Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Fachgebiet Neuroinformatik und Kognitive Robotik

Entwurf und Realisierung einer zentralen Ablaufsteuerung für komplexe rekonfigurierbare Anwendungen auf einem Serviceroboter

Masterarbeit zur Erlangung des akademischen Grades Master of Science

Alexander Katzmann

Betreuer: Dipl.-Inf. Steffen Müller,
Dr.-Ing. Christof Schröter
FG Neuroinformatik und Kognitive Robotik

Verantwortlicher Hochschullehrer:
Prof. Dr. H.-M. Groß
FG Neuroinformatik und Kognitive Robotik

Die Masterarbeit wurde am 02.07.2015 bei der Fakultät für Informatik und Automatisierung der Technischen Universität Ilmenau eingereicht.

Danