel 5 ab. Für weitere Anwendungen könnte die Definition weiterer spezifischer Metriken notwendig werden.

Metrik	Bezeichnung	Maß-einheit	Beschreibung	Berechnungsvorschrift	Art (*snap, span*)	Kategorie (*cost, utility, reliability*)
M-R-1	Zeitdauer	[s]	Zeitdauer zwischen Anfang und Ende eines Testblocks.	$\eta_{duration} = t_{stop} - t_{start}$	span	utility
M-R-2	Publikations-frequenz	[hz]	Die Publikationsfrequenz eines Topics, gemessen als die Anzahl der eingegangenen Nachrichten pro Zeitintervall.	$\eta_{publish\_rate} = \frac{N_{messages}}{t_{stop} - t_{start}}$	span	utility
M-R-3	Hindernisabstand	[m] oder [rad]	Minimaler Abstand zwischen Hindernissen und Robotergliedern.	$\eta_{obstacle\_distance} = \min(\|robot\_link - obstacle\|)$	snap, span	reliability
M-R-4	TF-Abstand	[m] oder [rad]	Kartesischer Abstand zwischen zwei Koordinatensystemen (TF-Frames).	$\eta_{tf\_distance} = \|root\_link - child\_link\|$	snap, span	utility
M-R-5	TF-Pfadlänge	[m] oder [rad]	Zurückgelegter kartesischer Weg eines Roboterglieds in Bezug auf ein Referenzkoordinatensystem.	$\eta_{path\_length} = \sum_{i=t_{start}}^{t_{stop}} (\|pos_i - pos_{i-1}\|)$	span	utility
M-R-6	TF-Pfad-geschwindigkeit	[m/s] oder [rad/s]	Durchschnittliche Geschwindigkeit entlang eines kartesischen Wegs.	$\eta_{path\_velocity} = \sum_{i=t_{start}}^{t_{stop}} \frac{\|pos_i - pos_{i-1}\|}{t_i - t_{i-1}}$	span	utility

M-R-7	Gelenkwinkelweg	[m] oder [rad]	Zurückgelegter Weg eines Gelenks in Gelenkkoordinaten.	$\eta_{joint\_length} = \sum_{i=t_{start}}^{t_{stop}}(\|joint\_pos_i - joint\_pos_{i-1}\|)$	span	utility
M-R-8	Gelenk-geschwindigkeit	[m/s] oder [rad/s]	Durchschnittliche Geschwindigkeit eines Roboterglieds in Bezug auf ein Referenzkoordinatensystem.	$\eta_{joint\_velocity} = \sum_{i=t_{start}}^{t_{stop}} \frac{pos_i - pos_{i-1}}{t_i - t_{i-1}}$	span	utility
M-R-9	API	[bool]	Prüft ob die im Komponentenmodell spezifizierten Schnittstellen eines Nodes (Publisher, Subscriber, Service Server, Action Server) bereitgestellt werden und deren Typenspezifikation übereinstimmt.	$\eta_{api} = API_{meassured} \equiv API_{specified}$	snap	utility
M-R-10	Nachrichteninhalt	[bool]	Prüft ob die empfangene Nachricht (Topic, Service, Action) der spezifizierten Nachricht entspricht.	$\eta_{api} = message\_content_{meassured} \equiv message\_content_{specified}$	snap	utility
M-R-11	Benutzerergebnis	[any]	Ergebnis ist frei setzbar aus Anwendung.	$\eta_{user\_result} = user\ defined$	snap, span	cost, utility, reliability

Tabelle 3.4: *Robotikspezifische Metriken.*

4 Konzeption und Umsetzung des Test- und Evaluierungsframeworks ATF

Im folgenden Kapitel wird das im Rahmen dieser Arbeit entwickelte Automatisierte Test- und Evaluierungsframework (engl. *Automated Test Framework*, ATF) vorgestellt. Grundlage für die Entwicklung sind die Ziele und Anforderungen aus Kapitel 3.1. Das ATF ist ein einfach in bestehende Anwendungen zu integrierendes Framework, das es ermöglicht Servicerobotik-Anwendungen zu testen und verschiedene Varianten miteinander zu vergleichen (Benchmarking). Weiter kann ATF entwicklungsbegleitend in einer TDD-Entwicklungsumgebung eingesetzt werden um den Fortschritt im Entwicklungsprozess zu überwachen.

Anmerkung: Teile des im folgenden vorgestellten automatisierten Test- und Evaluierungsframeworks ATF wurden bereits veröffentlicht. Bubeck et al. (2012) stellt Best Practices bei der Systemintegration und in der verteilen Entwicklung am Beispiel Care-O-bot dar. Für die Verbesserung der Softwareportierbarkeit wird in Weißhardt et al. (2014) die Aufteilung in roboter-, umgebungs- und anwendungsspezifische Tests vorgestellt. In Weißhardt & Koehler (2016) wird erstmals ATF als Werkzeug für die Evaluierung und Benchmarking von Servicerobotik-Anwendungen eingeführt. In Albus (2018) wird das ATF verwendet, um Parametersets für Navigationsverfahren zu analysieren und zu optimieren.

4.1 Übersicht Architektur des Testframeworks

Das ATF besteht, wie in Abbildung 4.1 zu sehen, aus mehreren Bestandteilen, welche die verschiedenen Schritte eines Tests abbilden: von der Testgenerierung über die Datenaufnahme und Auswertung bis hin zur Visualisierung der Ergebnisse. ATF ist Open Source auf Github (Weißhardt 2021) unter der Apache 2.0 Lizenz veröffentlicht und in der Programmiersprache Python implementiert sowie tief in ROS integriert, sodass es ohne großen Aufwand in bestehenden ROS Systemen verwendet werden kann. Ein ATF Testsystem kann hierbei neben reinen Softwarekomponenten auch aus Hardwarekomponenten, als sogenannte *hardware-in-the-loop* Tests, und/oder aus simulierten Komponenten als sogenannte *simulation-in-the-loop* Tests bestehen. Der Anwendungsbereich des ATF umfasst eine Vielzahl an Komponenten und Anwendungen, was insbesondere Benchmarking zwischen den verschiedenen Systemen erlaubt.

Abbildung 4.1 zeigt eine Übersicht über die wesentlichen Bestandteile des ATF. Nach der Erstellung einer Testanwendung bzw. eines Testsystems und der Konfiguration des ATF gliedert sich ein Testdurchlauf im wesentlichen in vier Schritte:

- Schritt 1: Definition und Generierung von Tests (Kapitel 4.2)
- Schritt 2: Ausführen der Anwendung mit Aufnahme von Testdaten (Kapitel 4.3)
- Schritt 3: Analyse der Testdaten (Kapitel 4.4)
- Schritt 4: Visualisierung und Vergleich der Testergebnisse (Kapitel 4.5)

Zuerst werden, ausgehend von der spezifizierten Kombinationsmatrix der zu variierenden Komponenten und Parameter, ausführbare Tests generiert. Danach werden die einzelnen Tests durchlaufen und dabei jeweils alle zur Auswertung erforderlichen Daten extrahiert und in ein ROS Bag gespeichert. Nach der Datenaufnahme folgt die Analyse der Testdaten anhand von Metriken (siehe Tabelle 3.3 und 3.4). In diesem Schritt werden die Testergebnisse für die

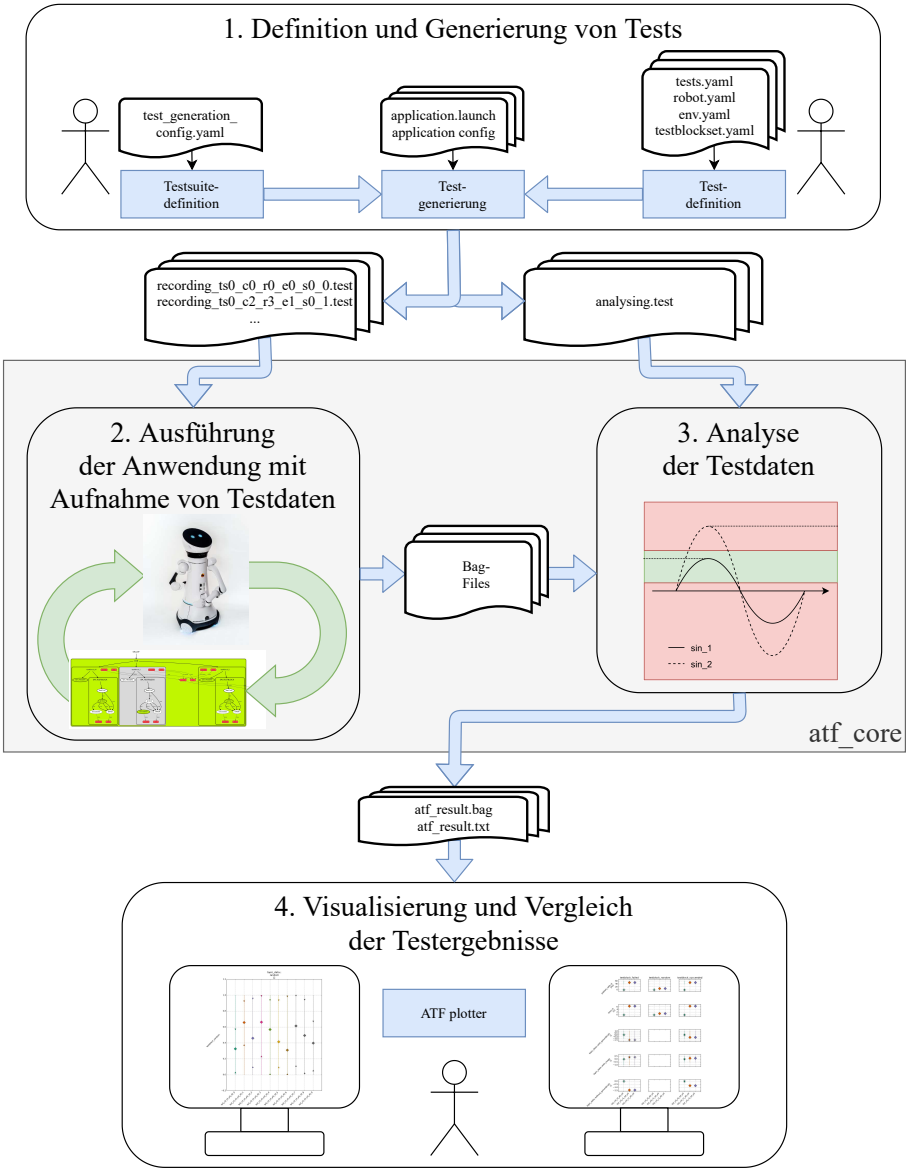


Abbildung 4.1: *ATF Architektur mit den vier Schritten eines Testdurchlaufs. Schritt 1: Definition und Generierung von Tests, Schritt 2: Ausführen der Anwendung mit Aufnahme von Testdaten, Schritt 3: Analyse der Testdaten und Schritt 4: Visualisierung und Vergleich der Testergebnisse.*

anschließende Auswertung und Visualisierung gespeichert und optional in einem Cloud-Speicher archiviert. Die Schritte laufen dabei größtenteils automatisch ab, lediglich die Testdefinition in Schritt 1 muss vom Benutzer definiert werden.

4.2 Schritt 1: Definition und Generierung von Tests

Das Ergebnis der Testgenerierung sind ausführbare Tests auf Basis von Rostests. Für die Generierung der Tests ist es notwendig, dass ein Testsystem bereitgestellt wird, die zu variierenden Testparameter definiert werden sowie diejenigen Kombinationen spezifiziert werden, die getestet werden sollen.

Quellcode 4.1 zeigt die Verzeichnisstruktur eines ATF ROS Pakets. Darin enthalten ist die vollständige Deployment-Konfiguration `application.launch`, die eigentliche Anwendung `application.py` sowie alle Konfigurationsparameter im Verzeichnis `atf`. Die Definition der zu testenden Tests findet sich in `test_generation_config.yaml` und wird unterschieden zwischen roboterspezifischen, umgebungsspezifischen sowie roboter- und umgebungsunabhängigen Parametern (siehe Kapitel 4.2.2).

```
1  -- atf
2  |     | -- envs
3  |     |     | -- env1.yaml
4  |     | -- robots
5  |     |     | -- robot1.yaml
6  |     |     | -- robot2.yaml
7  |     | -- testblocksets
8  |     |     | -- testblockset1.yaml
9  |     |     | -- testblockset2.yaml
10 |     | -- test_generation_config.yaml
11 |     | -- tests.yaml
12 |     |     | -- test1.yaml
13 |     |     | -- test2.yaml
14 | -- CMakeLists.txt
15 | -- launch
16 |     | -- application.launch
17 | -- package.xml
18 | -- scripts
19       | -- application.py
```

Quellcode 4.1: *Verzeichnisstruktur eines ATF ROS Pakets.*

4.2.1 Bereitstellen des Testsystems

Ein Testsystem bezeichnet eine lauffähige Deployment-Konfiguration eines ROS Systems in Form einer ROS Launch Datei. Die ROS Launch Datei enthält typischerweise mehrere Nodes, die über entsprechende Zuordnungen der Kommunikationskanäle miteinander kommunizieren, sowie Konfigurationsparameter. Die Nodes können dabei auch auf mehrere Rechner verteilt sein. Die eigentliche Anwendung ergibt sich aus dem Zusammenspiel der Nodes, wobei die Anwendungssteuerung selbst ein eigenständiger Node sein kann.

Falls nicht alle Komponenten eines zu testenden Systems verfügbar sind, da z.B. notwendige Hardware oder externe Schnittstellenanbindungen fehlen, können diese Komponenten durch ein oder mehrere Simulations- oder Mockupkomponenten sowie durch Komponenten mit zuvor aufgezeichnete Daten der realen Hardwarekomponenten (ROS Bags) wie z.B. Sensordaten von Sensorkomponenten ersetzt werden. Abbildung 4.2 zeigt ein beispielhaftes Testsystem, indem die verschiedenen Ersatzmöglichkeiten gezeigt werden.

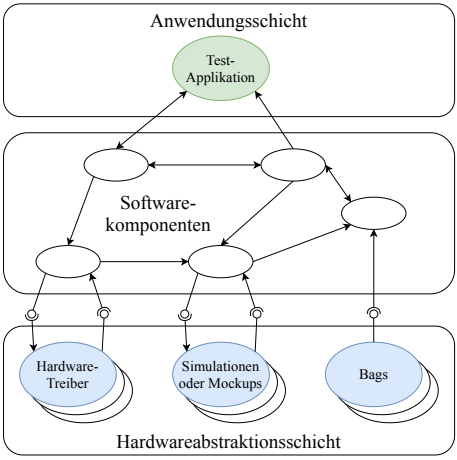


Abbildung 4.2: *Testsystem mit Test-Anwendung. Nodes mit Kommunikationskanälen für Test-Anwendung (grün), Komponenten zur Hardwareabstraktion (blau) wie z.B. Hardwaretreiber, Simulationen, Mockups oder Komponenten, die aufgezeichnete Daten über Bags bereitstellen sowie weiteren Softwarekomponenten (weiß).*

Ersetzen von Komponenten durch Mockup-Komponenten

Beim Ersatz durch eine Mockup-Komponente wird die fehlende Komponente durch eine leere Hülle mit den gleichen Schnittstellen ersetzt. Die Mockup-Komponente verhält sich dabei anders als die ursprüngliche Komponenten, implementiert jedoch die jeweiligen Schnittstellen und kann somit als Bindeglied zu anderen Komponenten eingesetzt werden. Ein Beispiel für eine Mockup-Komponente ist, wie in Abbildung 4.3 dargestellt, z.B. eine Komponente, die eine Datenbankabfrage realisiert falls die reale Datenbank nicht verfügbar ist.

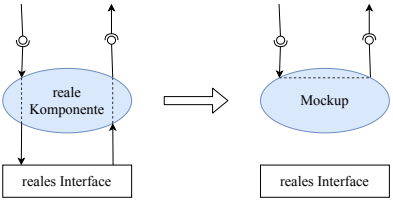


Abbildung 4.3: *Ersatz einer Komponente zur Datenbankabfrage durch eine Mockup-Komponente mit gleichen Schnittstellen.*

Ersetzen von Komponenten durch zuvor aufgezeichnete Daten

Sind zuvor Daten der ursprünglichen Komponenten aufgezeichnet worden, so können diese während des Testdurchlaufs abgespielt werden und ersetzten dadurch die fehlende Komponente. Mit dieser Methode können Komponenten ersetzt werden, die keine bidirektionale Interaktion mit den restlichen Komponenten des Systems haben, sondern reine Datenquellen sind.

Typische Beispiele hierfür sind Sensoren (z.B. Laserscanner oder Farbkameras), die ihre Daten als Topic bereitstellen und somit von einer Objekterkennungskomponente verwendet werden können. Die Verwendung von Sensordaten im Testsystem durch zuvor real aufgezeichnete Daten hat den Vorteil, dass Eigenschaften der Umgebung bzw. des Sensors wie z.B. unterschiedliche Lichtverhältnisse, Sensorrauschen und Sichtbereiche mit in den Test einfließen.

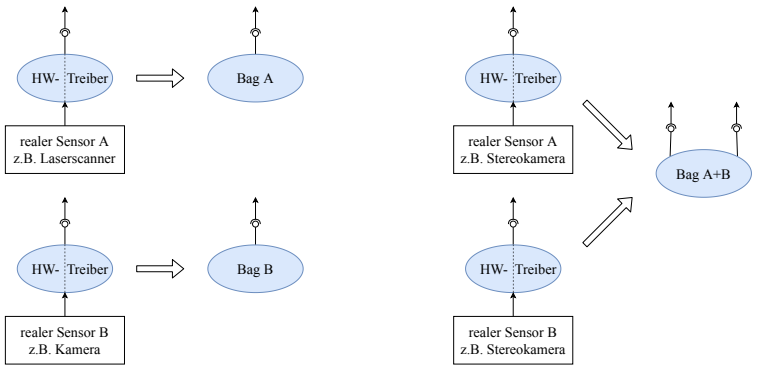


Abbildung 4.4: *Ersatz von Komponenten durch aufgezeichneten Daten. (a) Ersatz mehrerer unabhängiger Sensoren (z.B. Laserscanner und Kamera) durch mehrere Komponenten mit aufgezeichneten Daten, (b) Ersatz mehrerer synchronisierter Sensoren (z.B. Stereokamera) durch eine Komponenten mit aufgezeichneten Daten*

Abbildung 4.4 zeigt zwei verschiedene Einsatzmöglichkeiten für den Ersatz einer Komponenten durch aufgezeichnete Daten. In Abbildung 4.4(a) sind zwei unabhängig voneinander eingesetzte Sensoren (ein Laserscanner und eine Kamera) zu sehen, die durch zwei Komponenten mit aufgezeichneten Daten ersetzt werden. Im Gegensatz zu Abbildung 4.4(a) sind in Abbildung 4.4(b) die zwei Sensoren durch eine Komponente mit aufgezeichneten Daten ersetzt. Dies hat den Vorteil, dass die beiden Sensoren synchronisierte Daten liefern können, also z.B. zwei Farbkameras, die in unterschiedlichen Winkeln auf eine Szene gerichtet sind und somit zwar unterschiedliche Daten aber doch zusammengehörende Bildpaare liefern (z.B. Stereokamerasystem).

Ersetzen von Komponenten durch reale (*hardware-in-the-loop*) oder simulierte (*simulation-in-the-loop*) Hardwarekomponenten

Steht eine Hardwarekomponente in Interaktion zu anderen Komponenten des Testsystems oder der Umgebung wie in Abbildung 2.3 gezeigt, so ist der Einsatz von aufgezeichneten Daten nicht möglich, da diese erst durch die Eingaben der

anderen Komponenten oder der Veränderung der Umgebung sinnvolle Daten erzeugen. Ein typisches Beispiel hierfür ist ein Laserscanner am Fahrwerk eines mobilen Roboters, der je nach Bewegung des Fahrwerks in der Umgebung andere Sensordaten liefert. In diesem Fall können auch reale oder simulierte Hardwarekomponenten als Teil des Testsystems aufgenommen werden.

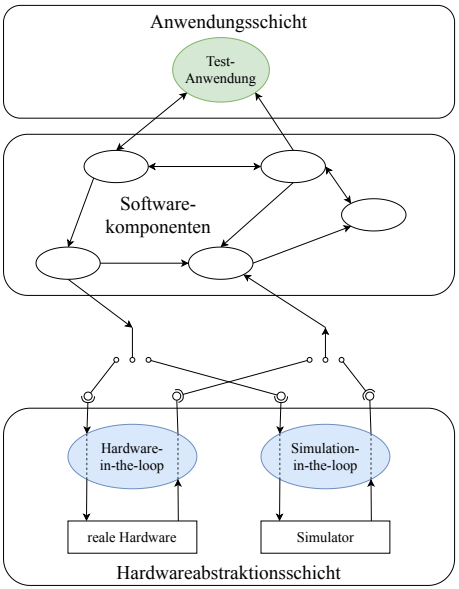


Abbildung 4.5: *Einsatz von Hardware-in-the-loop oder Simulation-in-the-loop.*

Abbildung 4.5 zeigt ein Testsystem bei dem eine Hardwarekomponente als *hardware-in-the-loop* oder *simulation-in-the-loop* verwendet wird.

4.2.2 Definition der Testparameter

Die Anzahl der zu variierenden Testparameter hat großen Einfluss auf die Testabdeckung und auf die Anzahl der zu generierenden Tests. Die zu variierenden Testparameter können sehr unterschiedlich sein. Tabelle 4.1 listet einige beispielhafte Parameter auf.

Parameter	Gruppierung
Maximalgeschwindigkeit einer mobilen Basis	rob
Beschleunigung einer mobilen Basis	rob
Radius der Räder einer mobilen Basis	rob
Anzahl der verwendeten Sensoren	rob
Platzierung von Sensoren	rob
Kinematischer Aufbau eines Armes	rob
Zielpunkt für mobile Basis	env
Anzahl der Hindernisse in Umgebung	env
Größe der Hindernisse in Umgebung	env
Algorithmus zur Pfadplanung	config
Schrittweite des numerischen Lösers eines Pfadplaners	config
u.v.m	rob/env/config

Tabelle 4.1: Beispiele von zu variierenden Testparametern und deren Zuordnung als roboterspezifischer Parameter (rob), umgebungsspezifischer Parameter (env) sowie roboter- und umgebungsunabhängiger Parameter (config).

Die Testparameter lassen sich wie in Tabelle 4.1 gezeigt in drei Kategorien gruppieren: in roboterabhängige Parameter (rob), in umgebungsabhängige Parameter (env) und in allgemeine Konfigurationsparameter (config), die weder vom eingesetzten Roboter noch von seiner Einsatzumgebung abhängen.

Aufstellen der Kombinationsmatrix mit zu variierenden Testparametern

Ein Testergebnis kann durch die Modifikation eines einzelnen Parameters völlig verändert werden. Deshalb ist es ohne weitere Annahmen erst einmal nötig für alle Permutationen von Testparametern zu testen, was eine faktorielle Testanzahl ergibt (vergleiche Kapitel 2.3.2 Testabdeckung und statistische Versuchsplanung). Abbildung 4.6 zeigt die aus Gleichung 4.1 errechnete Anzahl an Permutationen N_P in Bezug auf die Anzahl m der zu variierenden Parameter $param_i$, wenn jeder diese Parameter in N_{param_i} Variationen auftaucht.

$$N_P = \prod_{i=1}^{m} N_{param_i} \tag{4.1}$$

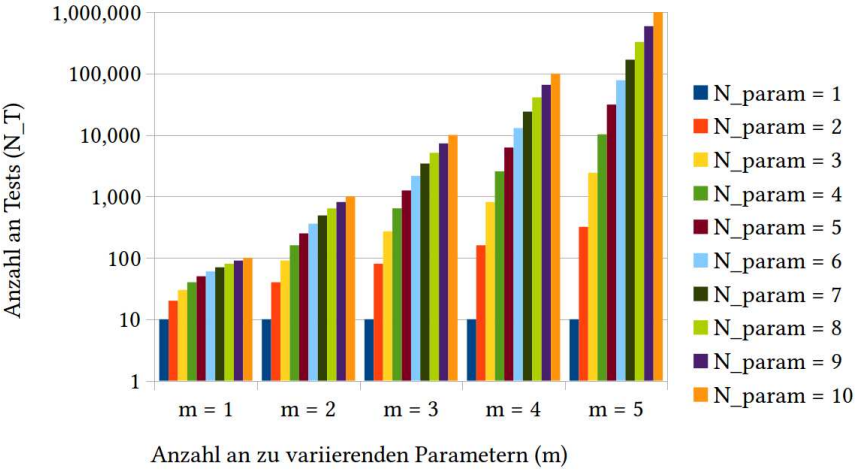


Abbildung 4.6: *Anstieg der Permutationen für* $m = 1..5$ *und* $N_{param} = 1..10$ *und* $N_{wdh} = 10$ *(logarithmische Skala).*

Um statistische Schwankungen beim Durchlaufen der Tests zu evaluieren sollte jede Permutation N_{wdh}-fach wiederholt getestet werden. Daraus ergibt sich die Anzahl N_T der Tests zu

$$N_T = N_P \cdot N_{wdh} = \prod_{i=1}^{m} N_{param_i} \cdot N_{wdh} \tag{4.2}$$

Für eine überschaubare Anzahl von $m = 5$ an zu variierenden Testparametern und jeweils $N_{param} = 10$ Variationen der einzelnen Parameter und $N_{wdh} = 10$ Wiederholungen explodiert die Anzahl der Permutationen, wie Abbildung 4.6 veranschaulicht, bereits auf 100.000 und damit die Anzahl der Tests bereits auf 1.000.000. Selbst mit automatisierten Tests kann es dabei vorkommen, dass entweder die Testressourcen erschöpft werden oder die Zeit bis zum Erhalt eines Testergebnisses zu lange wird, um damit in einem TDD Entwicklungszyklus arbeiten zu können. Demnach ist eine sinnvolle Einschränkung der Anzahl der Tests notwendig.

Einschränken der Anzahl der Tests durch Testsuites

Um die Vielzahl der Tests auf ein sinnvolles Maß beschränken zu können gibt es in ATF die Möglichkeit sogenannte Testsuites zu definieren. Eine Testsuite besteht aus einer Kombination von roboterspezifischen Parametern, umgebungsspezifischen Parametern sowie roboter- und umgebungsunabhängigen Parametern wie in Quellcode 4.2 gezeigt.

```
testsuites:
  # this will generate 2 (=1*2*1*1) test cases and 20 tests in total (=2*10)
  - tests:
      - test1
    robots:
      - robot1
      - robot2
    envs:
      - env1
    testblocksets:
      - testblockset1
    repetitions: 10

  # this will generate 12 (=2*3*2*1) test cases and 120 tests in total (=12*10)
  - tests:
      - test1
      - test2
    robots:
      - robot1
      - robot2
      - robot3
    robot_envs:
      - env1
      - env2
    testblocksets:
      - tetblockset1
    repetitions: 10
```

Quellcode 4.2: *Definition von Testsuites und Begrenzung der auszuführenden Tests durch Einschränken der Permutationen.*

Werden die Parameter wie in Tabelle 4.1 gezeigt in einer Testsuite zu Parametersets gruppiert, so errechnet sich mit der Anzahl N_{rob} der roboterspezifischen Parametersets, der Anzahl N_{env} der umgebungsspezifischen Parametersets und der Anzahl N_{config} der roboter- und umgebungsunabhängigen Parameter die Anzahl $N_{P_{suite}}$ der Permutationen einer Testsuite zu

$$N_{P_{suite}} = N_{rob} \cdot N_{env} \cdot N_{config} \tag{4.3}$$

und die Anzahl $N_{T_{suite}}$ der Tests einer Testsuite damit zu

$$N_{T_{suite}} = N_{P_{suite}} \cdot N_{wdh} = N_{rob} \cdot N_{env} \cdot N_{config} \cdot N_{wdh} \tag{4.4}$$

Nach Gleichung 4.4 hängt die Anzahl der Tests einer Testsuite nun nicht mehr von der Gesamtzahl der zu variierenden Parameter im Testsystem ab, was die Anzahl der auszuführenden Tests reduziert. Für das Beispiel in Quellcode 4.2 ergeben sich mit der Einführung von Testsuites statt 1.000.000 auszuführender Tests nunmehr noch 20 bzw. 120 durchzuführende Tests.

ATF ermöglicht durch die Definition von Testsuiten den Einsatz geeigneter Methoden zur Begrenzung der durchzuführenden Tests. Um trotz der reduzierten Anzahl an Tests aussagekräftige Ergebnisse zu erhalten, kann die Definition der Testsuites z.B. mithilfe statistischer Versuchsplanung erfolgen (siehe Kapitel 2.3.2 *Design of Experiments*). Statistische Versuchsplanung definiert Methoden, die bei der Testeinschränkung unterstützen, um negative Effekte der Testeinschränkungen zu reduzieren.

4.3 Schritt 2: Ausführen der Anwendung mit Aufnahme von Testdaten

Nach der Konfiguration der Testparameter und der Generierung von Tests folgt die Ausführung der Tests mit der Aufnahme von Testdaten zur späteren Auswertung.

4.3.1 Steuerung des Tests durch Testblöcke

Für die Aufnahme von Testdaten wird die Anwendung in sogenannte Testblöcke aufgeteilt, die beim Ausführen der Anwendung die Aufnahme von Testdaten steuern. Jeder Testblock folgt hierbei einer internen Statemachine, die den Lebenszyklus eines Testblocks steuert. In der Anwendung kann ein Testblock über `atf.start()` gestartet und über `atf.stop()` gestoppt werden. Weiterhin kann ein Testblock über `atf.pause()` pausiert und über `atf.purge()` geleert

werden. Abbildung 4.7 visualisiert die zugrundeliegende Statemachine. Ein Testblock kann dabei die folgenden Zustände haben:

- `INACTIVE` **(Startzustand):** Der Testblock ist initialisiert, aber wartet in einem inaktiven Zustand darauf gestartet zu werden.
- `ACTIVE`**:** Der Testblock wird ausgeführt und nimmt Daten auf.
- `PAUSED`**:** Der Testblock ist pausiert. Es werden keine weitere Daten aufgenommen.
- `PURGED`**:** Der Testblock wurde zurückgesetzt, d.h. alle bisher aufgenommenen Daten werden ignoriert. Der Testblock wartet darauf wieder gestartet zu werden.
- `SUCCEEDED` **(Endzustand):** Der Testblock wurde erfolgreich ausgeführt.
- `ERROR` **(Endzustand):** Beim Ausführen des Testblocks wurde ein Fehler detektiert.

In einer Anwendung können dabei mehrere, auch überlappende Testblöcke definiert werden, die jeweils gestartet, pausiert, geleert und gestoppt werden können. Die Aufnahme läuft dabei parallel im Hintergrund ab, sodass die zeitliche Ausführung der Anwendung nicht beeinflusst wird. Dies gilt, solange genügen Systemressourcen (z.B. CPU-Auslastung und Festplattenzugriffe) zum Parallelisieren verfügbar sind. Um dies zu überprüfen, können u.a. OS-Metriken aus Tabelle 3.3 eingesetzt werden (siehe Kapitel 4.3.2).

Das Beispiel ATF Paket aus Kapitel 4.2 (Quellcode 4.1) enthält die eigentliche Anwendung mit den integrierten Testblöcken in `application.py`. Quellcode 4.3 zeigt die Aufteilung in Testblöcke und deren Integration in die Anwendung. Abbildung 4.8 zeigt die parallele Ausführung von mehreren Testblöcken der Anwendung.

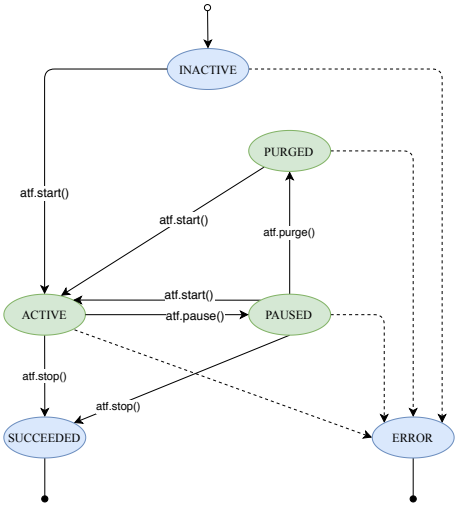


Abbildung 4.7: *Statemachine zur Steuerung eines Testblocks. Ein Testblock kann dabei die Zustände* `INACTIVE` *(Startzustand),* `ACTIVE`*,* `PAUSED`*,* `PURGED`*,* `SUCCEEDED` *(Endzustand) und* `ERROR` *(Endzustand) einnehmen.*

```
1  import atf_core
2
3  class Application:
4      def __init__(self):
5          # Initialize the ATF class
6          self.atf = atf_core.ATF()
7
8      def execute(self):
9          # You can call start/pause/purge/stop for each testblock
10         # during the execution of your app
11
12         self.atf.start("testblock_all")
13         self.atf.start("testblock_1")
14
15         # Do something
16
17         self.atf.stop("testblock_1")
18         self.atf.start("testblock_2")
19
20         # Do something else
21
22         self.atf.stop("testblock_2")
23         self.atf.stop("testblock_all")
24
25         # Finally we'll have to call shutdown() to tell
26         # the ATF to stop all recordings and wrap up
27         self.atf.shutdown()
```

Quellcode 4.3: *Beispiel zur Integration des ATF in Python-Anwendung.*

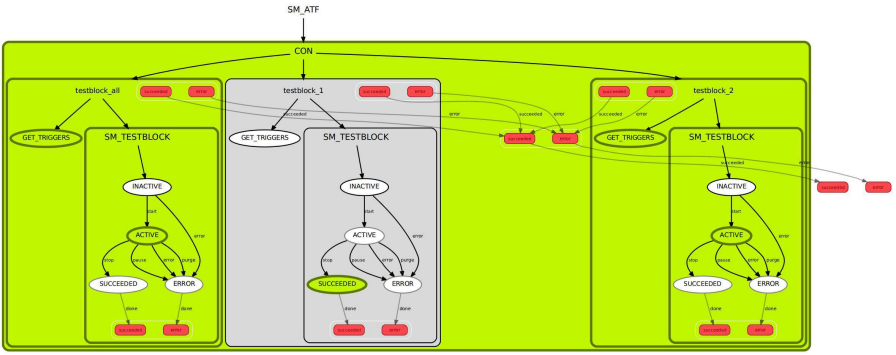


Abbildung 4.8: *Parallele Ausführung von mehreren Testblöcken in einer Anwendung. In diesem Beispiel ist die Anwendung in drei Testblöcke* `testblock_1`, `testblock_2` *sowie* `testblock_all` *unterteilt (siehe Quellcode 4.3). Aktive Zustände sind in grün hervorgehoben.*

4.3.2 Definition von anzuwendenden Metriken

Für jeden Testblock können beliebig viele Metriken aus Tabelle 3.3 und Tabelle 3.4 definiert werden. Quellcode 4.4 zeigt ein Beispiel zur Verwendung mehrerer Metriken in einem Testblock. Die Plugin-Architektur des ATF sorgt automatisch dafür, dass die Module zur Aufnahme von Daten für die jeweiligen Metriken geladen werden. Die Daten werden zusammen mit den Informationen über den Verlauf der Testblöcke in ein ROS Bag gespeichert. Somit sind alle notwendigen Informationen die zur späteren Auswertung benötigt werden an zentraler Stelle zusammengeführt und können dem nächsten Schritt, der Auswertung der Daten, übergeben werden.

4.3.3 Hinterlegen von Groundtruthdaten als Sollwerte

An selber Stelle, an der die anzuwendenden Metriken spezifiziert werden, können optional auch Groundtruthdaten hinterlegt werden. Nach Auswerten der Metriken werden die gemessenen Werte mit diesen Sollwerten verglichen und entsprechend das Testergebnis bestimmt. Weiter werden die Groundtruthdaten

```
testblock_1:
  time: []
  tf_length_translation:
    - root_frame: link1
      measured_frame: link2
    - root_frame: link1
      measured_frame: link3
  publish_rate:
    - topic: topic1
    - topic: topic2
    - topic: topic3
testblock_2:
  time: []
  path_length:
    - root_frame: link1
      measured_frame: link2
  interface:
    - node: publisher1
      publishers:
        - [topic1, std_msgs/String]
    - node: publisher2
      publishers:
        - [topic2, std_msgs/String]
testblock_all:
  time: []
```

Quellcode 4.4: *Beispiel zur Verwendung mehrerer Metriken in einem Testblock.*

zu Visualisierungszwecken verwendet. Quellcode 4.5 zeigt die Hinterlegung von Groundtruthdaten pro Metrik.

```
testblock_1:
  time:
    - groundtruth: 3
      groundtruth_epsilon: 0.5
  tf_length_translation:
    - root_frame: link1
      measured_frame: link2
      groundtruth: 12.00
      groundtruth_epsilon: 1.0
  publish_rate:
    - topic: topic1
      groundtruth: 10
      groundtruth_epsilon: 1
```

Quellcode 4.5: *Beispiel zur Hinterlegung von Groundtruthdaten als Sollwerte.*

4.4 Schritt 3: Analyse der Testdaten

Nach der Aufnahme der Daten folgt die Analyse der Daten anhand der definierten Metriken. Da alle Daten während der Ausführung der Anwendung in ein ROS Bag gespeichert werden, kann die Analyse komplett getrennt von der Aufnahme

erfolgen. Die damit ermöglichte Parallelisierung der Tests kann wie in Kapitel 4.6 beschrieben für die bessere Lastverteilung bei der Testautomatisierung genutzt werden. Während die Datenaufnahme mit Hardware meist in Realzeit parallel zur Anwendung erfolgen muss, kann die Analyse der Daten durch ein schnelleres Abspielen des ROS Bag beschleunigt werden. Besonders bei vielen langen Tests lässt sich dadurch die Gesamtzeit der Tests verringern. Durch die Parallelisierung der Auswertung als auch die beschleunigte Analyse der Daten erfolgt der Analyseschritt deutlich schneller als die Datenaufnahme.

4.4.1 Auswertung von nummerischem Testergebnis anhand von Metriken

Während der Analysephase werden die aufgenommenen Daten aus dem ROS Bag extrahiert und den jeweiligen Metrik-Plugins bereitgestellt. Die Metrik-Plugins ermitteln für jeden Testblock nummerische Ergebnisse.

Hierfür berechnet jede Metrik nummerische Ergebniswerte für jeden Zeitschritt und speichert diese als metrikspezifische Datenreihe $m_{series} = [m_{t_{start}}, m_{t_1}, m_{t_2}, ..., m_{t_{end}}]$ ab. Die Anzahl der Daten einer Datenreihe ist $N = len(m_{series})$. Aus der metrikspezifischen Datenreihe wird ein nummerisches Testergebnis m_{res} abgeleitet. Dabei wird abhängig von der Konfiguration der Metrik eine der folgenden Auswertungsmethoden genutzt (vergleiche Kapitel 3.2):

- **`SNAP`:** Es wird der letzte Wert der Datenreihe als Ergebnis verwendet, d.h. der Messwert beim Stoppen des Testblocks.

$$m_{res} = m_{snap} = m_{series}[N] \tag{4.5}$$

- **`SPAN_MEAN`:** Aus allen Datenpunkten der Datenreihe wird ein Durchschnittswert gebildet.

$$m_{res} = m_{mean} = \frac{\sum_{i=1}^{N} m_{series}[i]}{N} \tag{4.6}$$

- **SPAN_STDDEV:** Aus allen Datenpunkten der Datenreihe wird die Standardabweichung berechnet.

$$m_{res} = m_{stddev} = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(m_{series}[i] - m_{mean})^2} \tag{4.7}$$

- **SPAN_MIN:** Es wird der kleinste Wert der Datenreihe verwendet.

$$m_{res} = m_{min} = \min(m_{series}[i]) \qquad i \in 1..N \tag{4.8}$$

- **SPAN_ABSMIN:** Es wird der betragsmäßig kleinste Wert der Datenreihe verwendet.

$$m_{res} = m_{absmin} = \min(|m_{series}[i]|) \qquad i \in 1..N \tag{4.9}$$

- **SPAN_MAX:** Es wird der größte Wert der Datenreihe verwendet.

$$m_{res} = m_{max} = \max(m_{series}[i]) \qquad i \in 1..N \tag{4.10}$$

- **SPAN_ABSMAX:** Es wird der betragsmäßig größte Wert der Datenreihe verwendet.

$$m_{res} = m_{absmax} = \max(|m_{series}[i]|) \qquad i \in 1..N \tag{4.11}$$

Abbildung 4.9 zeigt beispielhaft zwei Sinus-Datenreihen mit Amplitude von $1,0$ (`sin_1`, durchgezogene Linie) und mit einer Amplitude von $2,0$ (`sin_2`, gestrichelte Linie). Weiterhin ist die Auswertung der Datenreihen zu unterschiedlichen nummerischen Ergebnissen, basierend auf der gewählten Auswertungsmethode, veranschaulicht.

Mithilfe der Gleichungen 4.5 bis 4.11 wird jeder Metrik eines Testblocks ein nummerisches Ergebnis zugewiesen.

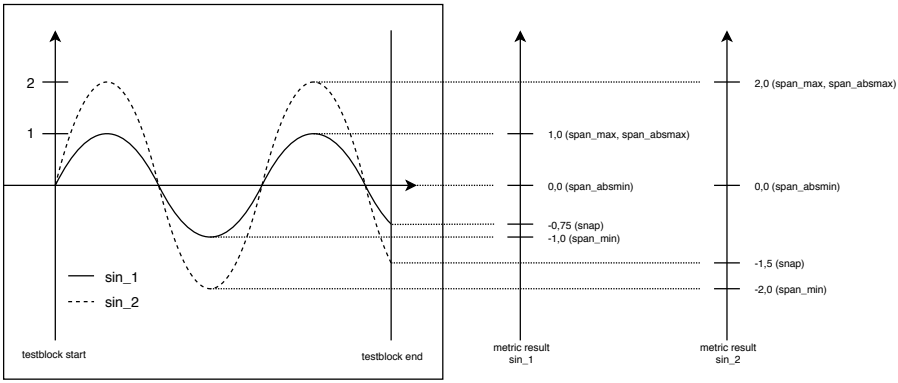


Abbildung 4.9: *Beispielhafte nummerische Auswertung für Datenreihe. Es sind zwei Sinus-Datenreihen abgebildet: mit einer Amplitude von* $1,0$ *(`sin_1`, durchgezogene Linie) und einer Amplitude von* $2,0$ *(`sin_2`, gestrichelte Linie). Basierend auf der Auswertungsmethode in der Konfiguration wird die Datenreihe zu einem nummerischen Testergebnis ausgewertet.*

4.4.2 Bestimmung eines binären Testergebnisses

Aus den nummerischen Ergebnissen der einzelnen Metriken wird im Folgenden ein binäres Ergebnis abgeleitet, sodass Tests als erfolgreich oder nicht erfolgreich bewertet werden können. Dieses binäre Testergebnis kann z.B. in einem CI-Setup genutzt werden, um Benachrichtigungen oder ggf. anschließende Schritte einzuleiten.

Falls Groundtruthdaten konfiguriert sind, wird das ermittelte nummerische Ergebnis aus den Gleichungen 4.5 bis 4.11 mit den in den Testblocks definierten Groundtruthdaten $m_{g_{data}}$ abgeglichen und damit für jede Metrik ein binäres Ergebnis M_{res} bestimmt. Hierbei wird vergleichen, ob das nummerische Ergebnis innerhalb eines Zielkorridors von $\pm m_{g_{epsilon}}$ um $m_{g_{data}}$ liegt (vergleiche Abbildung 4.10).

$$M_{res} = (m_{g_{data}} - m_{g_{epsilon}} \leq m_{res} \leq m_{g_{data}} + m_{g_{epsilon}}) \tag{4.12}$$

Sind keine Groundtruthdaten verfügbar, wird das Ergebnis der Metrik als erfolgreich gewertet, d.h. $M_{res} = true$.

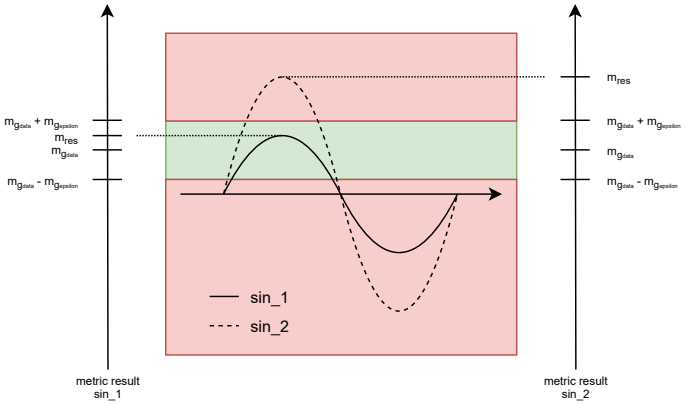


Abbildung 4.10: *Beispielhafte binäre Auswertung für Metrik mit Auswertungsmethode* ***SPAN_MAX****. In diesem Beispiel liegt das nummerische Ergebnis m_{res} von* ***sin_1*** *innerhalb des Zielkorridors von $m_{g_{data}} \pm m_{g_{epsilon}}$ und führt daher zu einem positiven binären Ergebnis, wohingegen das nummerische Ergebnis von* ***sin_2*** *außerhalb des Zielkorridors liegt, was zu einem negativen Ergebnis führt.*

Somit kann für jede Metrik eines Testblocks ein binäres Ergebnis M_j bestimmt werden. Daraus kann wiederum mit Gleichung 4.13 für jeden Testblock ein binäres Ergebnis TB_k und mit Gleichung 4.14 für jeden Test ein binäres Ergebnis T_l abgeleitet werden (Verknüpfung durch logische *Konjunktion* $\wedge$).

$$TB_k = (M_1 \wedge M_2 \wedge M_3 \wedge ... \wedge M_j) \qquad j = len(M) \tag{4.13}$$

$$T_l = (TB_1 \wedge TB_2 \wedge TB_3 \wedge ... \wedge TB_k) \qquad k = len(TB) \tag{4.14}$$

Aus allen binären Testergebnissen T_l wird weiterhin das binäre Ergebnis einer Testsuite TS_m mit

$$TS_m = (T_1 \wedge T_2 \wedge T_3 \wedge ... \wedge T_l) \qquad l = len(T) \tag{4.15}$$

und das binäre ATF Gesamtergebnis ATF mit

$$ATF = (TS_1 \wedge TS_2 \wedge TS_3 \wedge ... \wedge TS_m) \qquad m = len(TS) \tag{4.16}$$

bestimmt.

Ein ATF Testdurchlauf wird somit nur als erfolgreich gewertet, wenn alle Metriken aller Testblöcke, alle Testblöcke aller Tests, alle Tests einer Testsuite und alle Testsuites des ATF Durchlaufs erfolgreich sind.

4.4.3 Zusammenführen von mehrfachen Testwiederholungen

ATF bietet die Möglichkeit einzelne Tests mehrfach zu Wiederholen. Dies kann z.B. dann genutzt werden, wenn die Testbedingungen nicht wiederholbar oder Teile des Systems nicht deterministisch sind, sodass Testergebnisse nicht exakt reproduziert werden können, sondern von Durchlauf zu Durchlauf Schwankungen unterliegen. Um statistische Schwankungen für die einzelnen Tests ermitteln zu können wird eine Testkonfiguration mehrfach ausgeführt und die einzelnen Ergebnisse der Metriken anschließend zusammengefasst. Dies geschieht sowohl für die nummerischen als auch für die binären Ergebnisse der Metriken.

Hierbei werden für jede Metrik aus den nummerischen Ergebnissen m_{res} jeder Wiederholung, bei N_{wdh} Wiederholungen, ein aggregierter Minimalwert $m_{agg_{min}}$ mit

$$m_{agg_{min}} = \min(m_{res}[i]) \qquad i \in 1..N_{wdh} \tag{4.17}$$

und aggregierter Maximalwert $m_{agg_{max}}$ mit

$$m_{agg_{max}} = \max(m_{res}[i]) \qquad i \in 1..N_{wdh} \tag{4.18}$$

extrahiert, sowie ein aggregierter Mittelwert $m_{agg_{mean}}$ mit

$$m_{agg_{mean}} = \frac{\sum_{i=1}^{N_{wdh}} m_{res}[i]}{N_{wdh}} \tag{4.19}$$

und eine aggregierte Standardabweichung $m_{agg_{stddev}}$ mit

$$m_{agg_{stddev}} = \sqrt{\frac{1}{N_{wdh}-1} \sum_{i=1}^{N_{wdh}} (m_{res}[i] - m_{agg_{mean}})^2} \tag{4.20}$$

gebildet.

Für die Bestimmung eines binären Gesamtergebnisses M_{agg} wird geprüft, ob jedes binäre Einzelergebnis M_{res} jeder Wiederholung als positiv ausgewertet werden konnte. Nur wenn alle Einzelergebnisse positiv sind, ist das Gesamtergebnis positiv (Verknüpfung durch logische *Konjunktion* $\wedge$).

$$M_{agg} = M_{res_1} \wedge M_{res_2} \wedge M_{res_3} \wedge ... \wedge M_{res_i} \qquad i = len(N_{wdh}) \tag{4.21}$$

Analog zu den Gleichungen 4.13 bis 4.16 wird aus den einzelnen binären Ergebnissen der Metriken, Testblöcke, Tests und Testsuites das aggregierte ATF Gesamtergebnis bestimmt. Somit muss jede einzelne Wiederholung eines Tests die Groundtruth-Anforderungen erfüllen um den aggregierten Gesamttest zu bestehen.

4.4.4 Archivierung der Testergebnisse

Nach Analyse der Daten können die ATF Ergebnisse sowie die Rohdaten optional in einen Cloud-Speicher hochgeladen werden. Dies hat den Vorteil, dass der anschließende Schritt der Visualisierung wiederum unabhängig von der zur Analyse verwendeten Infrastruktur (z.B. CI-Service) stattfinden kann und eine Archivierung der Testergebnisse möglich ist.

4.5 Schritt 4: Visualisierung und Vergleich der Testergebnisse

ATF bietet die Möglichkeit die Testergebnisse über den ATF Plotter visuell anzuzeigen. Dabei gibt es die Möglichkeit die Ergebnisse einzelner oder aggregierter Tests zu visualisieren sowie eine Darstellung, mit der die Ergebnisse mehrerer Tests untereinander verglichen werden können (Benchmarking).

4.5.1 Visualisierung einzelner Tests

Für einzelne Tests können sowohl die binären Ergebnisse als auch die nummerischen Ergebnisse der Metriken dargestellt werden. Abbildung 4.11 zeigt die Visualisierung der Ergebnisse der Metriken eines einzelnen Tests (`ts0_c0_r0_e0_s0_0`). Dabei werden die nummerischen Ergebnisse mithilfe der Position einer Raute eingezeichnet. Basiert die Auswertung des nummerischen Ergebnisses auf einer Datenreihe (z.B. `SPAN`-Metrik), so werden deren Maximal- und Minimalwerte mit kleinen Dreiecken dargestellt. Die binären Ergebnisse sind mithilfe der Füllung der Raute gekennzeichnet: Ist die Raute gefüllt, ist das entsprechende Ergebnis positiv, bei leerer Raute negativ. Der dazugehörige Zielkorridor (siehe Abbildung 4.10) ist als Balken eingezeichnet. Besteht ein Test aus mehreren Testblöcken und dieser wiederum aus mehreren Metriken, so werden diese als Matrix dargestellt, um diese untereinander vergleichen zu können. Mit dieser Darstellung lassen sich die kompletten nummerischen und binären Ergebnisse für alle zu einem Test gehörigen Testblöcke und Metriken visualisieren.

4.5.2 Visualisierung von mehrfachen Testwiederholungen

Mehrfache Wiederholungen eines Tests können in einer Darstellung abgebildet werden. Abbildung 4.12(a) zeigt eine Reihe von Wiederholungen des gleichen Tests (`ts0_c0_r0_e0_s0_0` bis `ts0_c0_r0_e0_s0_10`) in unterschiedlichen Farben. Zu sehen ist, dass die dargestellten nummerischen Ergebnisse der einzelnen Tests schwanken. In Abbildung 4.12(b) sind die Daten der einzelnen Tests, wie in Kapitel 4.4.3 beschrieben, zu einem aggregierten Testergebnis zusammengefasst. Hierbei wird die gleiche Darstellung wie für einen Einzeltest gewählt, bei dem das aggregierte nummerische und binäre Ergebnis als Raute sowie die Maximal- und Minimalwerte der einzelnen Tests wieder als Dreiecke dargestellt sind. Da es sich um Wiederholungen des gleichen Tests handelt, ist der Zielkorridor für alle Tests der gleiche und ist wiederum als Balken dargestellt.

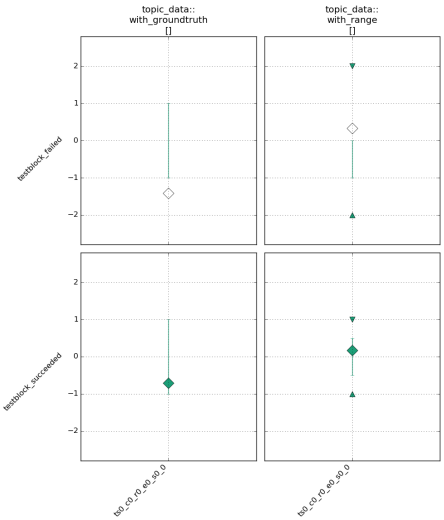
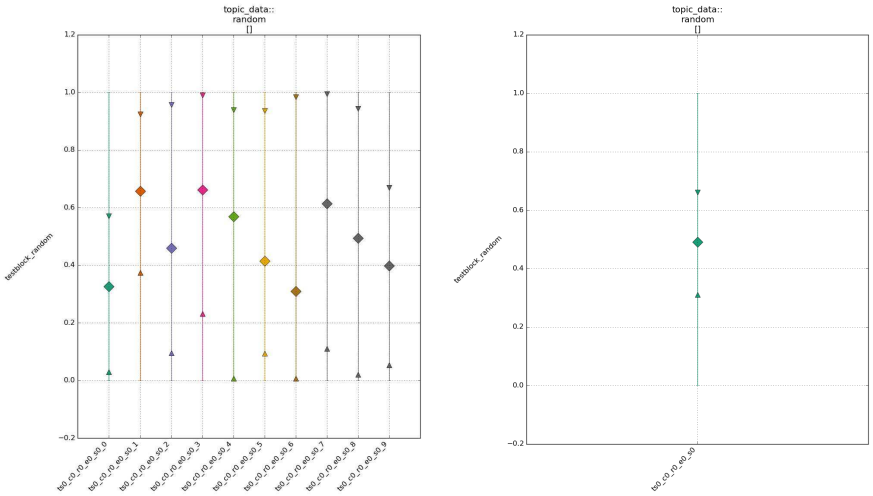


Abbildung 4.11: *Beispiel zur Visualisierung von Testergebnissen. Es können alle Metriken und Testblöcke eines Tests als Matrix dargestellt werden. Die nummerischen und binären Ergebnisse sind als Raute, der Zielkorridor als Balken und die Maximal- und Minimalwerte einer Datenreihe als Dreiecke eingezeichnet. Leere Raute: negatives Testergebnis (obere Reihe), voll Raute: positives Testergebnis (untere Reihe), Balken: Zielkorridor zur Bestimmung des binären Ergebnisses, Dreiecke: Maximal- und Minimalwerte einer Datenreihe (rechte Spalte).*



(a) (b)

Abbildung 4.12: *Beispiel zur Visualisierung von aggregierten Testergebnissen. (a) Visualisierung einer Reihe von Wiederholungen des gleichen Tests. (b) Darstellung des aggregierten Testergebnis. Hierbei wird die gleiche Darstellung wie für einen Einzeltest gewählt, bei dem das aggregierte nummerische und binäre Ergebnis als Raute, der Zielkorridor als Balken sowie die Maximal- und Minimalwerte der einzelnen Tests wieder als Dreiecke dargestellt sind. Leere/volle Raute: nummerisches und binäres aggregiertes Testergebnis, Balken: Zielkorridor zur Bestimmung des binären Ergebnisses, Dreiecke: Maximal- und Minimalwerte der einzelnen Tests.*

4.5.3 Benchmarkingdarstellung von vergleichbaren Tests

Der ATF Plotter bietet neben der Visualisierung einzelner und aggregierter Testergebnisse weiter die Möglichkeit mehrere unterschiedliche Tests zu visualisieren. Abbildung 4.13 zeigt mehrere Darstellungsmöglichkeiten von Tests, um diese miteinander zu vergleichen. Zum Einen kann die Matrixdarstellung über Spalten von gleichen Metriken oder Testblöcken gruppieren, zum Anderen können mehrere Tests in einem Matrixelement dargestellt werden. Weiterhin können Filter angewendet werden, um die Anzahl der darzustellenden Metriken, Testblöcke und Tests einzugrenzen. Mithilfe dieser Ansichten unterstützt ATF den Entwickler beim Benchmarking, um so basierend auf den Auswertungen geeigneter Metriken die für den jeweiligen Anwendungsfall beste Lösung zu identifizieren. Mit den Darstellungen aus Abbildung 4.13 können Tests miteinander vergleichen werden, die sich aus der Kombinationsmatrix aus Kapitel 4.2.2 *Aufstellen der Kombinationsmatrix mit zu variierenden Testparametern* ergeben. Dies kann z.B. der Vergleich unterschiedlicher Tests sein, bei denen roboter- oder umgebungsspezifische Parameter oder Konfigurationsparameter verändert wurden und daher zu unterschiedlichen Testergebnissen führen. Zur besseren Übersicht können auch nur Teile der Kombinationsmatrix mit entsprechenden Filtern visualisiert werden.

Ein Testergebnis wird zwar eindeutig durch ein binäres Gesamtergebnis beschrieben, jedoch kann es hilfreich sein mehrere Tests nicht nur aufgrund ihres binäres Gesamtergebnisses miteinander zu vergleichen, sondern anhand eines nummerischen Wertes. Mithilfe eines nummerischen Wertes für das Gesamtergebnis können die Tests in eine Reihenfolge geordnet werden und dadurch eine Bestenliste erstellt werden wie es für Benchmarkingvergleiche üblich ist. Dazu werden die verschiedenen nummerischen Einzelergebnisse m_{res} eines Tests für alle Metriken aus allen Testblöcken normiert und mithilfe von Gewichtungsfaktoren g nach Gleichung 4.22 zu einer wiederum normierten Gesamtsumme des Tests t_i addiert.

$$t_i = \frac{\sum g[i] \| m_{res}[i] \|}{\sum g[i]} \tag{4.22}$$

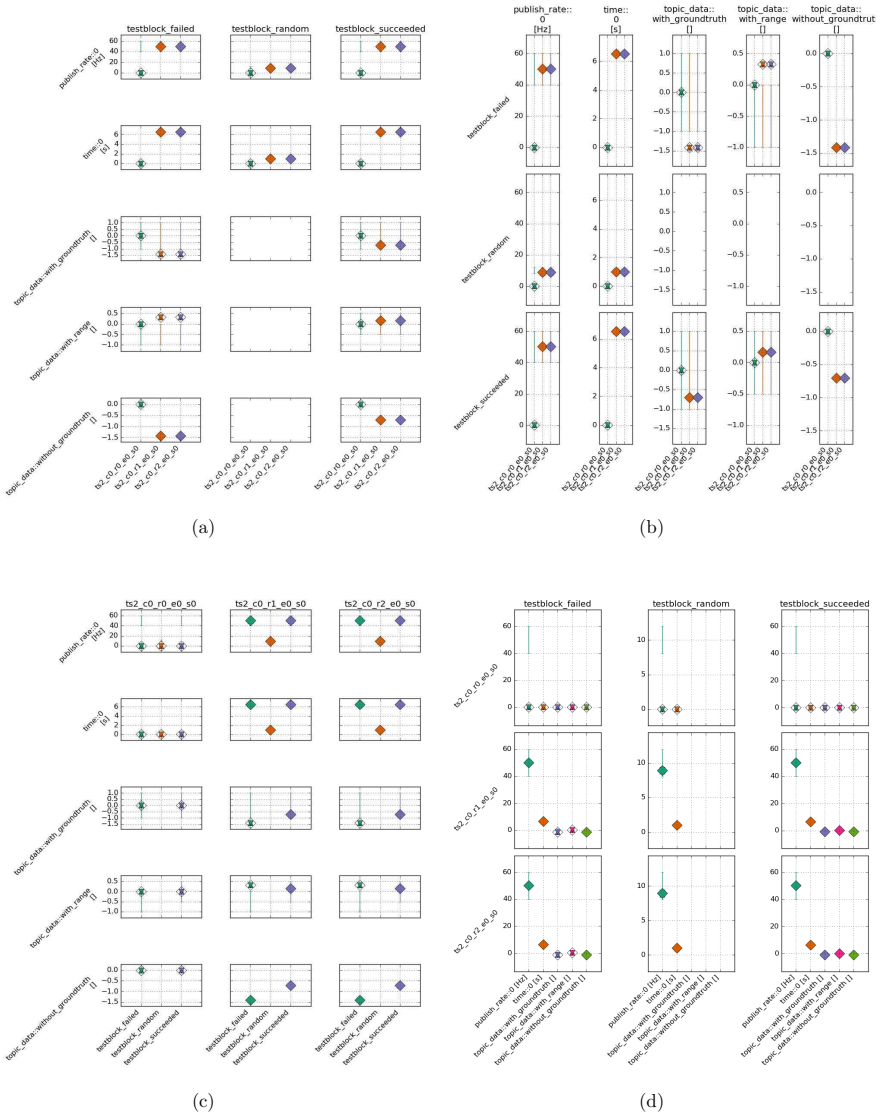


Abbildung 4.13: *Beispiele für Matrixdarstellungen für Benchmarking. (a) Gruppierung nach Metriken (Reihen) und Testblöcken (Spalten). (b) Gruppierung nach Testblöcken (Reihen) und Metriken (Spalten). (c) Gruppierung nach Metriken (Reihen) und Tests (Spalten). (d) Gruppierung nach Tests (Reihen) und Testblöcken (Spalten).*

4.6 Testautomatisierung

ATF bietet die Möglichkeit die Datenaufnahme bei der Ausführung der Tests zu automatisieren. Die Automatisierung der Testausführung ist in ATF über ein CMake Makro abgebildet, welches die Schritte 1 Testgenerierung, 2 Aufnahme und 3 Analyse (siehe Kapitel 4.2 bis 4.4) durchführt. Die Einbindung des Makros in ein ROS Paket ist in Quellcode 4.6 zu sehen. Das CMake Makro definiert zusätzliche CMake Targets, die nun z.B. über `catkin run_tests` getriggert werden können. In Verbindung mit Automatisierungsskripten von ROS-Industrial CI kann ATF z.B. in Travis CI oder Github Actions automatisiert werden. Dies kann zum Einen genutzt werden, um ATF in einem TDD-Entwicklungsprozess zu nutzen und zum Anderen, um kontinuierlich den Funktionsstand von Komponenten zu visualisieren.

```
1  cmake_minimum_required(VERSION 2.8.3)
2  project(atf_test)
3
4  find_package(catkin REQUIRED)
5
6  catkin_package()
7
8  catkin_install_python(PROGRAMS scripts/application.py
9      DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
10
11 ##################
12 ## ATF Testing ##
13 ##################
14 if(CATKIN_ENABLE_TESTING)
15   find_package(atf_core REQUIRED)
16   atf_test(atf/test_generation_config.yaml)
17 endif()
```

Quellcode 4.6: *Einbindung von ATF in ROS Paket über CMake Makro.*

ATF wird in einem TDD-Entwicklungsprozess verwendet, um Veränderungen in Softwarekomponenten kontinuierlich mit den hinterlegten Groundtruthdaten abzugleichen. Hierbei werden, wie in Kapitel 4.4 dargestellt, die definierten Metriken ausgewertet und anschließend mit den hinterlegten Groundtruthdaten abgeglichen, um ein binäres Testergebnis zu bekommen. Abbildung 4.14 zeigt die Ergebnisvisualisierung eines ATF Testpakets in Travis CI. Zu sehen ist, dass für die einzelnen Entwicklungszweige (engl. *Branches*) das binäre Testergebnis dargestellt wird.

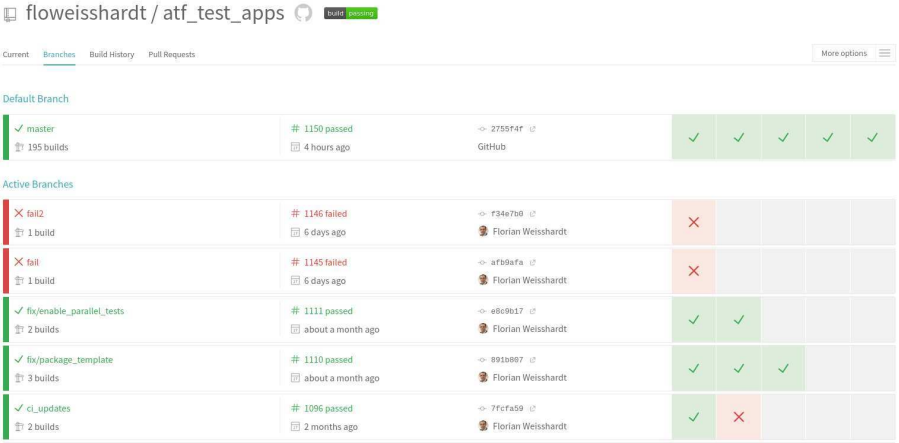


Abbildung 4.14: *Verwendung von ATF in einem TDD-Entwicklungsprozess mit Travis CI.*

Nicht-binäre Testergebnisse werden genutzt, um kontinuierlich den aktuellen Funktionsstand von Komponenten zu verfolgen. Hierbei werden die Testergebnisse des immer gleichen Tests über verschiedene Versionsstände der zu testenden Komponenten ausgewertet, sodass der Entwickler in einem Histogramm visualisieren kann, wie sich das Testsystem verändert hat. Die Daten für den Vergleich können über einen automatisierten CI-Prozess kontinuierlich generiert und mithilfe der Archivierung von Testergebnissen (siehe Kapitel 4.4.4) bereitgestellt werden.

Durch die Trennung des Testdurchlaufs in die in den Kapiteln 4.2 bis 4.5 beschriebenen Schritte lässt sich ein ATF Durchlauf sehr gut parallelisieren und z.B. auf einer Buildfarm in verschiedene Jobs aufteilen. Weiterhin entkoppelt es den zeitintensiven Schritt der Datenaufnahme, der oftmals in Realzeit zusammen mit der Anwendung ablaufen muss, und der Analyse, die nicht an die Realzeit gebunden ist und somit schneller erfolgen kann. Somit lässt sich auch bei lange dauernden Anwendungen die Gesamtzeit zur Ausführung der Tests reduzieren.

5 Evaluierung

In diesem Kapitel wird die Umsetzung des ATF und dessen Nutzung evaluiert. Hierzu wird in Kapitel 5.1 die Verwendung von ATF anhand von Anwendungsbeispielen vorgestellt und deren ATF Ergebnisse beleuchtet. Anschließend werden in Kapitel 5.2 die Ziele des ATF evaluiert und ATF gegenüber anderen Testframeworks abgegrenzt. In Kapitel 5.3 erfolgt eine Betrachtung der Nutzung und Verbreitung von ATF.

5.1 Anwendungsfälle des ATF

Im den folgenden Abschnitten werden einige Anwendungsfälle des ATF vorgestellt, um aufzuzeigen wie das ATF den Entwicklungsprozess bzw. die Servicerobotik-Lösung hinsichtlich der Ziele aus Kapitel 3.1 verbessert. Dazu wird für jedes Anwendungsbeispiel ein Steckbrief erstellt, indem der ATF Anwender und dessen Aufgabe sowie die Ausgangssituation des Anwendungsbeispiels und die Problemstellung des ATF Anwenders vorgestellt wird. Anschließend wird die jeweilige Verwendung des ATF erläutert, die ATF Ergebnisse dargestellt und eine Bewertung des Anwendungsfalls bezüglich der Anforderungen und Ziele des Testsystems in Anlehnung an Tabelle 3.2 durchgeführt.

5.1.1 Anwendungsbeispiel 1: Komponententest einer Perzeptionskomponente

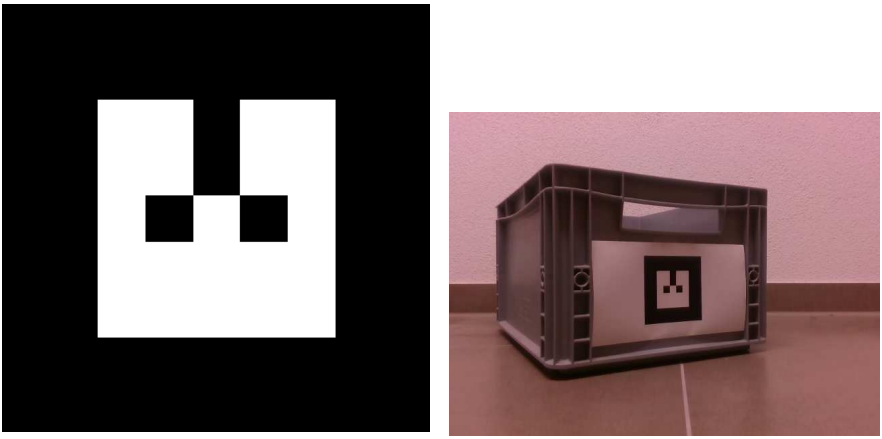
Steckbrief Anwendungsbeispiel 1

Anwender: Komponentenentwickler für Bildverarbeitungskomponente

Aufgabe: Der Komponentenentwickler hat die Aufgabe die 6D Pose einer Kiste mithilfe eines angebrachten Markers mit einer Farbkamera zu bestimmen.

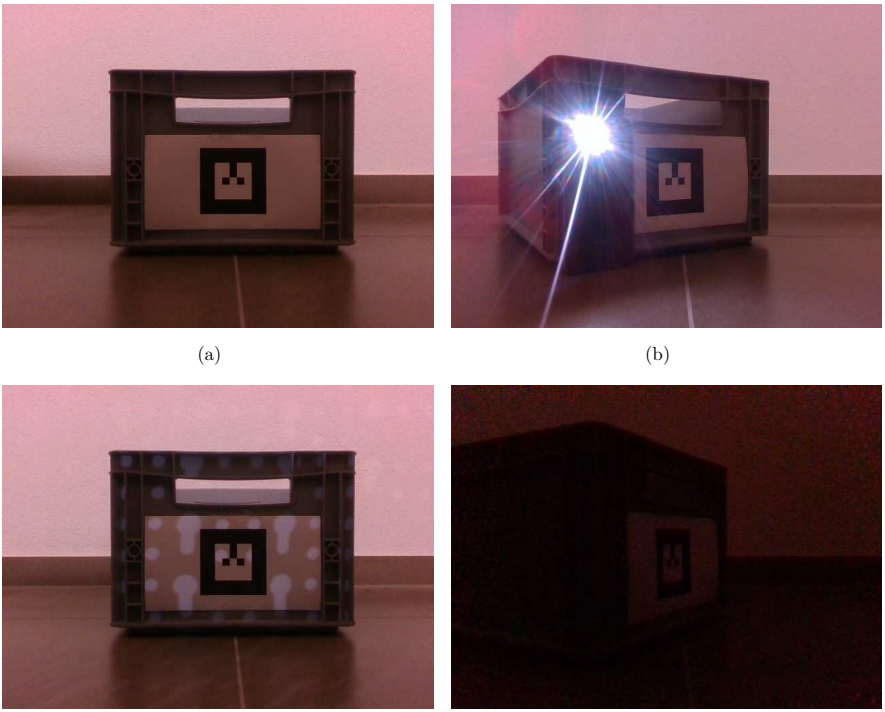
Ausgangssituation: An einer Kiste ist ein Marker angebracht. Als Marker wird ein AR-Marker (Open Robotics 2020l) genutzt. Hierbei muss die 6D Pose der Kiste mithilfe des aufgeklebten Markers unter verschiedenen Blickwinkeln und Abständen sowie unter verschiedenen Umgebungsbedingungen (z.B. Beleuchtungsverhältnisse) zuverlässig erkannt werden. Abbildung 5.1(a) zeigt den verwendeten AR-Marker und Abbildung 5.1(b) die Kiste mit angebrachtem AR-Marker. In Abbildung 5.2 ist die Szene mit verschiedenen Blickwinkeln, Abständen und Beleuchtungsbedingungen zu sehen.

Problemstellung: Die 6D Pose der Kiste muss trotz unterschiedlicher Blickwinkeln und Umgebungsbedingungen zuverlässig und genau von der eingesetzten Komponente zur Markererkennung erkannt werden. Für den Anwender gilt es herauszufinden, unter welchen Umgebungsbedingungen die Erkennung noch ausreichend gut funktioniert.



(a) (b)

Abbildung 5.1: *Markerbasierte Perzeption-Verarbeitungskette für 6D Posenbestimmung. (a) AR-Marker für 6D Posenbestimmung, (b) Aufnahme der Kiste mit angebrachtem Marker*



(a) (b)

(c) (d)

Abbildung 5.2: *Verschiedene Blickwinkel und Beleuchtungsbedingungen für Markererkennung. (a) Frontale Aufnahme mit normaler Umgebungsbeleuchtung, (b) Aufnahme mit 30° gedrehter Kiste und Blendlicht, (c) Frontale Aufnahme mit Schattenbildung, (d) Aufnahme mit 60° gedrehter Kiste und dunkler Umgebungsbeleuchtung*

ATF Verwendung

In diesem Anwendungsbeispiel unterstützt das ATF den Komponentenentwickler dabei sicherzustellen, dass die entwickelte Softwarekomponente alle definierten Einsatzfälle bezüglich Blickwinkel, Abstand und Umgebungsbedingungen abdeckt. Hierzu generiert der Komponentenentwickler eine Zusammenstellung von Aufnahmen der realen Kamera in Form von Bag-Files. ATF verwendet die Bag-File-Aufnahmen, um Daten für die Bildverarbeitungskomponente bereitzustellen. Das Ergebnis der Posenbestimmung wird mit den hinterlegten Groundtruth-daten abgeglichen. Der Komponentenentwickler konfiguriert ATF so, dass es entwicklungsbegleitend eingesetzt wird und mit jeder Änderung in der Softwarekomponente ein neuer Testdurchlauf auf dem CI-Server gestartet wird. Die für diesen Anwendungsfall verwendeten Metriken sind in Tabelle 5.1 aufgelistet. Insbesondere die Metrik M-R-4 TF-Abstand, aufgeteilt in einen rotatorischen und einen translatorischen Anteil, ermöglicht dem Komponentenentwickler das Ergebnis der Erkennung zu überprüfen.

Für die ATF Tests werden die folgenden Umgebungen definiert:

- **normal:** Normale Umgebungsbeleuchtung.
- **backlight:** Es herrscht normale Umgebungsbeleuchtung, in der jedoch eine starke punktförmige Lichtquelle aus dem Hintergrund leuchtet.
- **bright:** Erhöhte Umgebungsbeleuchtung.
- **dark:** Schwache Umgebungsbeleuchtung.
- **shady:** Es herrscht normale Umgebungsbeleuchtung, in der jedoch zusätzliche Schatten auf die Kiste geworfen werden.

Für jede Beleuchtungsbedingung wird die Posenbestimmung in mehreren Testblöcken durchgeführt, bei denen sich jeweils der Blickwinkel ändert. Der

Metrik	Bezeichnung
M-R-4	TF-Abstand

Tabelle 5.1: *Für Anwendungsbeispiel 1 verwendete Metriken.*

Testname	Konfiguration test	Konfiguration robot	Konfiguration env	Konfiguration testblockset
ts0_c0_r0_e0_s0	default	realsense	normal	testblockset_0
ts0_c0_r0_e1_s0	default	realsense	backlight	testblockset_0
ts0_c0_r0_e2_s0	default	realsense	bright	testblockset_0
ts0_c0_r0_e3_s0	default	realsense	dark	testblockset_0
ts0_c0_r0_e4_s0	default	realsense	shady	testblockset_0
ts0_c1_r0_e0_s0	calibrated	realsense	normal	testblockset_0
ts0_c1_r0_e1_s0	calibrated	realsense	backlight	testblockset_0
ts0_c1_r0_e2_s0	calibrated	realsense	bright	testblockset_0
ts0_c1_r0_e3_s0	calibrated	realsense	dark	testblockset_0
ts0_c1_r0_e4_s0	calibrated	realsense	shady	testblockset_0

Tabelle 5.2: *Testsuite für Anwendungsbeispiel 1. Es sind Tests für alle Permutationen aus test und env definiert.*

Blickwinkel wird in den Schritten 0° (`perspective_0`), 30° (`perspective_30`) und 60° (`perspective_60`) variiert. Weiterhin hat die Qualität der intrinsischen Kamerakalibrierung einen großen Einfluss auf die Erkennungsergebnisse. Alle Tests werden daher sowohl mit Standardkalibrierwerten (`default`) und verbesserten Kalibrierwerten (`calibrated`) durchgeführt. Tabelle 5.2 zeigt die für diesen Anwendungsfall definierte Testsuite.

Die ATF Tests dieses Anwendungsbeispiels werden in einen CI-Server integriert, sodass der Komponentenentwickler kontinuierlich auf die aktuellen ATF Ergebnisse zurückgreifen kann, während er an der Erkennungskomponente arbeitet.

ATF Ergebnisse

Die ATF Ergebnisse sind in den Abbildungen 5.3, 5.4 und 5.5 dargestellt und zeigen die Ergebnisse der Metrik TF-Abstand, aufgeteilt in einen rotatorischen Anteil (`tf_distance_rotation`) und einen translatorischen Anteil (`tf_distance_translation`).

Zu sehen ist, dass die Werte für die Posenbestimmung mit Standardkalibrierwerten (Abbildung 5.3) außerhalb der erwarteten Genauigkeitsgrenzen liegen, insbesondere für die um 30° gedrehte Kiste liegen die Werte außerhalb, sodass die ATF Tests als fehlgeschlagen markiert sind.

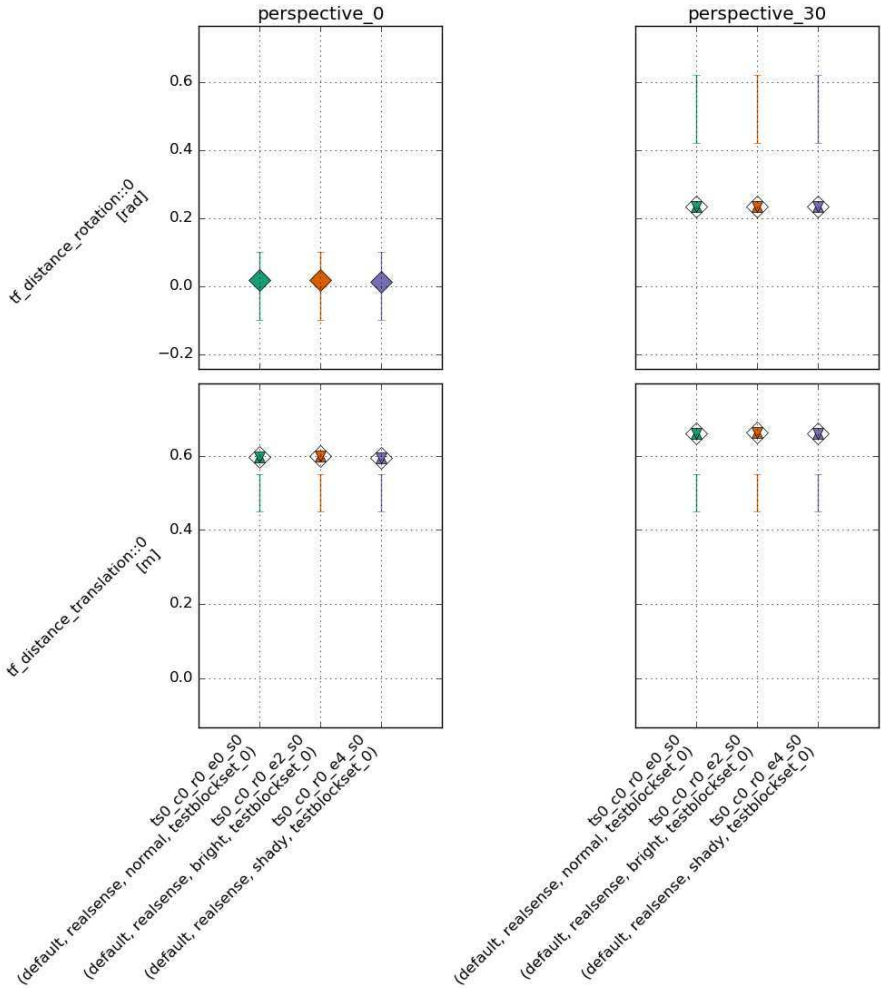


Abbildung 5.3: *ATF Ergebnisse ohne Kalibrierung. Die Werte für die Posenbestimmung liegen außerhalb der erwarteten Genauigkeitsgrenzen, insbesondere für die um 30° gedrehte Kiste liegen die Werte daneben, sodass die ATF Tests als fehlgeschlagen markiert sind.*

Werden die Tests hingegen mit den verbesserten Kalibrierwerten ausgeführt, so liegen die Werte für die Posenbestimmung innerhalb der erwarteten Genauigkeitsgrenzen (Abbildung 5.4), die ATF Tests sind demnach als erfolgreich markiert. Die Kalibrierung verbessert somit das Gesamtergebnis.

Abbildung 5.5 zeigt Testergebnisse für schlechtere Beleuchtungsbedingungen: `ts0_c1_r0_e1` für die Umgebung `backlight` und `ts0_c1_r0_e3` für die Umgebung `dark`. Für die Umgebung `backlight` ist eine Erkennung nur für die frontale Ansicht `perspective_0` möglich, nicht jedoch für die rotierten Ansichten `perspective_30`. Die Werte für die Posenbestimmung werden trotz verbesserter Kalibrierwerte mit schlechteren Beleuchtungsbedingungen schlechter bzw. eine Erkennung ist gar nicht mehr möglich. Hier besteht demnach noch Verbesserungsbedarf des Komponentenentwicklers bei der Optimierung der Erkennung.

Bewertung und Fazit

Durch die Verwendung von ATF in Verbindung mit einem CI-Server wird der Komponentenentwickler benachrichtigt, sobald die von ihm entwickelte Erkennungskomponente nicht mehr die in Form der Bag-Files und Groundtruthdaten spezifizierten Testfälle abdeckt. Zusätzlich kann der Komponentenentwickler den Verlauf der Erkennungsergebnisse für verschiedene Versionsstände oder Konfigurationen visualisieren, um zu erkennen, ob Änderungen daran das Gesamtergebnis verbessern. Diese Arbeitsweise entspricht dem in Kapitel 2.1.1 Test-Driven-Development TDD vorgestellten Ansatzes. D.h. der Komponentenentwickler würde solange weiter optimieren, bis alle definierten Testfälle, z.B. Beleuchtungsbedingungen, erfüllt werden.

Tabelle 5.3 zeigt die in Anwendungsbeispiel 1 validierten Anforderungen. Insbesondere wurden die Möglichkeiten von ATF zur Testautomatisierung und Benchmarking validiert (Anforderungen AF-8 Testautomatisierung, AF-9 TDD-Integration und AF-10 Benchmarking). Somit fördert ATF die Steigerung der Qualität der Erkennungskomponente, indem z.B. die Fehlerrate sinkt, die Genau-

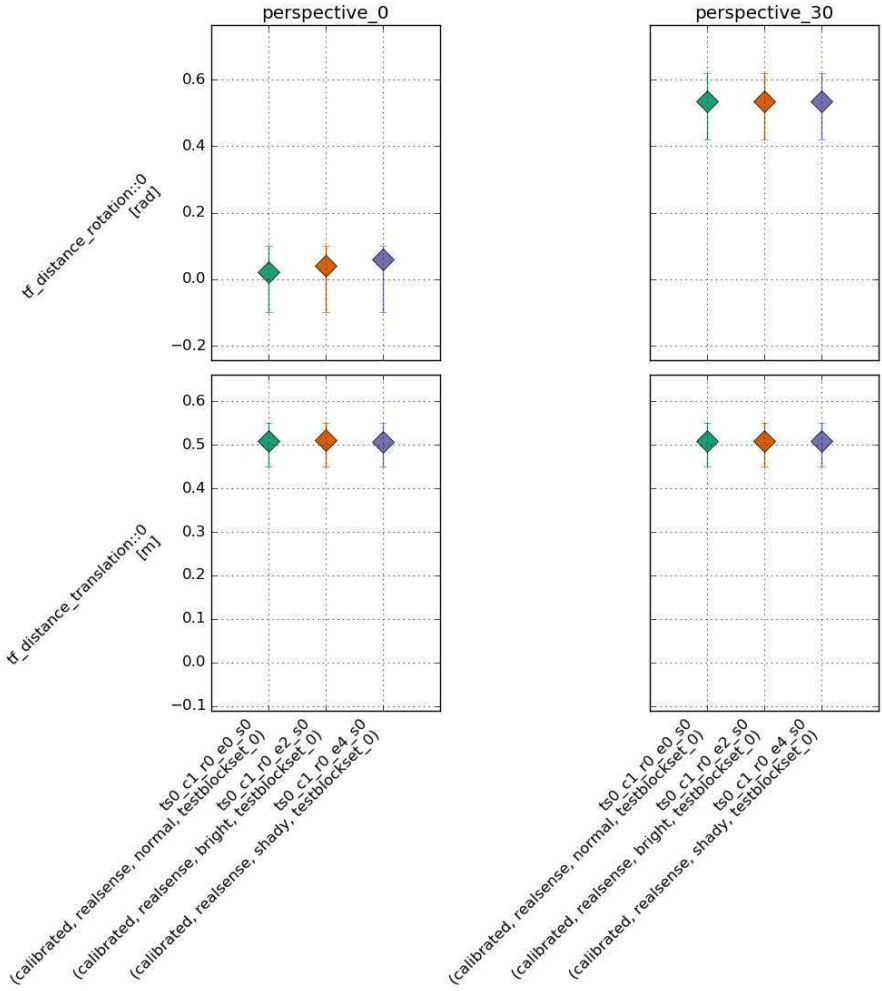


Abbildung 5.4: *ATF Ergebnisse mit Kalibrierung. Die Werte für die Posenbestimmung liegen innerhalb der erwarteten Genauigkeitsgrenzen, die ATF Tests sind demnach als erfolgreich markiert.*

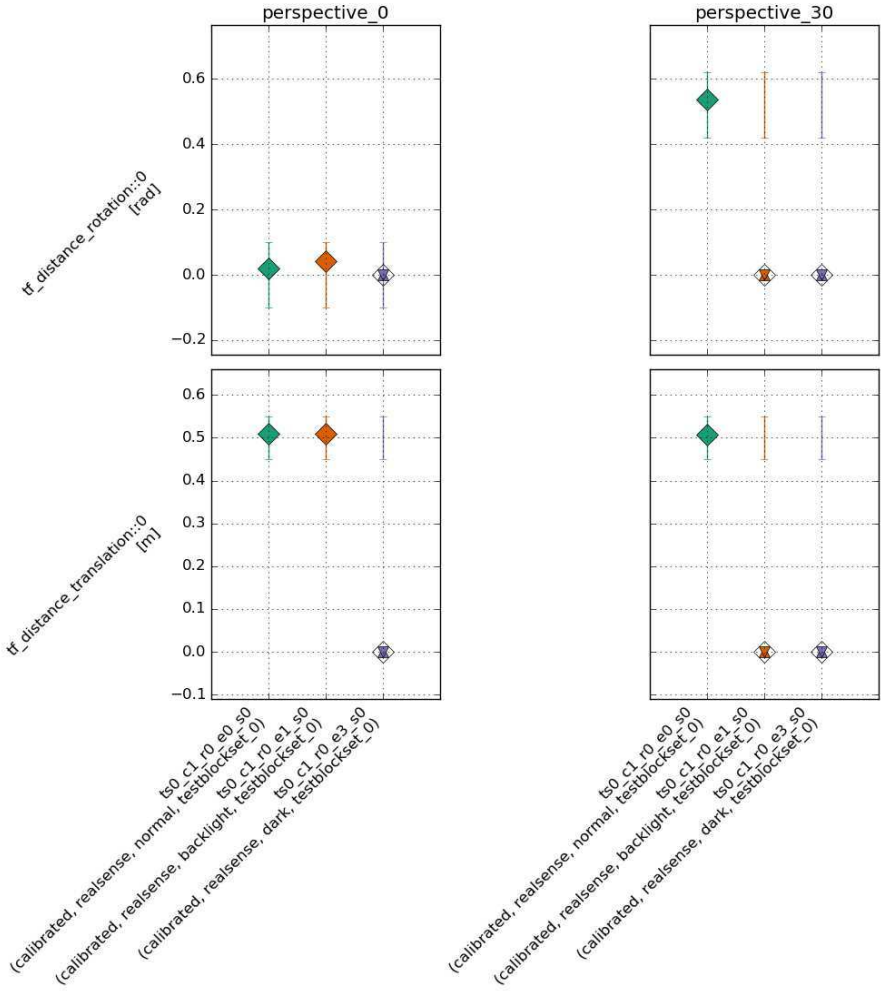


Abbildung 5.5: *ATF Ergebnisse mit schwierigen Beleuchtungsbedingungen. Die Werte für die Posenbestimmung werden trotz verbesserter Kalibrierwerte mit schlechteren Beleuchtungsbedingungen schlechter bzw. eine Erkennung ist gar nicht mehr möglich.*

Anforderungen	AF-1 Testbausteine	AF-2 *Hardware-in-the-loop*	AF-3 *Simulation-in-the-loop*	AF-4 Teststufen	AF-5 Testgenerierung	AF-6 Testauswertung	AF-7 Ergebnisvisualisierung	AF-8 Testautomatisierung	AF-9 TDD-Integration	AF-10 Benchmarking	AF-11 Minimale Beeinflussung	AF-12 ROS Integration
Anwendungsbeispiel 1	✓			✓	✓	✓	✓	✓	✓	✓		✓

Tabelle 5.3: *In Anwendungsbeispiel 1 umgesetzte Anforderungen.*

igkeit steigt und die Robustheit gegenüber veränderten Umgebungsbedingungen verbessert wird.

5.1.2 Anwendungsbeispiel 2: Integrationstest einer Manipulations-Verarbeitungskette

Steckbrief Anwendungsbeispiel 2

Anwender: Systemintegrator für Roboterarme

Aufgabe: Aufgabe ist es, mit einem stationären Roboterarm Kisten auf einem Tisch zu Greifen und Abzulegen (Pick-Place Aufgabe), wobei der direkte Zugriff von oben aufgrund eines Hindernisses nicht möglich ist. Die Kisten müssen frontal erreicht und dann nach vorne entnommen werden.

Dabei gilt es einen Roboterarm und eine geeignete Greifstrategie auszuwählen, mit der möglichst viele Kisten erreicht werden können. Neben der Reichweite des Roboterarms spielt dabei die umgesetzte Greifstrategie eine wesentliche Rolle, bei der das Zusammenspiel zwischen Armansteuerung, Bahnplaner, der Berechnung der Inversen Kinematik für die Bahnpunkte und der Aneinanderreihung von Trajektorien als Linearfahrt oder Punkt-zu-Punkt-Fahrt einen entscheidenden Einfluss hat. Da der Preis für die Roboterarme mit höherer Reichweite steigt, soll dabei der preisgünstigste Arm ausgewählt werden, der alle relevanten Kisten erreichen kann.

Ausgangssituation: Für dieses Anwendungsbeispiel werden mehrere Roboterarme des Herstellers Universal Robots untersucht. Abbildung 5.6 zeigt drei Varianten: UR3, UR5 und UR10, die sich in Reichweite, Traglast und Preis voneinander unterscheiden. Die relevanten Kisten befinden sich in einer Entfernung von $0.25m$ bis $0.75m$ und müssen jeweils frontal gegriffen und durch eine Linearbewegung entnommen werden.

Problemstellung: Je nach Auswahl der Greifstrategie, deren Parametrisierung und der Trajektorien (Linearfahrt oder Punkt-zu-Punkt-Fahrt) sind Kisten an der gleichen Zielposition nicht immer erreichbar. Daher wird für die

Arbeitsraumanalyse nicht nur die Reichweite des Datenblatts herangezogen, sondern eine Anfahr- und Greifbewegung in Simulation durchgeführt (siehe Abbildung 5.7). Es sind Zielposen mit $0.0m, 0.25m, 0.5m, 0.75m, 1.0m$ und $1.25m$ modelliert. Die Kisten müssen, wie in Abbildung 5.7(b) dargestellt, zuerst nach vorne entnommen werden und können dann erst angehoben werden, sodass eine Kollision mit dem über den Kisten angebrachten Hindernis vermieden wird. Die Problemstellung kann daher nicht allein durch Betrachtung der der Reichweite laut Datenblatt oder durch analytische Methoden der Arbeitsraumberechnung gelöst werden, sondern erfordert eine Analyse der Greifstrategie. Auf Basis dieser Greifanalyse kann anschließend das entsprechende Robotermodell ausgewählt werden.

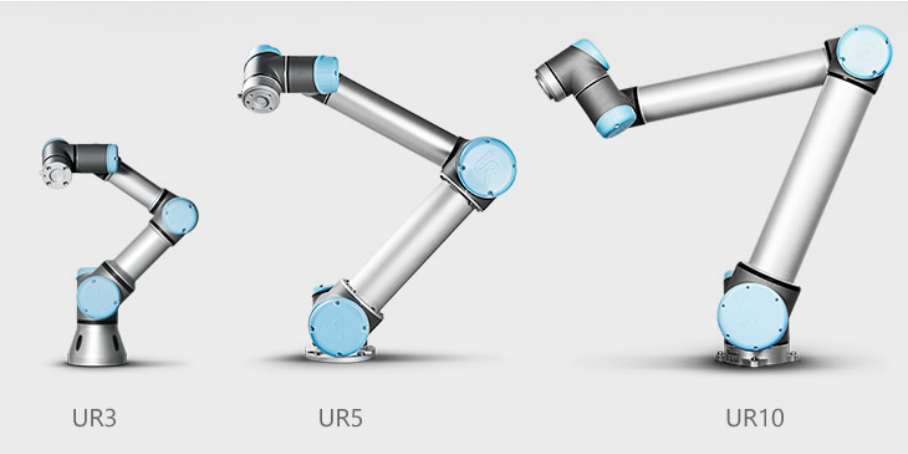
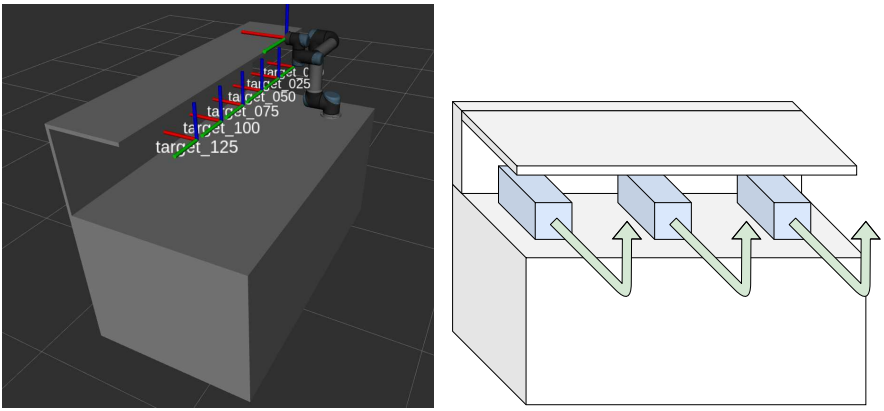


Abbildung 5.6: *Untersuchte Roboterarme mit unterschiedlichen Reichweiten. Es werden drei Varianten des Herstellers Universal Robots untersucht: UR3, UR5 und UR10. (Quelle: Bachmann Engineering AG)*

ATF Verwendung

In diesem Anwendungsbeispiel wird ATF für die automatisierte Auswertung der Greifstrategie genutzt. Hierzu wird für jede Kiste bzw. jede Zielpose ein



(a) (b)

Abbildung 5.7: *Modellierte Szene für die Greifuntersuchung. (a) Der Endeffektor des Roboterarms soll an unterschiedlichen Zielpositionen Greifen. Dazu sind Zielposen mit* $0.0m, 0.25m, 0.5m, 0.75m, 1.0m$ *und* $1.25m$ *modelliert. (b) Die Kisten müssen zuerst nach vorne entnommen werden und können dann erst angehoben werden, sodass eine Kollision mit dem über den Kisten angebrachten Hindernis vermieden wird.*

Metrik	Bezeichnung
M-R-4	TF-Abstand
M-R-11	Benutzerergebnis

Tabelle 5.4: *Für Anwendungsbeispiel 2 verwendete Metriken.*

Testblock angelegt, der mit den Metriken aus Tabelle 5.4 ausgewertet wird. Die Metrik M-R-4 TF-Abstand wird genutzt, um den Abstand zur Zielpose zu bewerten (`reach`). In der Metrik M-R-11 Benutzerergebnis wird der Grad der Zielerreichung als Wert zwischen 0 (= 0% der Trajektorie zum Ziel) und 1 (= 100% der Trajektorie zum Ziel) erfasst. Tabelle 5.5 zeigt die für diesen Anwendungsfall definierte Testsuite. Es ist jeweils ein Tests für jedes Robotermodell definiert. Die Tests werden als *Simulation-in-the-loop* Tests durchgeführt.

Testname	Konfiguration test	Konfiguration robot	Konfiguration env	Konfiguration testblockset
ts0_c0_r0_e0_s0	default	UR3	default	default
ts0_c0_r1_e0_s0	default	UR5	default	default
ts0_c0_r2_e0_s0	default	UR10	default	default

Tabelle 5.5: *Testsuite für Anwendungsbeispiel 2. Es ist jeweils ein Tests für jedes Robotermodell definiert: UR3, UR5 und UR10.*

ATF Ergebnisse

Die ATF Ergebnisse sind in der Abbildung 5.8 dargestellt und zeigen die Ergebnisse der Metriken TF-Abstand als Reichweite `reach` und den Grad der Zielerreichung als `fraction`. Die Tests zeigen je nach verwendetem Roboterarm Reichweiten zwischen $0.25m$ (`ts0_c0_r0_e0_s0` mit Roboterarm UR3) und $1.0m$ (`ts0_c0_r2_e0_s0` mit Roboterarm UR10). Das Ziel mit einer Entfernung von $1.25m$ konnte keiner der Roboterarme erreichen. Das Ziel direkt vor dem Roboter (Entfernung $0.0m$) können zwar alle Armvarianten aufgrund ihres Arbeitsraums erreichen, jedoch würde die Linearfahrt beim Entnehmen der Kiste durch den Roboterfuß führen, sodass dieses Ziel aufgrund einer Eigenkollision ebenfalls von keinem der Arme erreicht werden kann.

Das Modell UR3 scheidet aus, da dieses die relevanten Kisten in einer Entfernung von $0.25m$ bis $0.75m$ nicht erreicht. Die Modelle UR5 und UR10 erfüllen die Anforderungen und erreichen alle relevanten Kisten. Aus wirtschaftlichen Gründen kann der UR5 gewählt werden.

Bewertung und Fazit

Mithilfe von ATF konnte die Arbeitsraumanalyse automatisiert werden und so, ohne zusätzlichen manuellen Aufwand, unterschiedliche Varianten von Roboterarmen und Greifstrategien untersucht werden. Der Vorteil daran ist, dass die Arbeitsraumanalyse nicht allein auf Basis der Reichweiten des Datenblatts untersucht werden konnte, sondern direkt mit der umgesetzten Greifstrategie als *Simulation-in-the-loop* Tests. Die analysierten Greifstrategien lassen sich so direkt auf den echten Anwendungsfall übertragen.

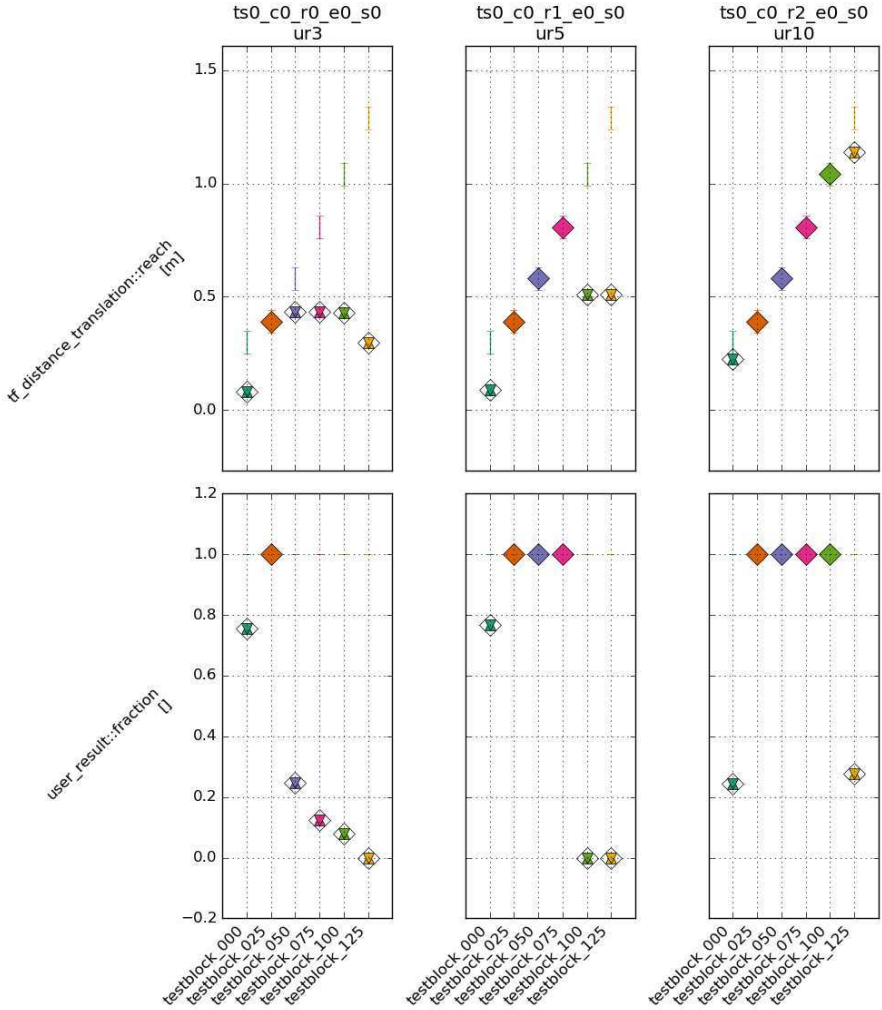


Abbildung 5.8: *ATF Ergebnisse für die Manipulationstests. Dargestellt ist die Reichweite* `reach` *und der Grad der Zielerreichung* `fraction` *als Wert zwischen* 0 *(=* 0% *der Trajektorie zum Ziel) und* 1 *(=* 100% *der Trajektorie zum Ziel). Die Tests zeigen je nach verwendetem Roboterarm Reichweiten zwischen* 0.25*m (*`ts0_c0_r0_e0_s0` *mit Roboterarm UR3) und* 1.0*m (*`ts0_c0_r2_e0_s0` *mit Roboterarm UR10). Die Ziele mit einer Entfernung von* 0.0*m und* 1.25*m kann keiner der Roboterarme erreichen.*

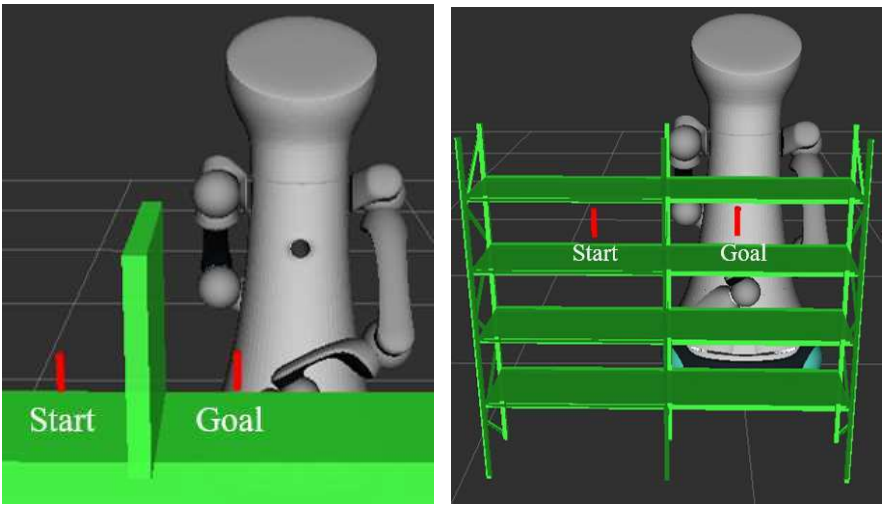
Anforderungen	AF-1 Testbausteine	AF-2 *Hardware-in-the-loop*	AF-3 *Simulation-in-the-loop*	AF-4 Teststufen	AF-5 Testgenerierung	AF-6 Testauswertung	AF-7 Ergebnisvisualisierung	AF-8 Testautomatisierung	AF-9 TDD-Integration	AF-10 Benchmarking	AF-11 Minimale Beeinflussung	AF-12 ROS Integration
Anwendungsbeispiel 2	✓		✓		✓	✓	✓	✓		✓		✓

Tabelle 5.6: *In Anwendungsbeispiel 2 umgesetzte Anforderungen.*

Tabelle 5.6 zeigt die in Anwendungsbeispiel 2 validierten Anforderungen. Insbesondere wurden die Möglichkeiten von ATF zur automatisierten Auswertung anhand von Testbausteinen und *Simulation-in-the-loop* Tests genutzt (Anforderungen AF-1 Testbausteine und AF-3 *Simulation-in-the-loop*).

Weitere Einsätze von ATF für Manipulationstests

Ein ähnlicher Anwendungsfall zu Anwendungsbeispiel 2 ist in Weißhardt & Koehler (2016) beschrieben und diskutiert. Hierbei wurden komplexere Hindernissituationen modelliert (siehe Abbildung 5.9) und diese für den zweiarmigen Care-O-bot 4 ausgewertet. Dabei wurden unterschiedliche Greifstrategien für die in Summe über 20 Freiheitsgrade (verteilt auf Torso, Arme und Greifer) untersucht.



(a) (b)

Abbildung 5.9: *Greifanalyse mit zweiarmigem Care-O-bot 4 in komplexer Hindernissituation aus Weißhardt & Koehler (2016). (a) Tischszene, (b) Regalszene.*

5.1.3 Anwendungsbeispiel 3: Systemtest Navigation

Steckbrief Anwendungsbeispiel 3

Anwender: Systemintegrator für mobile Serviceroboter

Aufgabe: Ziel ist es, ein autonomes Navigationssystem für einen mobilen Serviceroboter zu entwickeln, das es ermöglicht, in einer zuvor definierten Umgebung autonom Navigationsziele anzufahren. In der mobilen Robotik ist es ein gebräuchlicher Ansatz ein Navigationssystem in Teilkomponenten wie z.B. Lokalisierung sowie lokale und globale Pfadplanung aufzuteilen (Marder-Eppstein et al. 2010).

In diesem Anwendungsfall soll insbesondere auf die Pfadplanung und deren Aufteilung in die beiden folgenden Komponenten eingegangen werden:

- **Globaler Pfadplaner**, der offline einen kollisionsfreien Pfad von der aktuellen Roboterposition zur Zielposition in einem reduzierten Konfigurationsraum plant.
- **Lokaler Pfadplaner**, der online in einem vollständigen, aber zeitlich und räumlich limitierten Konfigurationsraum die Bewegungsbefehle so optimiert, dass sie dem globalen Pfad folgen, dabei aber weitere Randbedingungen wie z.B. kinematische oder kinodynamische Beschränkungen berücksichtigt und vorher unbekannten Hindernissen ausweicht sowie Störungen durch die Lokalisierung oder Fahrwerksregelung ausgleicht.

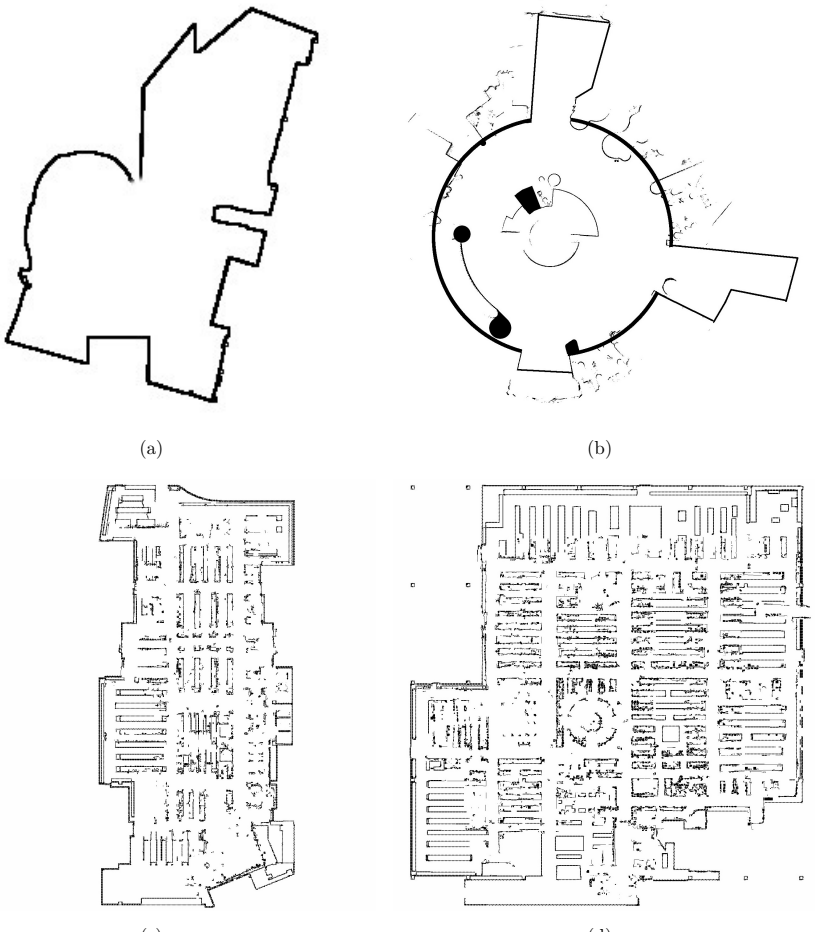
Hierbei gilt es eine Kombination von globalem und lokalem Pfadplaner sowie einen Konfigurationssatz zu finden der die Anforderungen an die Navigation mit dem jeweiligen Roboter in der jeweiligen Umgebungen abdeckt.

Ausgangssituation: Für dieses Anwendungsbeispiel wird Care-O-bot 4 mit einer Auswahl an Pfadplanern aus dem ROS Umfeld eingesetzt. Care-O-bot 4 kann durch sein quasi-omnidirektionales Fahrwerk mit drei Fahr-Dreh-Modulen sowohl eine omnidirektionale, als auch eine differentielle Kinematik abbilden.

Das Navigationssystem ist dabei eine Zusammenstellung von verschiedenen Softwarekomponenten zur Lokalisierung, Pfadplanung und -regelung, sowie Fahrwerks- und Odometrieregelung (z.B. omnidirektional oder differentiell). Hierbei gibt es die Möglichkeit verschiedene Softwarekomponenten, sowie verschiedene Konfigurationssätze für diese Softwarekomponenten einzusetzen. Tabelle 5.7 zeigt die zu testenden Variationen.

Abbildung 5.10 zeigt Beispiele für Umgebungen, in denen die zu entwickelnde Navigationslösung eingesetzt werden soll. Die hier gezeigten Umgebungen unterscheiden sich wesentlich in Größe, Art und Form der Hindernisse, Kartenqualität und Anzahl an dynamischen Hindernissen. In diesem Anwendungsfall sind Tests in der Zielumgebung u.a. wegen Zugangsbeschränkungen, erhöhtem Aufwand und Kosten für Robotertransport jedoch nur eingeschränkt möglich. Als Testumgebung steht lediglich eine Laborumgebung sowie Simulatoren zur Verfügung.

Problemstellung: Je nach Kombination der Softwarekomponenten und Auswahl von Konfigurationssätzen kann das Fahrverhalten des Roboters sowie die Robustheit und Qualität der Zielerreichung variieren. Wird beispielsweise eine Lokalisierungskomponente eingesetzt, die die dynamischen Veränderungen in der Umgebung nicht abdecken kann, so kann es zu Fehllokalisierungen kommen, die das Weiterfahren des Roboters und das Erreichen des Ziels unmöglich machen. Weiter könnte z.B. die Auswahl der Pfadplanungskomponente bzw. deren Konfigurationssatz das Passieren von Engstellen unmöglich machen.



(a) (b)

(c) (d)

Abbildung 5.10: *Beispiele für Zielumgebungen für Anwendungsbeispiel 3. (a) Museum Haus der Geschichte in Bonn (ca. 50qm), (b) Hotel 25hours Köln (ca. 200qm), (c) Saturnmarkt Hamburg (ca. 4.000qm), (d) Saturnmarkt Ingolstadt (ca. 5.000qm)*

Zu variierende Parameter (Kategorie)	Zu testende Variationen (Testsuite Bezeichnung)
Fahrwerkskinematik (robot)	Differentiell (`diff` = r0) Omnidirektional (`omni` = r1)
Lokale Pfadplaner (test)	Elastic Band (`ipa_eband_planner` = c0) Dynamic Window (`dwa_local_planner` = c1) Trajectory Planner ROS (`base_local_planner` = c2)
Umgebungen (env)	Museum Haus-der-Geschichte Bonn Hotel 25hours Köln Saturnmarkt Hamburg Saturnmarkt Ingolstadt

Tabelle 5.7: *Für Anwendungsbeispiel 3 zu untersuchende Variationen.*

ATF Verwendung

Der Systemintegrator nutzt in diesem Anwendungsfall das ATF, um die Komposition aller Navigationskomponenten, sowie ihre Konfiguration für eine Einsatzumgebung zu optimieren. Die hierfür verwendeten Metriken sind in Tabelle 5.8 dargestellt. Insbesondere die Metriken Zeitdauer und TF-Pfadlänge sind geeignet, um zu beurteilen, ob das Ziel schnell und ohne Umwege erreicht werden konnte.

Für Anwendungsbeispiel 3 wurde eine Auswahl an Navigationsbereichen aus den Umgebungen in Abbildung 5.10 extrahiert und in einen Testparcour überführt. Der Testparcour ist angelehnt an die Tests, wie sie in Garcia Lopez

Metrik	Bezeichnung
M-R-1	Zeitdauer
M-R-3	Hindernisabstand
M-R-4	TF-Abstand
M-R-5	TF-Pfadlänge
M-R-6	TF-Pfadgeschwindigkeit
M-R-11	Benutzerergebnis

Tabelle 5.8: *Für Anwendungsbeispiel 3 verwendete Metriken.*

(2019) vorgestellt und evaluiert werden. Der Testparcour ist in Abbildung 5.11 und die dazugehörigen Navigationsziele in Abbildung 5.12 dargestellt. Der Testparcour gliedert sich in die folgenden Abschnitte:

- **basic:** In diesem Abschnitt werden die grundsätzlichen Fahrmanöver wie z.B. Anfahren eines vorwärts gerichteten Ziels, Anfahren eines rückwärts gerichteten Ziels und Anfahren eines seitlichen Ziels geprüft.
- **narrow_passage:** Dieser Abschnitt bildet die Durchfahrt einer Engstelle wie z.B. einer Türdurchfahrt ab.
- **round_trip:** In diesem Abschnitt wird das Anfahren einer Sequenz von Navigationszielen abgebildet, sodass sich insgesamt eine geschlossene Rundfahrt ergibt. Alle Navigationsziele liegen dabei in Fahrtrichtung.
- **rooms:** Dieser Abschnitt kombiniert das Anfahren von mehreren Navigationszielen unterschiedlicher Orientierung mit dem Passieren von Engstellen.

Für jeden Abschnitt wird in ATF eine Umgebung mit entsprechenden Navigationszielen definiert. Damit werden die in Tabelle 5.9 aufgelisteten Tests als Testsuite für Anwendungsbeispiel 3 definiert.

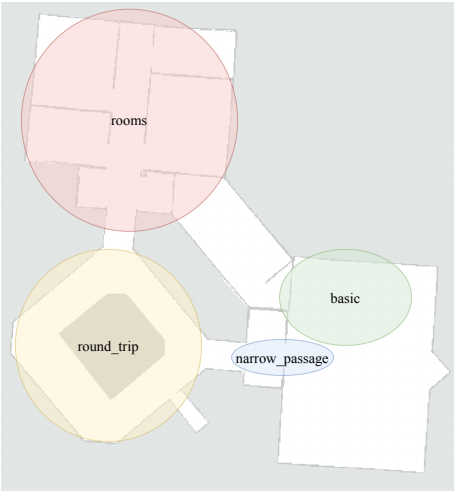
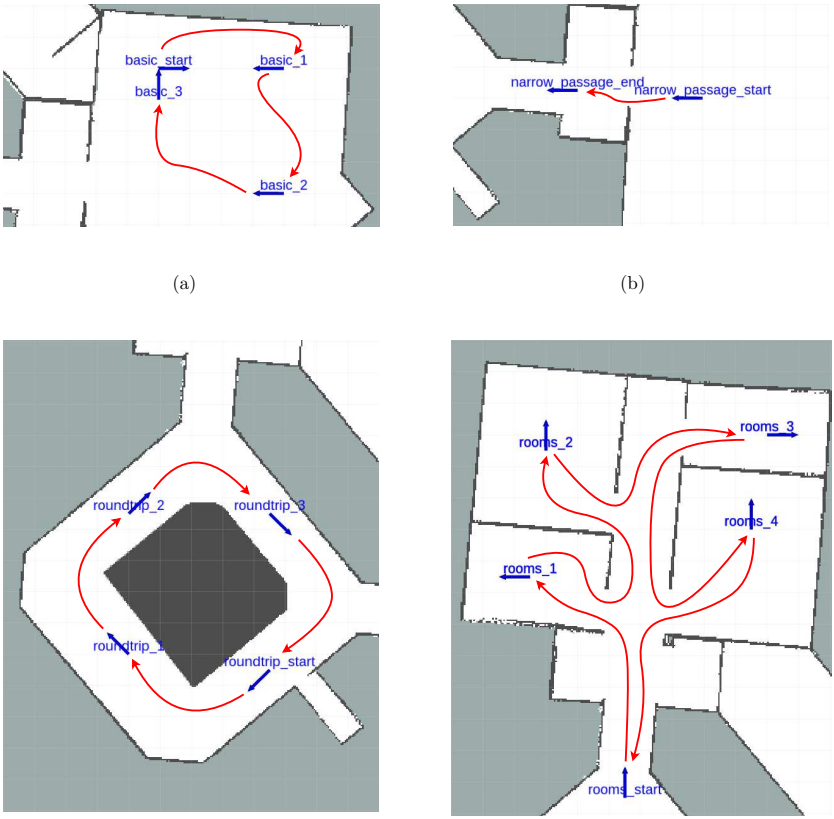


Abbildung 5.11: *Testparcour in Büroumgebung. Die relevanten Navigationsbereiche aus Abbildung 5.10 wurden in einen Testparcour überführt und in mehrere Abschnitte aufgeteilt: basic (grün), narrow_passage (blau), round_trip (gelb), rooms (rot).*



(a) (b)

(c) (d)

Abbildung 5.12: *Umgebungen für Anwendungsbeispiel 3. Dargestellt sind Navigationsziele mit Orientierung als blaue Pfeile und deren Verbindung als Route in rot. (a) Testen von unterschiedlichen Zielorientierungen in Umgebung basic, (b) Durchfahrt durch Engstelle in Umgebung narrow_passage, (c) Geschlossene Rundfahrt mit in Fahrtrichtung weisenden Zielorientierungen in Umgebung round_trip, (d) Kombination aus verschiedenen Engstellen und Zielorientierungen in Umgebung rooms.*

Testname	Konfiguration test	Konfiguration robot	Konfiguration env	Konfiguration testblockset
ts0_c0_r0_e0_s0	ipa_eband_planner	diff	basic	sim
ts0_c0_r0_e1_s0	ipa_eband_planner	diff	narrow_passage	sim
ts0_c0_r0_e2_s0	ipa_eband_planner	diff	round_trip	sim
ts0_c0_r0_e3_s0	ipa_eband_planner	diff	rooms	sim
ts0_c0_r1_e0_s0	ipa_eband_planner	omni	basic	sim
ts0_c0_r1_e1_s0	ipa_eband_planner	omni	narrow_passage	sim
ts0_c0_r1_e2_s0	ipa_eband_planner	omni	round_trip	sim
ts0_c0_r1_e3_s0	ipa_eband_planner	omni	rooms	sim
ts0_c1_r0_e0_s0	dwa_local_planner	diff	basic	sim
ts0_c1_r0_e1_s0	dwa_local_planner	diff	narrow_passage	sim
ts0_c1_r0_e2_s0	dwa_local_planner	diff	round_trip	sim
ts0_c1_r0_e3_s0	dwa_local_planner	diff	rooms	sim
ts0_c1_r1_e0_s0	dwa_local_planner	omni	basic	sim
ts0_c1_r1_e1_s0	dwa_local_planner	omni	narrow_passage	sim
ts0_c1_r1_e2_s0	dwa_local_planner	omni	round_trip	sim
ts0_c1_r1_e3_s0	dwa_local_planner	omni	rooms	sim
ts0_c2_r0_e0_s0	base_local_planner	diff	basic	sim
ts0_c2_r0_e1_s0	base_local_planner	diff	narrow_passage	sim
ts0_c2_r0_e2_s0	base_local_planner	diff	round_trip	sim
ts0_c2_r0_e3_s0	base_local_planner	diff	rooms	sim
ts0_c2_r1_e0_s0	base_local_planner	omni	basic	sim
ts0_c2_r1_e1_s0	base_local_planner	omni	narrow_passage	sim
ts0_c2_r1_e2_s0	base_local_planner	omni	round_trip	sim
ts0_c2_r1_e3_s0	base_local_planner	omni	rooms	sim
ts1_c0_r0_e0_s0	ipa_eband_planner	diff	basic	sim
ts1_c0_r0_e0_s1	ipa_eband_planner	diff	basic	hw
ts1_c0_r1_e0_s0	ipa_eband_planner	omni	basic	sim
ts1_c0_r1_e0_s1	ipa_eband_planner	omni	basic	hw
ts1_c1_r0_e0_s0	dwa_local_planner	diff	basic	sim
ts1_c1_r0_e0_s1	dwa_local_planner	diff	basic	hw
ts1_c1_r1_e0_s0	dwa_local_planner	omni	basic	sim
ts1_c1_r1_e0_s1	dwa_local_planner	omni	basic	hw
ts1_c2_r0_e0_s0	base_local_planner	diff	basic	sim
ts1_c2_r0_e0_s1	base_local_planner	diff	basic	hw
ts1_c2_r1_e0_s0	base_local_planner	omni	basic	sim
ts1_c2_r1_e0_s1	base_local_planner	omni	basic	hw

Tabelle 5.9: *Testsuites für Anwendungsbeispiel 3. Die Tests sind aufgeteilt in eine Testsuite mit reinen simulation-in-the-loop Tests für alle Permutationen aus test, robot und env (`ts0`) und eine Testsuite für den Abgleich zwischen Simulation und realem Roboter als hardware-in-the-loop Tests (`ts1`).*

Das ATF wird für diesen Anwendungsfall als *simulation-in-the-loop* System verwendet. Dies hat den Vorteil, dass die ansonsten ressourcen- und zeitaufwendigen Tests mit einem echten Roboter in den Einsatzumgebungen durch eine Simulationsumgebung ersetzt werden können. Weiterhin kann mithilfe der Simulationsumgebung die Ausführung von mehreren Tests parallelisiert werden, sodass eine Vielzahl an Tests in kurzer Zeit realisiert werden kann. Hierbei gilt es sicherzustellen, dass die in der Simulationsumgebung gewonnenen Ergebnisse auf die Realität übertragbar sind. Dies kann durch Stichproben überprüft werden, in denen die gleichen Tests in Simulation und unter Realbedingungen als *hardware-in-the-loop* Tests durchgeführt und anschließend vergleichen werden.

Um statistische Schwankungen erfassen zu können werden für diesen Anwendungsfall alle Tests aus Tabelle 5.9 10 mal wiederholt. Aus den Einzelergebnissen wird anschließen ein Mittelwert sowie eine Spannweite als Minimal- und Maximalwerte gebildet. Für ein insgesamt erfolgreiches Gesamtergebnis, müssen alle Wiederholungen jeweils einzeln erfolgreich abgeschlossen werden. Ist eine der Wiederholungen nicht erfolgreich, wird der gesamte Test als fehlgeschlagen markiert.

ATF Ergebnisse

Die ATF Ergebnisse sind in den Abbildungen 5.13, 5.14, 5.15 und 5.16 dargestellt und zeigen die Ergebnisse der Metriken Zeitdauer (`time`), als Fahrzeit zum Navigationsziel sowie TF-Pfadlänge (`tf_length`) und Benutzerergebnis (`user_result`). `tf_length` gibt dabei die Fahrstrecke an und ist in einen translatorischen (`tf_length_translation`) und einen rotatorischen (`tf_length_translation`) Anteil aufgeteilt. Mit der Metrik `user_result` wird erfasst, ob das Navigationsziel erfolgreich erreicht werden konnte, oder ob die Fahrt abgebrochen werden musste. Insgesamt ist zu sehen, dass die Anwendung von verschiedenen Konfigurationssätzen auf die verschiedenen Umgebungen zu unterschiedlichem Systemverhalten in Bezug auf die Zielerreichung führt. Mithilfe der Metriken kann so in einem Benchmarkverfahren festgestellt werden,

welche Kombination und welcher Konfigurationssatz die besten Ergebnisse liefert.

Im folgenden werden beispielhaft einige Erkenntnisse bezüglich der eingesetzten Navigationskomponenten aus den ATF Ergebnissen für die verschiedenen Umgebungen abgeleitet.

Ergebnisse aus Umgebung `basic`: In Abbildung 5.13 sind die Werte der Metriken für Umgebung `basic` dargestellt. Hier lässt sich erkennen, dass sich die translatorische Pfadlänge nicht wesentlich zwischen den unterschiedlichen Konfigurationen unterscheidet. Wohingegen sich bei der Fahrzeit und der rotatorischen Pfadlänge deutliche Unterschiede zeigen. Hier ist beispielsweise die Kombination aus Test `ts0_c0_r1_e0_s0`, d.h. `ipa_eband_planner` mit omnidirektionalem Fahrwerk, am schnellsten am Ziel und weist zudem die geringste Rotationsrate auf.

Ergebnisse aus Umgebung `narrow_passage`: Abbildung 5.14 zeigt die Ergebnisse für die Umgebung `narrow_passage`. Es zeigt sich, dass alle Kombinationen im Mittel eine ähnliche translatorische Fahrstrecke und auch ähnliche Fahrzeiten aufweisen. Die Rotationsrate ist allerdings bei der Kombination von omnidirektionalem Fahrwerk und `ipa_eband_planner` (Test `ts0_c0_r1_e1_s0`) fast Null, d.h. der Roboter macht hier kaum rotatorische Rangierbewegungen. Wohingegen beim gleichen Pfadplaner mit differentiellem Fahrwerk (Test `ts0_c0_r0_e1_s0`) deutlich mehr Rangierbewegungen nötig sind, um die Engstelle zu meistern. Bei Test `ts0_c0_r0_e1_s0` ist zudem zu erkennen, dass die Rangierbewegungen für mehrere Durchläufe nicht konstant sind, sondern sich unterschiedliche Werte für die Rotationsrate und die Fahrzeit ergeben. Die Spannweite ist durch kleine Dreiecke dargestellt und geht beispielsweise bei der Fahrzeit von ca. 10 Sekunden bis auf fast 30 Sekunden. Der Anwender kann hierdurch abschätzen, wie verlässlich eine Kombination gleiche Ergebnisse liefert oder ob diese großen Schwankungen trotz gleicher Konfiguration und gleichen Umgebungsbedingungen unterliegen.

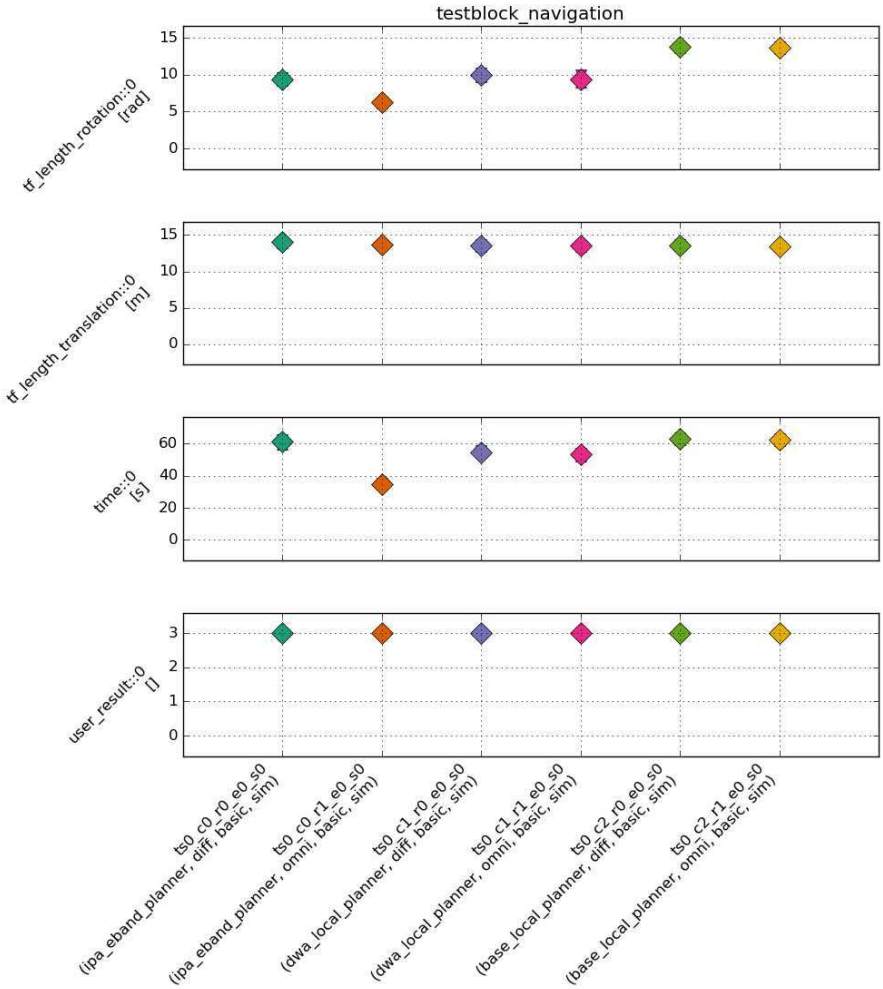


Abbildung 5.13: *ATF Ergebnisse für Umgebung* `basic`*. Die translatorische Pfadlänge unterscheidet sich nicht wesentlich zwischen den unterschiedlichen Konfigurationen. Wohingegen sich bei der Fahrzeit und der rotatorischen Pfadlänge deutliche Unterschiede zeigen.*

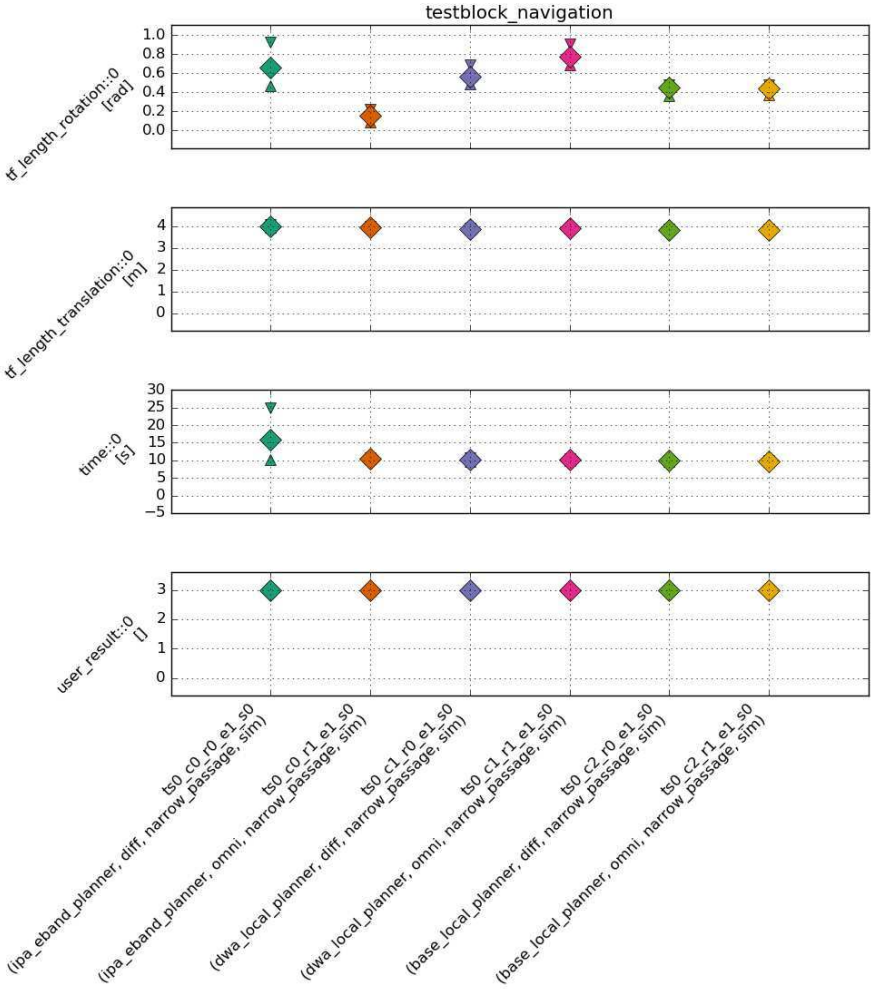


Abbildung 5.14: *ATF Ergebnisse für Umgebung* ***narrow_pssage****. Alle Kombinationen weisen im Mittel eine ähnliche translatorische Fahrstrecke und auch ähnliche Fahrzeiten auf. Die Rotationsrate ist allerdings bei der Kombination von omnidirektionalem Fahrwerk und* ***ipa_eband_planner*** *(Test* `ts0_c0_r1_e1_s0`*) fast Null, d.h. der Roboter macht hier kaum rotatorische Rangierbewegungen, wohingegen alle anderen Kombinationen deutliche Rangierbewegungen zeigen.*

Ergebnisse aus Umgebung `round_trip`: Wie in Abbildung 5.15 zu sehen ist, sind die Unterschiede zwischen den Tests in der Umgebung `round_trip` gering, da die Umgebung keine besonderen Hindernisse oder Engstellen enthält und alle Navigationsziele jeweils in Fahrtrichtung angefahren werden können.

Ergebnisse aus Umgebung `rooms`: Abbildung 5.16 zeigt die Ergebnisse für die Umgebung `rooms`. Diese stellt sich für die Navigation am herausfordernsten dar, da hier Hindernisse, Engstellen und unterschiedlich orientierte Navigationsziele auf engem Raum auftreten. Mit der Metrik `user_result` wird erfasst, ob die Navigationskomponente das Navigationsziel erfolgreich erreichen konnte oder die Fahrt abgebrochen werden musste. Für die Tests `ts0_c0_r0_e3_s0`, `ts0_c1_r1_e3_s0`, `ts0_c2_r0_e3_s0` und `ts0_c2_r1_e3_s0` ist die erfolgreiche Zielerreichung nicht immer möglich, sodass sich diese Kombinationen nicht für einen robusten Einsatz in dieser Umgebung eignen.

Übertragbarkeit von *simulation-in-the-loop* Tests auf reale Einsatzbedingungen

Die in den Abbildungen 5.13, 5.14, 5.15 und 5.16 dargestellten Testergebnisse wurden in einer Simulationsumgebung aufgezeichnet, um die Vorteile bezüglich Parallelisierbarkeit ausnutzen zu können und die Tests unüberwacht durchführen zu können. Um sicherzustellen, dass die in Simulation gewonnenen Ergebnisse (als *simulation-in-the-loop* Tests) auf die Realität übertragbar sind, wurden einige Tests mit einem echten Roboter in echter Umgebung als *hardware-in-the-loop* Tests verifiziert. Hierzu wurde die Umgebung `basic` entsprechend nachgebaut und die Tests dort mit identischer Konfiguration durchgeführt. In Abbildung 5.17 ist zu sehen, dass die aus *simulation-in-the-loop* Tests gewonnen Ergebnisse nicht von denen unter realen Einsatzbedingungen abweichen. Die in Simulation gewonnenen Ergebnisse sind somit auf reale Einsatzbedingungen übertragbar, d.h. die Simulationsumgebung bildet die realen Einsatzbedingungen hinreichend genau ab (vergleiche Kapitel 2.1.3 Simulationsumgebungen).

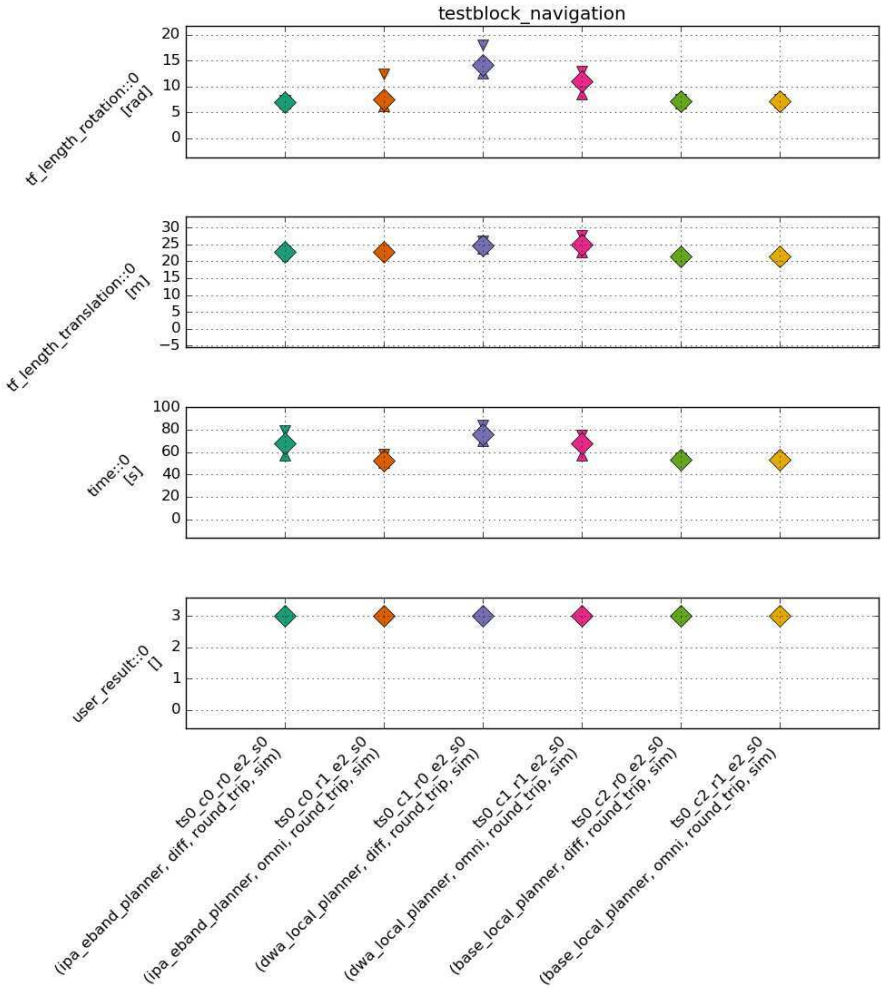


Abbildung 5.15: *ATF Ergebnisse für Umgebung* ***round_trip****. Die Unterschiede zwischen den Tests sind gering, da die Umgebung keine besonderen Hindernisse oder Engstellen enthält und alle Navigationsziele jeweils in Fahrtrichtung angefahren werden können.*

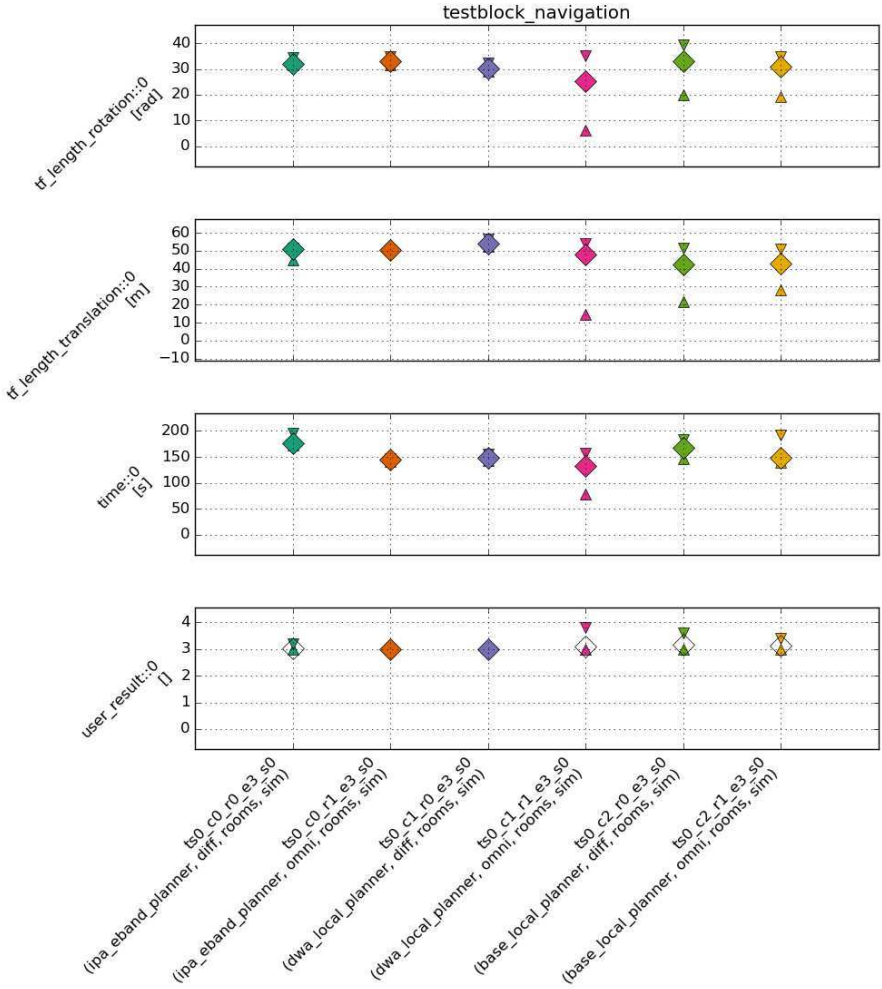


Abbildung 5.16: *ATF Ergebnisse für Umgebung* `rooms`*. Diese stellt sich für die Navigation am herausfordernsten dar, da hier Hindernisse, Engstellen und unterschiedlich orientierte Navigationsziele auf engem Raum auftreten. Für die Tests* `ts0_c0_r0_e3_s0`*,* `ts0_c1_r1_e3_s0`*,* `ts0_c2_r0_e3_s0` *und* `ts0_c2_r1_e3_s0` *ist die erfolgreiche Zielerreichung nicht immer möglich, sodass sich diese Kombinationen nicht für einen robusten Einsatz in dieser Umgebung eignen.*

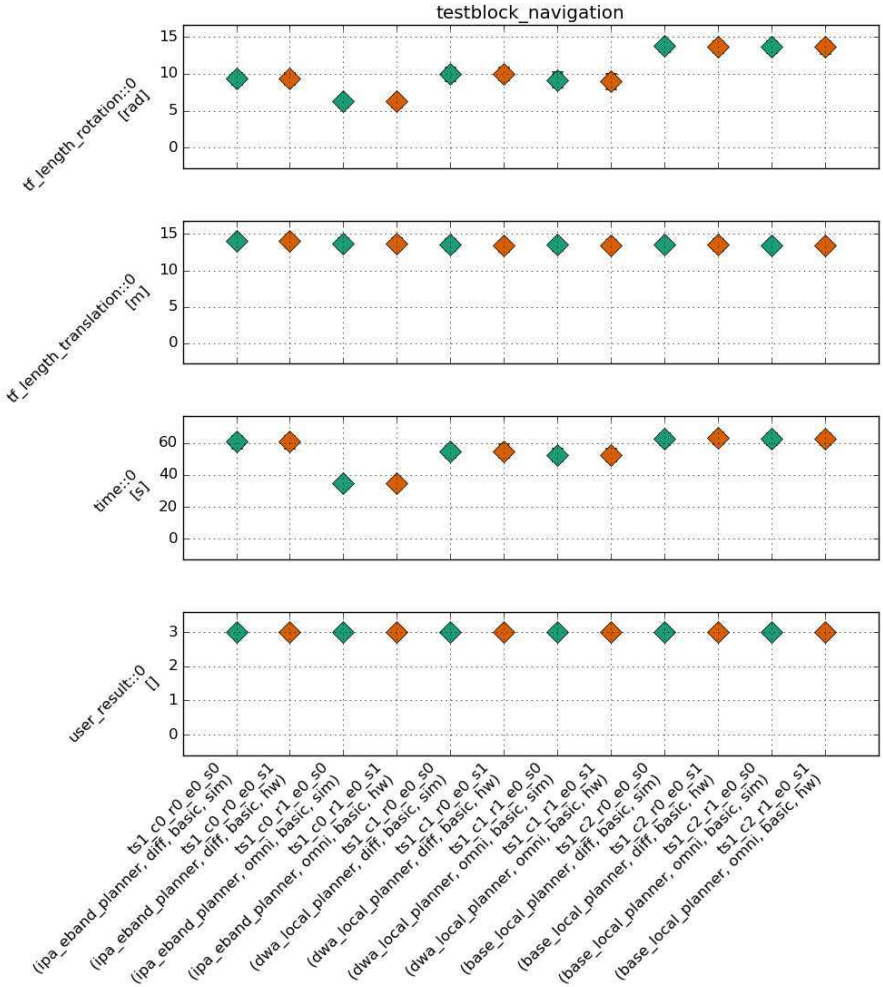


Abbildung 5.17: *Vergleich von ATF Ergebnissen für Anwendungsbeispiel 3. Vergleich von Ergebnissen aus simulation-in-the-loop Tests und hardware-in-the-loop Tests für die Umgebung* `basic`*. Die Ergebnisse aus Tests in Simulation (grün) weichen nicht wesentlich von den Ergebnissen aus Tests vom realen Roboter (orange) ab, d.h. die Simulationsumgebung bildet die realen Einsatzbedingungen hinreichend genau ab.*

Anforderungen	AF-1 Testbausteine	AF-2 *Hardware-in-the-loop*	AF-3 *Simulation-in-the-loop*	AF-4 Teststufen	AF-5 Testgenerierung	AF-6 Testauswertung	AF-7 Ergebnisvisualisierung	AF-8 Testautomatisierung	AF-9 TDD-Integration	AF-10 Benchmarking	AF-11 Minimale Beeinflussung	AF-12 ROS Integration
Anwendungsbeispiel 3	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓

Tabelle 5.10: *In Anwendungsbeispiel 3 umgesetzte Anforderungen.*

Bewertung und Fazit

Mithilfe der Benchmarkingtests des ATF kann die beste Kombination an Komponenten und Konfigurationssatz gefunden werden. Dies ermöglicht es, bereits vor der Inbetriebnahme in der Zielumgebung, eine gute Startkonfiguration für das Systemverhalten zu ermitteln und somit den Testaufwand vor Ort zu verringern.

Tabelle 5.10 zeigt die in Anwendungsbeispiel 3 validierten Anforderungen. Insbesondere wurde in diesem Anwendungsfall gezeigt, wie ATF für Benchmarking mit *simulation-in-the-loop* Tests verwendet werden kann (Anforderung AF-3 *Simulation-in-the-loop*) und die Ergebnisse visuell ausgewertet werden können (Anforderung AF-7 Ergebnisvisualisierung).

Vergleich des Testaufwands mit und ohne ATF

In Garcia Lopez (2019) wird ein ähnlicher Testaufbau wie in Anwendungsbeispiel 3 zur Auswertung des dort entwickelten Pfadplanungsalgorithmus verwendet. Es wurden Tests für unterschiedliche Pfadplaner aus dem ROS Umfeld (`dwa_local_planner`, `teb_local_planner`, `base_local_plabber` und `ipa_eband_planner`) sowie mit unterschiedlichen Roboterkinematiken (differentiell, omnidirektional und ackermann) durchgeführt. Die Tests in Garcia Lopez

(2019) wurden dabei in verschiedenen Umgebungen bzw. Umgebungsbedingungen durchgeführt.

Die Erstellung, Durchführung und Auswertung der insgesamt 216 Tests hat dabei über 500h in Anspruch genommen, da die Tests nur sequentiell ausgeführt werden konnten und jeweils manuell anhand von aufgezeichneten Daten ausgewertet wurden. In Tabelle 5.11 sind die zeitlichen Aufwände zur Durchführung der Tests aus Garcia Lopez (2019) und aus Anwendungsbeispiel 3 gegenübergestellt. Dabei wurde die Zeit für das Konfigurieren des Tests bzw. der Testumgebung pro Test für beide Fälle mit je 1h und für die Konfiguration mit ATF ein Mehraufwand von 4h für das zusätzliche Erstellen der Testsuite angesetzt. Die Durchführung in Garcia Lopez (2019) konnte nur sequentiell erfolgen, da die Tests manuell gestartet und überwacht werden mussten. Im Fall von Anwendungsbeispiel 3 laufen die Tests automatisiert ab und können auch parallelisiert werden. Als Faktor für das parallele Ausführen von Tests wurde lediglich Faktor 2 gewählt, um diese parallel auf einem normalen Desktoprechner ohne Ressourcenkonflikte durchführen zu können. Stehen entsprechend mehr Rechner bzw. Rechenressourcen zur Verfügung kann die Ausführung der ATF Tests weiter beschleunigt werden.

Insgesamt ergibt sich für eine vergleichbare Anzahl an Tests ein deutlich reduzierter Aufwand für die Durchführung und Auswertung der Tests. Würden die Tests aus Garcia Lopez (2019) erneut mithilfe von ATF durchgeführt, ergäbe sich, mit der Annahme, dass die Tests unüberwacht durchgeführt werden können, eine Reduktion des Aufwands von über 500 Stunden auf unter 50 Stunden und damit ein Verhältnis von ca. 10:1, d.h. eine 10-fache Reduktion des Aufwands.

Mit Blick auf die Ziele aus Tabelle 3.1 konnte in Anwendungsbeispiel 3 gezeigt werden, dass durch den Einsatz von ATF die Testautomatisierung (Ziel Z-EP-2) gesteigert wurde und damit die Dauer (Ziel Z-EP-3) und der Aufwand zur Testerstellung (Ziel Z-EP-6) und -durchführung (Ziel Z-EP-6) deutlich reduziert werden konnte.

	Tests in Garcia Lopez (2019) (ohne ATF)		Tests in Anwendungsbeispiel 3 (mit ATF)
	Konfiguration		
Anzahl `config`	6		3
Anzahl `robot`	3		2
Anzahl `env`	4		4
Anzahl Wiederholungen	3		10
Summe Permutationen	72		24
Summe Tests	216		240
	Aufwände pro Test		
Testerstellung	1h	=	1h
Testdurchführung	1h	>	0,2h*
Parallele Tests	1	<	2
Testauswertung	1h	>	0h
	Aufwände Gesamt		
Testerstellung	72h		24h
Setup ATF	0h		4h**
Testdurchführung	216h		24h
Testauswertung	216h		0h
Summe Aufwand	**504h** (ohne ATF)		52h
	Aufwandsreduktion durch Einsatz von ATF		
Aufwand pro Test	2,33h	>	0,22h
Aufwandsreduktion	2,33h / 0,22h = **10,77**		
Summe Aufwand	**46,8h** (mit ATF)		

Tabelle 5.11: *Vergleich des Testaufwands ohne und mit ATF. Mit dem Einsatz von ATF ergibt sich eine Aufwandsreduktion um Faktor 10 gegenüber den Tests ohne ATF. *=Annahme, dass die Tests unüberwacht durchgeführt werden können. **=Annahme, dass die nötigen Metriken bereits in ATF verfügbar sind.*

5.2 Evaluierung Ziele und Anforderungen

Im Folgenden werden die Ziele aus Kapitel 1.3 und Anforderungen aus Kapitel 3.1 evaluiert.

5.2.1 Zusammenfassung der Anforderungen anhand der Anwendungsbeispiele

Jedes der Anwendungsbeispiele aus Kapitel 5.1 beleuchtet unterschiedliche Aspekte des Einsatzes und der Umsetzung von ATF. Tabelle 5.12 zeigt eine Zusammenfassung der in den Anwendungsbeispielen 1 bis 3 evaluierten Anforderungen. Zu sehen ist, dass die Umsetzung der Anforderungen AF-1 (Bereitstellung vordefinierter Testbausteine), AF-6 (Unterstützung zur automatisierten Auswertung der Tests anhand von objektiven Metriken) und AF-7 (Bereitstellung einer graphischen Visualisierung der quantitativen Testergebnisse) in allen Anwendungsbeispielen relevant ist und damit eine wesentliche Basis für die Umsetzung eines Testframeworks darstellt. Weiterhin ist, insbesondere durch die weite Verbreitung von ROS in der Servicerobotik, Anforderung AF-12 (Integration in ROS zur einfache Nutzung im ROS Umfeld) eine wichtige Voraussetzung zur Akzeptanz und Verbreitung von ATF. Darüber hinaus reduziert ATF die Abhängigkeit der Qualität der Testergebnisse zur Person die die Tests ausführt. Durch die Automatisierung der Durchführung und Auswertung der Tests in ATF ist kein Fachexperte mehr zum Testen notwendig.

Anforderungen	AF-1 Testbausteine	AF-2 *Hardware-in-the-loop*	AF-3 *Simulation-in-the-loop*	AF-4 Teststufen	AF-5 Testgenerierung	AF-6 Testauswertung	AF-7 Ergebnisvisualisierung	AF-8 Testautomatisierung	AF-9 TDD-Integration	AF-10 Benchmarking	AF-11 Minimale Beeinflussung	AF-12 ROS Integration
Anwendungsbeispiel 1	✓			✓	✓	✓	✓	✓	✓	✓		✓
Anwendungsbeispiel 2	✓		✓		✓	✓	✓	✓		✓		✓
Anwendungsbeispiel 3	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
Zusammenfassung über alle Anwendungsbeispiele	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Tabelle 5.12: *Auswertung der Anforderungen an das ATF.*

5.2.2 Abgrenzung von ATF zu existierenden Testframeworks

Tabelle 5.13 zeigt die Abgrenzung von ATF gegenüber existierenden Testframeworks, wie sie in Kapitel 2.3.3 vorgestellt wurden. Ein wesentliches Alleinstellungsmerkmal von ATF gegenüber anderen Testframeworks in der Servicerobotik ist die Unterstützung von *Hardware-in-the-loop* sowie die Möglichkeit zur quantitativen Analyse. Weiterhin zeichnet ATF die Möglichkeit aus, verschiedene Testergebnisse zu Benchmarkingzwecken gegenüberzustellen. ATF ermöglicht es somit die Vorzüge vieler Testframeworks zusammenzuführen und bietet damit eine ganzheitliche Testumgebung, insbesondere für die Bedürfnisse in der Servicerobotikentwicklung, an.

Testframework	Methode / Prozess	Statische Codeanalyse	Unittests	Integrationstests	Systemtests	*Hardware-in-the-loop* Unterstützung	*Simulation-in-the-loop* Unterstützung	Qualitative Analyse	Quantitative Analyse	Benchmarking Unterstützung	Funktionale Tests	Deployment Tests	Regression Tests	Vordefinierte Testbausteine	CI Integration	ROS Integration
Robot Unit Testing (Bihlmaier & Wörn 2014)	✓					✗			✗	✗						
ROS Style Guide (Open Robotics 2020g)	✓					✗			✗	✗						✓
Code Review	✓	✓				✗		✓	✗	✗						
ROS-I Rating (ROS-Industrial Consortium 2020c)	✓					✗			✗	✗						✓
ROS2 Quality Rating (Open Robotics 2020m)	✓					✗			✗	✗						✓
Linter (catkin_lint, ros_lint)		✓				✗		✓	✓	✓				✓	✓	✓
ROS Code Quality (Pitzer 2012)		✓				✗		✓	✓	✓					✓	✓
Rostest (Open Robotics 2020f)			✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓		✓	✓
ROS Buildfarm (Open Robotics 2020e)		✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓		✓	✓
Buildbot ROS (Ferguson 2014)		✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓		✓	✓
ROS-Industrial CI (ROS-Industrial Consortium 2020a)		✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓		✓	✓
ROS Gitlab CI (Lamoine 2020)		✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓		✓	✓
SR Buildtools (Shadow Robotics 2020)		✓	✓			✗		✓	✗	✗						✓
DDT (Reiser 2014)	✓			✓	✓	✓			✗	✗		✓				✓
ATF	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Tabelle 5.13: *Gegenüberstellung existierender Testframeworks für die Servicerobotik und Vergleich mit ATF.*

5.3 Nutzung und Verbreitung von ATF

ATF ist mit Stand 12/2021 bereits seit drei Jahren im praktischen Einsatz. In dieser Zeit konnten wertvolle Erfahrungen aus verschiedenen Einsatzgebieten und von unterschiedlichen Entwicklungsteams gesammelt werden. Da ATF als Open Source Paket der weltweiten Gemeinschaft zu Verfügung gestellt wird, sind genaue Nutzungsstatistiken schwer zu erfassen. Im Folgenden wird die Nutzung und Akzeptanz anhand der Statistiken auf Github abgeschätzt, wobei hier von einer gewissen Unschärfe und Dunkelziffer auszugehen ist. Bessere Daten für die Erstellung von Statistiken liegen für den Einsatz bei Mojin Robotics vor, die Nachfolgend ausgewertet werden.

5.3.1 Verbreitung in der Open Source Gemeinschaft

Auf Github sind 18 offizielle Forks des ATF Repositories (Weißhardt 2021) gelistet. Dies zeigt jedoch nur diejenigen Nutzer auf, die Änderungen am Source Code gemacht bzw. beigetragen haben. Die tatsächliche Nutzerzahl liegt durch eine hohe Dunkelziffer vermutlich deutlich höher, da das Erstellen eines Forks für die eigentliche Nutzung nicht notwendig ist. Weiterhin lässt das Vorhandensein eines Forks noch keinen Rückschluss darauf zu, wie häufig und intensiv ATF verwendet wird.

Aufschlussreicher ist die Zugriffsstatistik auf Github. Jedoch lässt diese nur einen Rückschluss auf die Nutzung im Rahmen von CI zu, da das ATF Repository dafür typischerweise für jeden Durchlauf neu ausgecheckt wird. Mehrfache lokale Nutzung kann auch bei dieser Statistik nicht berücksichtigt werden, da es typischerweise kein erneutes Auschecken erfordert. Github listet im Jahresdurchschnitt ca. 100 Downloads pro Woche von ca. 20 einzelnen Nutzern auf, wovon einer Mojin Robotics ist. Abbildung 5.18 zeigt die Zugriffsstatistik über einen repräsentativen zweiwöchigen Zeitraum. Von den 100 Downloads entfallen ca. 35% auf Mojin Robotics und ca. 65% auf externe Nutzer. Damit ist Mojin Robotics einer der intensiven Nutzer von ATF und wird im Folgenden detaillierter ausgewertet.

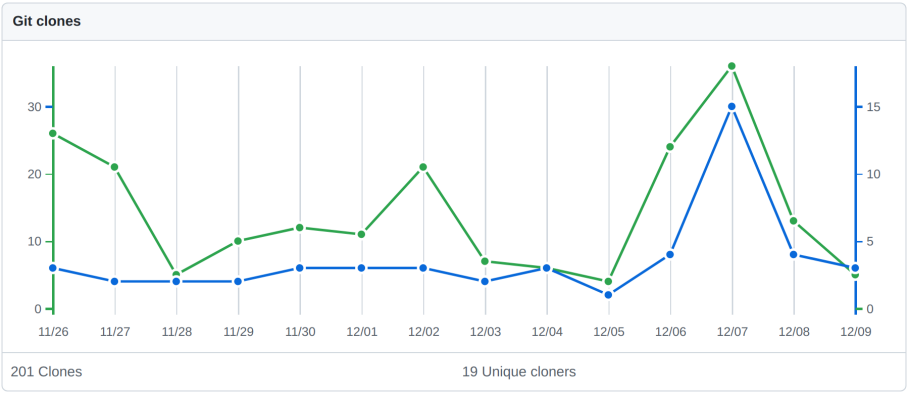


Abbildung 5.18: *Zugriffsstatistik auf ATF Github Repository. Die Statistik weist ca. 100 Downloads (Clones) pro Woche von ca. 20 einzelnen Nutzern auf. (Quelle: Github, Stand: 09.12.2021)*

5.3.2 Nutzung bei Mojin Robotics

Mojin setzt ATF hauptsächlich für entwicklungsbegleitendes und kontinuierliches Testen über einen CI-Server ein. Der Prozess ist dabei soweit automatisiert, dass die Tests sowohl regelmäßig über einen Zeitplan, als auch durch Source Code Änderungen angestoßen werden. Die Tests laufen über die CI-Services von Travis CI und Github Actions.

ATF wird bei Mojin Robotics sowohl für Komponenten-, als auch Integrations- und Systemtests eingesetzt. Tabelle 5.14 zeigt die Verwendung unterschiedlicher Pakete, mit denen über ATF getestet wird. Die Pakete definieren dabei bis zu 40 einzelne Tests und verwenden in Summe 75 Metriken für die Testauswertung. Die typische Testzeit für einen CI Durchlauf liegt dabei zwischen $1min$ und $20min$ und im Durchschnitt bei $6,35min$, was eine akzeptabel kurze Rückmeldezeit an die Entwickler ermöglicht. Es werden ca. 35 CI Durchläufe pro Woche durchgeführt. Damit lässt sich, unter Berücksichtigung des Einsparungsfaktors von 10,77 aus Kapitel 5.1.3, ein eingesparter manueller Testaufwand von ca. $35h$ pro Woche abschätzen. Dies entspricht nahezu einer Vollzeitstelle.

ATF Paket	Kategorie	Anzahl Testsuites	Anzahl Tests	Anzahl Testblöcke	Anzahl Metriken	Verwendete Metriken	Durchschnittliche Testzeit pro CI Durchlauf*	Durchschnittliche Anzahl CI Durchläufe pro Woche**	Eingesparter manueller Testaufwand pro Woche***
Anwendung I	Systemtest	2	2	1	2	M-R-1 Zeitdauer M-R-3 Hindernisabstand	7,33min	2,37	3,12h
Anwendung II	Systemtest	2	4	1	7	M-R-1 Zeitdauer M-R-3 Hindernisabstand	4,23min	7,28	5,53h
Anwendung III	Systemtest	1	5	4	20	M-R-1 Zeitdauer M-R-3 Hindernisabstand M-R-11 Benutzerergebnis	4,08min	4,25	3,11h
Manipulation PickPlace	Integrationstest	1	40	3	3	M-R-11 Benutzerergebnis	1,22min	7,14	1,56h
Manipulation Random	Integrationstest	1	10	3	3	M-R-11 Benutzerergebnis	5,65min	7,14	7,24h
Navigation Lokalisierung	Komponententest	2	5	1	4	M-R-1 Zeitdauer M-R-4 TF-Abstand M-R-9 API	3,05min	3,67	2,01h
Navigation Pfadplanung	Integrationstest	2	8	9	36	M-R-1 Zeitdauer M-R-4 TF-Abstand M-R-5 TF-Pfadlänge M-R-11 Benutzerergebnis	18,87min	3,67	12,43h
Gesamt		11	74	22	75		6,35min	35,52	**35,00**h

Tabelle 5.14: *Statistik zu Nutzung und Verbreitung von ATF bei Mojin Robotics. *=Via self-hosted Github Actions, **=Erfassungszeitraum 01/2021 bis 12/2021, ***=Abschätzung analog zu Berechnung in Kapitel 5.1.3 mit Faktor 10,77. (Quelle: Mojin Robotics, Stand: 09.12.2021)*

5.4 Fazit

In Kapitel 5 konnte gezeigt werden, dass mit der Unterstützung des Komponentenentwicklers und des Systemintegrators in der Servicerobotikentwicklung durch ATF als Testframework der Integrationsprozess bezüglich Aufwand, Zeit, erforderlichen Experten und deren Knowhow wesentlich vereinfacht werden kann. Weiterhin ist durch die effiziente Einbindung von ATF in den Entwicklungsprozess der Aufwand zur Erstellung und Durchführung von Tests wesentlich reduziert. ATF unterstützt damit die These, dass, wenn die Hürde zum Testschreiben gering ist, Tests auch genutzt und beachtet werden. Dies führt zu einer besseren Qualität der Servicerobotik-Lösungen und fördert damit die weitere Verbreitung von robusten und wirtschaftlichen Servicerobotik-Anwendungen.

6 Zusammenfassung und Ausblick

Kapitel 6.1 enthält eine Zusammenfassung der Beiträge und Ergebnisse dieser Arbeit in Bezug auf die in Kapitel 1.3 formulierten Ziele und die in Kapitel 3.1 definierten Anforderungen. Darüber hinaus wird in Kapitel 6.2 ein Ausblick auf zukünftige Erweiterungen der Umsetzung sowie die Übertragung auf weitere Bereiche beschrieben.

6.1 Zusammenfassung

Die erfolgreiche Umsetzung von komplexen Serviceroboteranwendungen ist trotz des Fortschritts bei der Auswahl an verfügbaren Hard- und Softwarekomponenten weiterhin eine Herausforderung. Insbesondere im Bereich der Servicerobotik gibt es eine Vielzahl an Anwendungen, die sich durch dynamische und oftmals unbekannte Randbedingungen auszeichnen und daher eine umfangreiche und komplexe Steuerungssoftware benötigen. Dabei ist es für einzelne Softwareentwickler oder einzelne Entwicklerteams nicht möglich alle Funktionalitäten von Grund auf neu zu implementieren, sodass diese auf vorhandene, extern entwickelte Softwarekomponenten zurückgreifen müssen. Da für die externen Komponenten meist kein Detailwissen über deren inneren Aufbau oder über die Auswirkung von veränderten Randbedingungen auf die umgesetzte Funktionalität vorhanden ist, ist für eine erfolgreiche Integration ausführliches Testen der Komponenten sowie von Teilsystemen oder des Gesamtsystems notwendig.

Weiterhin zeichnet sich die Servicerobotik durch eine hohe Abhängigkeit der einzelnen Hardware- und Softwarekomponenten aus, sodass die Integration von Komponenten in das Gesamtsystem und insbesondere die Durchführung von einzelnen Komponenten und Systemtests erschwert wird. Bisherige Entwicklungs-

methoden (z.B. Test-Driven-Development TDD), Testframeworks (z.B. gtest oder python Unittests) und Werkzeuge zur kontinuierlichen Integration (z.B. CI-Services wie Travis CI und Github Actions) unterstützen den Komponentenentwickler bei der Erstellung und Durchführung von Unittests, jedoch existieren wenige Werkzeuge, die speziell für die Herausforderungen der Servicerobotik umfassende Möglichkeiten zur Unterstützung bei der Systemintegration bieten.

In der vorliegenden Arbeit wird daher die folgende **Forschungsfrage** untersucht:

Wie kann während der Entwicklungsphase einer Servicerobotik-Anwendung der Entwicklungs- und Testaufwand reduziert, die Testabdeckung gesteigert sowie die Robustheit des Gesamtsystems im späteren Betrieb erhöht werden?

In dieser Arbeit wird die Methode des Test-Driven-Development für die Servicerobotik aufgegriffen und um ein Werkzeug zum Testen und Vergleichen von komponentenbasierter Software erweitert, das sich für Integrationstests und Benchmarkingvergleiche nutzen lässt. Dazu wird ein Test- und Evaluierungsframework, das ***Automated Test Framework* (ATF)** konzipiert und umgesetzt.

Das ATF ist ein einfach in bestehende Anwendungen zu integrierendes Framework, das es ermöglicht Servicerobotik-Anwendungen zu testen und verschiedene Varianten miteinander zu vergleichen (Benchmarking). Weiter kann ATF entwicklungsbegleitend in einer TDD-Entwicklungsumgebung eingesetzt werden um den Fortschritt im Entwicklungsprozess zu überwachen. Insbesondere im Kontext der TDD Entwicklungsmethode bietet ATF die Möglichkeit Integrations- und Systemtests intensiv, jedoch ohne großen Aufwand zu nutzen. Durch die Integration in einen CI-Service kann ATF entwicklungsbegleitend genutzt werden. Mithilfe von ATF kann der Aufwand für die Erstellung, Durchführung und Auswertung von Tests deutlich gesenkt werden. ATF ermöglicht es verschiedene Varianten eines zu testenden Systems zu untersuchen. Dies können unterschiedliche Software und Hardwarekomponenten oder unterschiedliche Konfigurationen dieser Komponenten sein.

ATF trennt dabei die Schritte Generierung von Tests, Ausführen der Anwendung mit Aufnahme von Testdaten, Analyse der Testdaten sowie Auswertung

und Visualisierung der Testergebnisse. Dies ermöglicht eine Parallelisierung von mehreren Tests sowie eine effiziente Ausnutzung vorhandener Testressourcen. ATF ist in das in der Servicerobotik gebräuchliche Framework ROS integriert, sodass ROS Komponenten sowie deren Schnittstellen mit wenig Aufwand getestet werden können.

Die Auswertung der ATF Tests basiert auf Metriken, die während der Ausführung der Tests anfallende Daten, wie z.B. Gelenkwinkel oder kartesische Bewegungen, auswerten. Weiterhin kann die Durchführung der Tests mit ATF automatisiert werden. Dies ermöglicht es Tests auch ohne Expertenwissen durchzuführen und liefert objektive Ergebnisse. Die qualitativen und quantitativen Testergebnisse können mithilfe von ATF visualisiert und verglichen werden. Dies ermöglicht den Einsatz von ATF als Benchmarkingwerkzeug, mit dem die Auswahl geeigneter Komponenten oder Konfigurationen ermöglicht wird.

Im Rahmen von Anwendungsbeispielen aus verschiedenen Bereichen der Servicerobotik wird der Einsatz von ATF aufgezeigt und dessen Nutzen erfolgreich nachgewiesen und evaluiert. Als Anwendungsbeispiele werden je ein Anwendungsfall aus den Bereichen Perzeption, Manipulation und Navigation ausgewählt. Mit den Anwendungsbeispielen kann gezeigt werden, dass ATF die in Kapitel 1.3 formulierten Ziele und die in Kapitel 3.1 definierten Anforderungen erfüllt. Insbesondere kann in Anwendungsbeispiel 3 in einem Systemtest für die Navigation eine Aufwandsreduktion um den Faktor 10 nachgewiesen werden.

6.2 Ausblick

Durch den modularen Aufbau von ATF ergeben sich weitreichende Anpassungs- und Erweiterungsmöglichkeiten. ATF kann z.B. in Zukunft um weitere generische oder auch anwendungsspezifische Metriken erweitert werden, sodass der Einsatz von ATF in neuen Anwendungsfällen ermöglicht wird.

Weiterhin könnte ATF im Zusammenspiel mit modellbasierter Softwareentwicklung genutzt werden. Hier ist eine Verknüpfung mit modellbasierten Tools (z.B. BRIDE (Bubeck et al. 2014) oder SERONET (SERONET Consortium

2020)) denkbar, sodass aus den Komponentenmodellen automatisch auch Testmodelle mit den entsprechenden Gegenschnittstellen und entsprechende Implementierung von Testkomponenten erzeugt werden. Hierdurch könnte ein Komponentenersteller mihilfe von ATF Bausteinen ebenfalls ein entsprechendes Testsystem für seine Komponente erstellen und somit durch Tests bereits früh im Entwicklungsprozess unterstützt werden.

Literatur

Absint 2020a Absint, 2020.
Astrée Laufzeitfehleranalyse,
URL: https://www.absint.com/astree/index_de.htm
Zugriff am: 01.01.2020

Absint 2020b Absint, 2020.
RuleChecker: Automatische Überprüfung von MISRA-Regeln,
URL:
https://www.absint.com/rulechecker/index_de.htm
Zugriff am: 01.01.2020

Aguirre 2007 Aguirre, Marco Antonio Morales, 2007.
Metrics for sampling-based motion planning.
Texas, Texas A&M University, PhD Thesis, 2007.
URN: https://hdl.handle.net/1969.1/ETD-TAMU-2462

Albus 2018 Albus, Marcel, 2018.
Analyse und Optimierung von Parametersets für Bewegungsregelungsverfahren für mobile Service-Roboter in dynamischer Umgebung.
Stuttgart, Universität Stuttgart, ISW, Studienarbeit, 2018

Ando et al. 2005 Ando, Noriaki; Suehiro, Takashi; Kitagaki, Kosei; Kotoku, Tetsuo; Yoon, Woo-Keun, 2005.
RT-middleware: distributed component middleware for RT (robot technology).
In: IEEE (Hrsg.):
International Conference on Intelligent Robots and Systems IROS.
Edmonton, Canada, 02.08.2005–06.08.2005, S. 3933–3938.
DOI: 10.1109/IROS.2005.1545521

Axivion 2020 Axivion, 2020.
Axivion Bauhaus Suite,
URL: https://www.axivion.com/de/produkte-58#produkte_bauhaussuite
Zugriff am: 01.01.2020

Baker et al. 2011
Baker, Christopher; Dolan, John; Wang, Shige; Litkouhi, Bakhtiar, 2011.
Toward adaptation and reuse of advanced robotic software.
In: IEEE (Hrsg.):
International Conference on Robotics and Automation ICRA.
Shanghai, China, 09.05.2011–13.05.2011, S. 6071–6077.
DOI: 10.1109/ICRA.2011.5980355

Balaguer et al. 2007
Balaguer, Benjamin; Carpin, Stefano; Balakirsky, Stephen, 2007.
Towards quantitative comparisons of robot algorithms: Experiences with SLAM in simulation and real world systems.
In: IEEE (Hrsg.):
Conference on Intelligent Robots and Systems IROS, Workshop.
San Diego, USA, 29.10.2007–02.11.2007

Baltes 2000
Baltes, Jacky, 2000.
A benchmark suite for mobile robots.
In: IEEE/RSJ (Hrsg.):
International Conference on Intelligent Robots and Systems IROS.
Takamatsu, Japan, 30.10.2000–05.11.2000, S. 1101–1106.
DOI: 10.1109/IROS.2000.893166

Bartels & Beetz 2019
Bartels, Georg; Beetz, Michael, 2019.
Perception-Guided Mobile Manipulation Robots for Automation of Warehouse Logistics.
In: *KI-Künstliche Intelligenz*.
Springer, S. 189–192.
DOI: 10.1007/s13218-019-00585-2

Beck et al. 2001
Beck, Kent; Beedle, Mike; Van Bennekum, Arie; Cockburn, Alistair; Cunningham, Ward; Fowler, Martin; Grenning, James; Highsmith, Jim; Hunt, Andrew; Jeffries, Ron et al., 2001.
Manifesto for agile software development

Beck 2003
Beck, Kent, 2003.
Test-driven development: by example. Addison-Wesley Professional.
ISBN: 978-0321146533

Beck & Andres 2004
Beck, Kent; Andres, Cynthia, 2004.
Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional.
ISBN: 978-0-321-27865-4

Beckert 2016
Beckert, Bernd et al., 2016.
Automatisierung und Robotik-Systeme, Studien zum deutschen Innovationssystem No. 11-2016.
URL: https://www.e-fi.de/fileadmin/Assets/Studien/2016/StuDIS_11_2016.pdf
Zugriff am: 01.01.2020

Bengel et al. 2009
Bengel, Matthias; Pfeiffer, Kai; Graf, Birgit; Bubeck, Alexander; Verl, Alexander, 2009.
Mobile robots for offshore inspection and manipulation.
In: IEEE (Hrsg.):
Conference on Intelligent Robots and Systems IROS.
St. Louis, USA, 11.10.2009–15.10.2009, S. 3317–3322.
DOI: 10.1109/IROS.2009.5353885

Bihlmaier & Wörn 2014
Bihlmaier, Andreas; Wörn, Heinz, 2014.
Robot unit testing.
In: Springer (Hrsg.):
International Conference on Simulation, Modeling, and Programming for Autonomous Robots.
Cham, S. 255–266.
DOI: 10.1007/978-3-319-11900-7_22

Billingsley et al. 2008
Billingsley, John; Visala, Arto; Dunn, Mark, 2008.
Robotics in agriculture and forestry.
Springer handbook of robotics, S. 1065–1077.
URL: 978-3-319-32550-7

Bröhl 1993
Bröhl, Adolf-Peter, 1993.
Das V-Modell: Der Standard für die Softwareentwicklung mit Praxisleitfaden. Oldenbourg.
ISBN: 9783486234701

Broy & Rausch 2005
Broy, Manfred; Rausch, Andreas, 2005.
Das neue V-Modell XT – Ein anpassbares Vorgehensmodell für Software und System Engineering.
Informatik Spektrum **28** (3), S. 220–229.
DOI: 10.1007/s00287-005-0488-z

Brugali & Scandurra 2009
Brugali, Davide; Scandurra, Patrizia, 2009.
Component-based robotic engineering (part i)[tutorial].
IEEE Robotics & Automation Magazine **16** (4), S. 84–96.
DOI: 10.1109/MRA.2009.934837

Bruyninckx 2001
Bruyninckx, Herman, 2001.
Open robot control software: the OROCOS project.
In: IEEE (Hrsg.):
International Conference on Robotics and Automation ICRA.
Seoul, Korea, 21.05.2001–26.05.2001, S. 2523–2528.
DOI: 10.1109/ROBOT.2001.933002

Bubeck et al. 2012
Bubeck, Alexander; Weißhardt, Florian; Sing, Tobias; Reiser, Ulrich; Hägele, Martin; Verl, Alexander, 2012.
Implementing best practices for systems integration and distributed software development in service robotics-the Care-O-bot robot family.
In: IEEE (Hrsg.):
International Symposium on System Integration (SII).
Fukuoka, Japan, 16.12.2012–18.12.2012, S. 609–614.
DOI: 10.1109/SII.2012.6427386

Bubeck et al. 2014
Bubeck, Alexander; Weißhardt, Florian; Verl, Alexander, 2014.
BRIDE-A toolchain for framework-independent development of industrial service robot applications.
In: VDE (Hrsg.):
41st International Symposium on Robotics ISR.
München, 02.06.2014–04.06.2014, S. 1–6.
ISBN: 978-3-8007-3601-0

Calisi et al. 2008
Calisi, Daniele; Iocchi, Luca; Nardi, Daniele, 2008.
A unified benchmark framework for autonomous mobile robots and vehicles motion algorithms (MoVeMA benchmarks).
In:
Workshop on experimental methodology and benchmarking in robotics research (RSS 2008)

Carpin et al. 2007
Carpin, Stefano; Lewis, Mike; Wang, Jijun; Balakirsky, Stephen; Scrapper, Chris, 2007.
USARSim: a robot simulator for research and education.
In: IEEE (Hrsg.):
International Conference on Robotics and Automation ICRA.
Roma, Italy, 10.04.2011–14.04.2007, S. 1400–1405.
DOI: 10.1109/ROBOT.2007.363180

Ceriani et al. 2009
Ceriani, Simone; Fontana, Giulio; Giusti, Alessandro; Marzorati, Daniele; Matteucci, Matteo; Migliore, Davide; Rizzi, Davide; Sorrenti, Domenico G; Taddei, Pierluigi, 2009.
Rawseeds ground truth collection systems for indoor self-localization and mapping.
Autonomous Robots **27** (4), S. 353.
DOI: 10.1007/s10514-009-9156-5

Cervera 2006
Cervera, Enric, 2006.
Cross-Platform Software for Benchmarks on Visual Servoing.
In: IEEE (Hrsg.):
IROS Workshop on Benchmarks in Robotics Research.
Beijing, China

Chemuturi 2006
Chemuturi, Murali, 2006.
Test effort estimation.
Chemuturi Consultants

Collett et al. 2005
Collett, Toby; MacDonald, Bruce; Gerkey, Brian, 2005.
Player 2.0: Toward a practical robot programming framework.
In:
Proceedings of the Australasian conference on robotics and automation (ACRA 2005),
S. 145

Côté et al. 2007
Côté, Carle; Létourneau, Dominic; Raievsky, Clément; Brosseau, Yannick; Michaud, François, 2007.
Using marie for mobile robot component development and integration.
In: *Software Engineering for Experimental Robotics*.
Springer, S. 211–230.
DOI: 10.1007/978-3-540-68951-5_12

Dillmann 2004
Dillmann, Rüdiger, 2004.
Benchmarks for robotics research.
EURON

Durieux et al. 2019
Durieux, Thomas; Abreu, Rui; Monperrus, Martin; Bissyandé, Tegawendé F; Cruz, Luis, 2019.
An analysis of 35+ million jobs of Travis CI.
In:
2019 IEEE International Conference on Software Maintenance and Evolution (ICSME),
S. 291–295.
DOI: 10.1109/ICSME.2019.00044

Duvall et al. 2007
Duvall, Paul M; Matyas, Steve; Glover, Andrew, 2007.
Continuous integration: improving software quality and reducing risk. Pearson Education.
ISBN: 978-0321336385

Echeverria et al. 2011
Echeverria, Gilberto; Lassabe, Nicolas; Degroote, Arnaud; Lemaignan, Séverin, 2011.
Modular open robots simulation engine: Morse.
In: IEEE (Hrsg.):
International Conference on Robotics and Automation ICRA.
Shanghai, China, 09.05.2011–13.05.2011, S. 46–51.
DOI: 10.1109/ICRA.2011.5980252

Eckel & Allison 2016
Eckel, Bruce; Allison, Chuck, 2016.
Thinking in C++ Vol 2 - Practical Programming.
Linuxtopia

Edwards & Lewis 2012 Edwards, Shaun; Lewis, Chris, 2012.
ROS-Industrial: applying the robot operating system (ROS) to industrial applications.
In: IEEE (Hrsg.):
International Conference on Robotics and Automation ICRA, ECHORD Workshop.
Saint Paul, USA, 14.05.2012–18.05.2012

Elkady & Sobh 2012 Elkady, Ayssam; Sobh, Tarek, 2012.
Robotics middleware: A comprehensive literature survey and attribute-based bibliography.
Journal of Robotics.
DOI: 10.1155/2012/959013

Elkmann et al. 2009 Elkmann, Norbert; Hortig, Justus; Fritzsche, Markus, 2009.
Cleaning automation.
In: *Springer Handbook of Automation*.
Springer, S. 1253–1264.
ISBN: 978-3-030-96728-4

Endres et al. 1998 Endres, Hermann; Feiten, Wendelin; Lawitzky, Gisbert, 1998.
Field test of a navigation system: Autonomous cleaning in supermarkets.
In: IEEE (Hrsg.):
International Conference on Robotics and Automation ICRA.
Leuven, Belgium, 16.05.1998–20.05.1998, S. 1779–1781.
DOI: 10.1109/ROBOT.1998.677424

Erdogmus et al. 2005 Erdogmus, Hakan; Morisio, Maurizio; Torchiano, Marco, 2005.
On the effectiveness of the test-first approach to programming.
IEEE Transactions on software Engineering **31** (3), S. 226–237.
DOI: 10.1109/TSE.2005.37

Estivill-Castro et al. 2018 Estivill-Castro, Vladimir; Hexel, René; Lusty, Carl, 2018.
Continuous Integration for Testing Full Robotic Behaviours in a GUI-stripped Simulation.
In:
MODELS Workshops,
S. 453–464.
URN: http://hdl.handle.net/10072/381588

EuRobotics aisbl 2015 EuRobotics aisbl, 2015.
Robotics 2020 Multi-Annual Roadmap for Robotics in Europe.
Call 2 ICT 24 Horizon 2020

Ferguson 2014
Ferguson, Mike, 2014.
Continuous Integration for ROS in Commercial Environments.
In: OSRF (Hrsg.):
ROSCon.
Chicago, USA, 12.09.2014–13.09.2014

Fernández-Madrigal et al. 2009
Fernández-Madrigal, Juan-Antonio; Cruz-Martin, Ana; Galindo, Cipriano; González, Javier, 2009.
Heterogeneity as a corner stone of software development in robotics.
In: *Software Engineering and Development*.
Nova Publishers, S. 13–22.
ISBN: 978-1-60692-146-3

Fitzpatrick et al. 2008
Fitzpatrick, Paul; Metta, Giorgio; Natale, Lorenzo, 2008.
Towards long-lived robot genes.
Robotics and Autonomous Systems **56** (1), S. 29–45.
DOI: 10.1016/j.robot.2007.09.014

Fraunhofer IPA 2020
Fraunhofer IPA, 2020.
Care-O-bot 4,
URL: http://www.care-o-bot-4.de/
Zugriff am: 01.01.2020

Garcia Lopez 2019
Garcia Lopez, Felipe, 2019.
Predictive and cooperative online motion planning: a contribution to networked mobile robot navigation in industrial applications.
Stuttgart, Universität Stuttgart, Dissertation, 2019.
DOI: 10.18419/opus-10715

Geraerts 2006
Geraerts, Roland Jan, 2006.
Sampling-based motion planning: Analysis and path quality.
Utrecht, Netherlands, Utrecht University, PhD Thesis, 2006.
URN: http://hdl.handle.net/1874/8771

Gerkey et al. 2003
Gerkey, Brian; Vaughan, Richard; Howard, Andrew, 2003.
The player/stage project: Tools for multi-robot and distributed sensor systems.
In:
Proceedings of the 11th international conference on advanced robotics ICAR.
Coimbra, Portugal, 30.06.2003–03.07.2003, S. 317–323

Github 2020a
Github, 2020.
Github Actions,
URL: https://github.com/features/actions
Zugriff am: 01.12.2020

Github 2020b
Github, 2020.
Github Code Review,
URL: https://github.com/features/code-review
Zugriff am: 01.12.2020

Graf et al. 2012
Graf, Birgit; Jacobs, Theo; Luz, Jochen; Compagna, Diego; Derpmann, Stefan; Shire, Karen, 2012.
Einsatz und Pilotierung mobiler Serviceroboter zur Unterstützung von Dienstleistungen in der stationären Altenpflege.
In: *Technologiegestützte Dienstleistungsinnovation in der Gesundheitswirtschaft.*
Springer, S. 265–288.
ISBN: 978-3-8349-3505-2

Grechenig 2010
Grechenig, Thomas, 2010.
Softwaretechnik: mit Fallbeispielen aus realen Entwicklungsprojekten. Pearson Deutschland GmbH.
ISBN: 978-3-86894-007-7

Grunwald et al. 2008
Grunwald, Gerhard; Borst, Christoph; Zöllner, Marius, 2008.
Benchmarking dexterous dual-arm/hand robotic manipulation.
In: IEEE (Hrsg.):
Conference on Intelligent Robots and Systems IROS, Workhop on Performance Evaluation and Benchmarking for Intelligent Robots and Systems.
Nice, France, 22.09.2008–26.09.2008

Hägele & Pfeiffer 2007
Hägele, Martin; Pfeiffer, Kai, 2007.
RoSta Deliverable D4.1: Report on State of the Art on Benchmarks for Mobile Manipulation and Service Robots.
RoSta Project Deliverables

Hägele, Martin et al. 2011
Hägele, Martin et al., 2011.
Wirtschaftlichkeitsanalysen neuartiger Servicerobotik-Anwendungen und ihre Bedeutung für die Robotik-Entwicklung. Eine Analyse der Fraunhofer Institute IPA (Stuttgart) und ISI (Karlsruhe) im Auftrag des BMBF.

Hájek et al. 2011
Hájek, Alan et al., 2011.
Interpretations of probability.
In: *Stanford Encyclopedia of Philosophy.*
Stanford University

Hebert et al. 2012
Hebert, Martial H; Thorpe, Charles E; Stentz, Anthony, 2012.
Intelligent unmanned ground vehicles: autonomous navigation research at Carnegie Mellon. Springer Science & Business Media.
ISBN: 978-1-4613-7904-1

Heise 2020
Heise, 2020.
ROS-Industrial: Open Source etabliert sich auch in der Industrierobotik,
URL: https://www.heise.de/news/ROS-Industrial-Open-Source-etabliert-sich-auch-in-der-Industrierobotik-4991132.html
Zugriff am: 01.12.2020

Henkel et al. 2012
Henkel, Zachary; Murphy, Robin; Srinivasan, Vasant; Bethel, Cindy L, 2012.
A proxemic-based HRI testbed.
In:
Proceedings of the Workshop on Performance Metrics for Intelligent Systems PerMIS,
S. 75–81.
DOI: 10.1145/2393091.2393108

Hernandez et al. 2016
Hernandez, Carlos; Bharatheesha, Mukunda; Ko, Wilson; Gaiser, Hans; Tan, Jethro; van Deurzen, Kanter; de Vries, Maarten; Van Mil, Bas; van Egmond, Jeff; Burger, Ruben et al., 2016.
Team Delft's robot winner of the amazon picking challenge 2016.
In: Springer (Hrsg.):
Robot World Cup.
2016, S. 613–624.
DOI: 10.1007/978-3-319-68792-6_51

Hicks & Hall 2000
Hicks, Rob Warren II; Hall, Ernest L., 2000.
Survey of robot lawn mowers.
In: For Optics, International Society; Photonics (Hrsg.):
Intelligent Robots and Computer Vision XIX: Algorithms, Techniques, and Active Vision,
S. 262–269.
DOI: 10.1117/12.403770

Hojo 2004
Hojo, Taisuke, 2004.
Quality management systems: process validation guidance.
Global Harmonization Task Force, Study Group

IEEE 1061:1998
IEEE 1061:1998.
IEEE Standard for a Software Quality Metrics Methodology

IFR 2020a
IFR, 2020.
IFR Service Robots,
URL: https://www.ifr.org/service-robots
Zugriff am: 01.01.2020

IFR 2020b
IFR, 2020.
World Robotics Report for Service Robotics

Iossifidis et al. 2005
Iossifidis, Ioannis; Lawitzky, Gisbert; Knoop, Steffen; Zöllner, Raoul, 2005.
Towards benchmarking of domestic robotic assistants.
In:
Springer, S. 403–414.
DOI: 10.1007/978-3-540-31509-4_33

ISO 25010:2011
ISO 25010:2011.
System and software quality models

ISO 8373:2012
ISO 8373:2012.
Robots and robotic devices

Jimenez et al. 2006
Jimenez, J.; Rano, I.; Minguez, J., 2006.
Advances in the framework for automatic evaluation of obstacle avoidance methods.
In: IEEE (Hrsg.):
Conference on Intelligent Robots and Systems IROS, Workshop.
Beijing, China, 09.10.2006–15.10.2006

Jones 2006
Jones, Joseph L, 2006.
Robots at the tipping point: the road to iRobot Roomba.
IEEE Robotics & Automation Magazine **13** (1), S. 76–78.
DOI: 10.1109/MRA.2006.1598056

Kittmann et al. 2015
Kittmann, Ralf; Fröhlich, Tim; Schäfer, Johannes; Reiser, Ulrich; Weißhardt, Florian; Haug, Andreas, 2015.
Let me Introduce Myself: I am Care-O-bot 4, a Gentleman Robot.
In: Diefenbach, Sarah; Henze, Niels; Pielot, Martin (Hrsg.):
Mensch und Computer 2015 – Proceedings,
S. 223–232.
DOI: 10.1515/9783110443929-024

Koenig & Howard 2004
Koenig, Nathan; Howard, Andrew, 2004.
Design and use paradigms for gazebo, an open-source multi-robot simulator.
In: IEEE (Hrsg.):
Conference on Intelligent Robots and Systems IROS.
Sendai, Japan, 28.09.2004–02.10.2004, S. 2149–2154.
DOI: 10.1109/IROS.2004.1389727

Krotkov et al. 2017
Krotkov, Eric; Hackett, Douglas; Jackel, Larry; Perschbacher, Michael; Pippine, James; Strauss, Jesse; Pratt, Gill; Orlowski, Christopher, 2017.
The DARPA robotics challenge finals: results and perspectives.
Journal of Field Robotics **34** (2), S. 229–240.
DOI: 10.1007/978-3-319-74666-1_1

Kuhn & Reilly 2002
Kuhn, D Richard; Reilly, Michael J, 2002.
An investigation of the applicability of design of experiments to software testing.
In: IEEE (Hrsg.):
27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.
S. 91–95

Lamoine 2020
Lamoine, Victor, 2020.
ROS Gitlab CI,
URL:
https://gitlab.com/VictorLamoine/ros_gitlab_ci
Zugriff am: 01.01.2020

Lier et al. 2012
Lier, Florian; Schulz, Simon; Lütkebohle, Ingo, 2012.
Continuous integration for iterative validation of simulated robot models.
In:
International Conference on Simulation, Modeling, and Programming for Autonomous Robots,
S. 101–112.
DOI: 10.1007/978-3-642-34327-8_12

Llarena & Rojas 2016
Llarena, Adalberto; Rojas, Raúl, 2016.
I am Alleine, the autonomous wheelchair at your service.
In: *Intelligent Autonomous Systems 13*.
Springer, S. 1613–1626.
DOI: 10.1007/978-3-319-08338-4_116

Madeyski 2009
Madeyski, Lech, 2009.
Test-driven development: An empirical evaluation of agile practice. Springer Science & Business Media.
ISBN: 978-3-642-04287-4

Madhavan & Messina 2006
Madhavan, Raj; Messina, Elena, 2006.
Performance Metrics for Intelligent Systems (PerMIS) 2006 Workshop: Summary and Review.
In: IEEE (Hrsg.):
35th IEEE Applied Imagery and Pattern Recognition Workshop (AIPR'06),
S. 31–31

Madhavan et al. 2009a
Madhavan, Raj; Lakaemper, Rolf; Kalmár-Nagy, Tamás, 2009.
Benchmarking and standardization of intelligent robotic systems.
In:
2009 International Conference on Advanced Robotics,
S. 1–7.
ISBN: 978-1-4244-4855-5

Madhavan et al. 2009b Madhavan, Raj; Tunstel, Edward; Messina, Elena, 2009. *Performance evaluation and benchmarking of intelligent systems*. Springer. ISBN: 978-1-4419-0491-1

Marder-Eppstein et al. 2010 Marder-Eppstein, Eitan; Berger, Eric; Foote, Tully; Gerkey, Brian; Konolige, Kurt, 2010. The office marathon: Robust navigation in an indoor office environment. In: IEEE (Hrsg.): International Conference on Robotics and Automation ICRA. Anchorage, USA, 03.05.2010–07.05.2010, S. 300–307. DOI: 10.1109/ROBOT.2010.5509725

Mari & Eila 2003 Mari; Eila, 2003. The impact of maintainability on component-based software systems. In: IEEE (Hrsg.): Euromicro Conference, S. 25–25. DOI: 10.1109/EURMIC.2003.1231563

McConnell 1999 McConnell, Steve, 1999. Open-source methodology: Ready for prime time. *IEEE software* **16** (4), S. 6–8

Metta et al. 2006 Metta, Giorgio; Fitzpatrick, Paul; Natale, Lorenzo, 2006. YARP: yet another robot platform. *International Journal of Advanced Robotic Systems* **3** (1), S. 8. DOI: 10.5772/5761

Meyer 2014 Meyer, Mathias, 2014. Continuous Integration and Its tools. *IEEE software* **31** (3), S. 14–16. DOI: 10.1109/MS.2014.58

MISRA 2008 MISRA, Motor Industry Software Reliability Association, 2008. *MISRA C++ 2008: guidelines for the use of the C++ language in critical systems*. MISRA

Moll et al. 2015 Moll, Mark; Sucan, Ioan; Kavraki, Lydia, 2015. Benchmarking motion planning algorithms: An extensible infrastructure for analysis and visualization. *IEEE Robotics & Automation Magazine* **22** (3), S. 96–102. DOI: 10.1109/MRA.2015.2448276

Müller & Padberg 2003 Müller, Matthias; Padberg, Frank, 2003.
About the return on investment of test-driven development.
In:
Edser-5 5 th international workshop on economic-driven software engineering research,
S. 26.
DOI: 10.5445/IR/1000061750

Muñoz et al. 2007 Muñoz, N.; Valencia, J.; Londono, N., 2007.
Evaluation of navigation of an autonomous mobile robot.
In:
Proceedings of the 2007 Workshop on Performance Metrics for Intelligent Systems,
S. 15–21.
DOI: 10.1145/1660877.1660878

Nowak et al. 2010 Nowak, Walter; Zakharov, Alexey; Blumenthal, Sebastian; Prassler, Erwin, 2010.
BRICS Deliverable D3.1: Benchmarks for mobile manipulation and robust obstacle avoidance and navigation.
BRICS Project Deliverables

Open Robotics 2020a Open Robotics, 2020.
C++ unit testing framework for ROS,
URL: http://wiki.ros.org/gtest
Zugriff am: 01.01.2020

Open Robotics 2020b Open Robotics, 2020.
Open-source Robot operating system ROS,
URL: https://www.ros.org/
Zugriff am: 01.01.2020

Open Robotics 2020c Open Robotics, 2020.
Python unit testing framework for ROS,
URL: http://wiki.ros.org/unittest
Zugriff am: 01.01.2020

Open Robotics 2020d Open Robotics, 2020.
ROS bags,
URL: http://wiki.ros.org/Bags
Zugriff am: 01.01.2020

Open Robotics 2020e Open Robotics, 2020.
ROS Buildfarm,
URL: http://build.ros.org/
Zugriff am: 01.01.2020

Open Robotics 2020f Open Robotics, 2020.
ROS rostest framework,
URL: http://wiki.ros.org/rostest
Zugriff am: 01.01.2020

Open Robotics 2020g Open Robotics, 2020.
ROS style guide,
URL: http://wiki.ros.org/StyleGuide
Zugriff am: 01.01.2020

Open Robotics 2020h Open Robotics, 2020.
ROS style guide roscpp,
URL: http://wiki.ros.org/CppStyleGuide
Zugriff am: 01.01.2020

Open Robotics 2020i Open Robotics, 2020.
ROS style guide rospy,
URL: http://wiki.ros.org/PyStyleGuide
Zugriff am: 01.01.2020

Open Robotics 2020j Open Robotics, 2020.
ROS Unit Testing,
URL:
http://wiki.ros.org/Quality/Tutorials/UnitTesting
Zugriff am: 01.01.2020

Open Robotics 2020k Open Robotics, 2020.
ROS wiki,
URL: http://wiki.ros.org/
Zugriff am: 01.01.2020

Open Robotics 2020l Open Robotics, 2020.
ROS wrapper for Alvar, an open-source AR tag tracking library,
URL: http://wiki.ros.org/ar_track_alvar
Zugriff am: 01.01.2020

Open Robotics 2020m Open Robotics, 2020.
ROS2 Package Quality Categories,
URL: https://www.ros.org/reps/rep-2004.html
Zugriff am: 01.01.2020

Pitzer 2012 Pitzer, Benjamin, 2012.
Code Quality in ROS.
In: OSRF (Hrsg.):
ROSCon.
St. Paul, USA, 19.05.2012–20.05.2012

Quigley et al. 2009 Quigley, Morgan; Conley, Ken; Gerkey, Brian; Faust, Josh; Foote, Tully; Leibs, Jeremy; Wheeler, Rob; Ng, Andrew Y, 2009.
ROS: an open-source Robot Operating System.
In: IEEE (Hrsg.):
ICRA workshop on open-source software.
Kobe, Japan, 12.05.2009–17.05.2009, S. 5

Rafique & Mišić 2012 Rafique, Yahya; Mišić, Vojislav B, 2012. The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering* **39** (6), S. 835–856. DOI: 10.1109/TSE.2012.28

Reichardt et al. 2013 Reichardt, Max; Föhst, Tobias; Berns, Karsten, 2013. On software quality-motivated design of a real-time framework for complex robot control systems. *Electronic Communications of the EASST* **60**. DOI: 10.14279/tuj.eceasst.60.855

Reiser 2014 Reiser, Ulrich, 2014. *Eine webbasierte Integrations-und Testplattform zur Unterstützung des verteilten Entwicklungsprozesses von komplexen Serviceroboter-Applikationen*. Stuttgart, Universität Stuttgart, Dissertation, 2014. DOI: 10.18419/opus-6848

Rohmer et al. 2013 Rohmer, Eric; Singh, Surya PN; Freese, Marc, 2013. V-REP: A versatile and scalable robot simulation framework. In: IEEE (Hrsg.): Conference on Intelligent Robots and Systems IROS. Tokyo, Japan, 03.11.2013–08.11.2013, S. 1321–1326. DOI: 10.1109/IROS.2013.6696520

ROS-Industrial Consortium 2020a ROS-Industrial Consortium, 2020. *ROS-Industrial CI for Travis CI*, URL: https://github.com/ros-industrial/industrial_ci Zugriff am: 01.01.2020

ROS-Industrial Consortium 2020b ROS-Industrial Consortium, 2020. *ROS-Industrial Initiative*, URL: https://rosindustrial.org/ Zugriff am: 01.01.2020

ROS-Industrial Consortium 2020c ROS-Industrial Consortium, 2020. *ROS-Industrial Software Status Rating*, URL: http://wiki.ros.org/Industrial/Software_Status Zugriff am: 01.01.2020

ROSIN Consortium 2017 ROSIN Consortium, 2017. ROSIN Deliverable D3.1: Quality Assurance Process and Community Management in ROS. *ROSIN Project Deliverables*

ROSIN Consortium 2020
ROSIN Consortium, 2020.
ROSIN Project: ROS-Industrial Quality-Assured Robot Software Components,
URL: https://rosin-project.eu/
Zugriff am: 01.01.2020

Royce 1987
Royce, Winston W., 1987.
Managing the development of large software systems: concepts and techniques.
In:
Proceedings of the 9th international conference on Software Engineering,
S. 328–338

Santos et al. 2018
Santos, André; Cunha, Alcino; Macedo, Nuno, 2018.
Property-based testing for the robot operating system.
In:
Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation,
S. 56–62.
DOI: 10.1145/3278186.3278195

Schantz & Schmidt 2002
Schantz, Richard E; Schmidt, Douglas C, 2002.
Middleware.
Encyclopedia of Software Engineering.
DOI: 10.1002/0471028959.sof205

Schmitt et al. 2017
Schmitt, Carolin; Schäfer, Johannes; Burmester, Michael, 2017.
Wie wirkt der Care-O-bot 4 im Verkaufsraum?
Mensch und Computer 2017-Usability Professionals.
DOI: 10.18420/muc2017-up-0171

Schraft et al. 2001
Schraft, Dieter Rolf; Graf, Birgit; Traub, Andreas; John, Dirk, 2001.
A mobile robot platform for assistance and entertainment.
Industrial Robot: An International Journal **28** (1), S. 29–35.
DOI: 10.1108/01439910110380424

Schwaber & Sutherland 2011
Schwaber, Ken; Sutherland, Jeff, 2011.
The scrum guide.
Scrum Alliance

SERONET Consortium 2020
SERONET Consortium, 2020.
SERONET Projekt,
URL: https://www.seronet-projekt.de/
Zugriff am: 01.01.2020

Seugling & Rölin 2006 Seugling, Axel; Rölin, Martin, 2006. Evaluation of physics engines and implementation of a physics module in a 3d-authoring tool. *Umea University* **2**. ISBN: 978-3-642-33190-9

Shadow Robotics 2020 Shadow Robotics, 2020. *Shadow Robotics Build Tools for ROS*, URL: https://sr-build-tools.readthedocs.io/en/latest/README/ Zugriff am: 01.01.2020

Shakhimardanov et al. 2011 Shakhimardanov, Azamat; Hochgeschwender, Nico; Reckhaus, Michael; Kraetzschmar, Gerhard, 2011. Analysis of software connectors in robotics. In: IEEE (Hrsg.): Conference on Intelligent Robots and Systems. San Francisco, USA, 25.09.2011–30.09.2011, S. 1030–1035. DOI: 10.1109/IROS.2011.6095183

Smart 2011 Smart, John Ferguson, 2011. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. O'Reilly Media, Inc. ISBN: 978-1449305352

Smart 2007 Smart, William, 2007. Writing code in the field: Implications for robot software development. In: *Software Engineering for Experimental Robotics*. Springer, S. 93–105. DOI: 10.1007/978-3-540-68951-5_5

Smits & Bruyninckx 2011 Smits, Ruben; Bruyninckx, Herman, 2011. Composition of complex robot applications via data flow integration. In: IEEE (Hrsg.): International Conference on Robotics and Automation ICRA. Shanghai, China, 09.05.2011–13.05.2011, S. 5576–5580. DOI: 10.1109/ICRA.2011.5979958

Spillner & Linz 2012 Spillner, Andreas; Linz, Tilo, 2012. *Basiswissen Softwaretest*. dpunkt. ISBN: 978-3-86490-583-4

Stamelos et al. 2002 Stamelos, Ioannis; Angelis, Lefteris; Oikonomou, Apostolos; Bleris, Georgios L, 2002. Code quality analysis in open-source software development. *Information systems journal* **12** (1), S. 43–60. DOI: 10.1046/j.1365-2575.2002.00117.x

Staranowicz & Mariottini 2011 Staranowicz, Aaron; Mariottini, Gian Luca, 2011.
A survey and comparison of commercial and open-source robotic simulator software.
In:
Proceedings of the 4th International Conference on Pervasive Technologies Related to Assistive Environments,
S. 56.
DOI: 10.1145/2141622.2141689

Staron 2017 Staron, Miroslaw, 2017.
Automotive Software Architectures: An Introduction.
Springer.
ISBN: 978-3-030-65938-7

Travis CI 2020 Travis CI, 2020.
Travis CI,
URL: https://travis-ci.com/
Zugriff am: 01.01.2020

Tsui et al. 2012 Tsui, Katherine; Desai, Munjal; Yanco, Holly, 2012.
Towards measuring the quality of interaction: communication through telepresence robots.
In:
Proceedings of the workshop on performance metrics for intelligent systems,
S. 101–108.
DOI: 10.1145/2393091.2393112

Valavanis 2008 Valavanis, Kimon P, 2008.
Advances in unmanned aerial vehicles: state of the art and the road to autonomy. Springer Science & Business Media.
ISBN: 978-1-4020-6113-4

Vaughan & Gerkey 2007 Vaughan, Richard; Gerkey, Brian, 2007.
Reusable robot software and the player/stage project.
In: *Software Engineering for Experimental Robotics.*
Springer, S. 267–289.
DOI: 10.1007/978-3-540-68951-5_16

Weißhardt et al. 2014 Weißhardt, Florian; Kett, Jannik;
Araujo, Thiago de Freitas Oliveira; Bubeck, Alexander;
Verl, Alexander, 2014.
Enhancing software portability with a testing and evaluation platform.
In: VDE (Hrsg.):
ISR/Robotik 2014: 41st International Symposium on Robotics.
München, 02.06.2014–03.06.2014.
DOI: 10.24406/publica-fhg-384644

Weißhardt & Koehler 2016 Weißhardt, Florian; Koehler, Florian, 2016.
Automatic Testing Framework for Benchmarking Applications.
In: VDE (Hrsg.):
ISR/Robotik 2016: 47st International Symposium on Robotics.
München, 21.06.2016–22.06.2016.
ISBN: 978-3-8007-4231-8

Weißhardt 2021 Weißhardt, Florian, 2021.
ATF Repository,
URL: https://github.com/floweisshardt/atf
Zugriff am: 01.12.2021

Wisspeintner et al. 2009 Wisspeintner, Thomas; Van Der Zant, Tijn; Iocchi, Luca; Schiffer, Stefan, 2009.
RoboCup@Home: Scientific competition and benchmarking for domestic service robots.
Interaction Studies **10** (3), S. 392–426.
DOI: 10.1075/is.10.3.06wis

Wurman et al. 2008 Wurman, Peter R; D'Andrea, Raffaello; Mountz, Mick, 2008.
Coordinating hundreds of cooperative, autonomous vehicles in warehouses.
AI magazine **29** (1), S. 9–9.
DOI: 10.1609/aimag.v29i1.2082

Yuh 2000 Yuh, Junku, 2000.
Design and control of autonomous underwater robots: A survey.
Autonomous Robots **8** (1), S. 7–24.
DOI: 10.1023/A:1008984701078

Zillich 2012 Zillich, Michael, 2012.
My Robot is Smarter than Your Robot - On the Need for a Total Turing Test for Robots.
Revisiting Turing and his Test: Comprehensiveness, Qualia, and the Real World, S. 12.
URN: http://hdl.handle.net/20.500.12708/72905

Zug et al. 2013 Zug, Sebastian; Poltrock, Thomas; Penzlin, Felix; Walter, Christoph; Hochgeschwender, Nico, 2013.
Analyse und Vergleich von Frameworks für die Implementierung von Robotikanwendungen.
Elektronische Zeitschriftenreihe der Fakultät für Informatik der Otto-von-Guericke-Universität Magdeburg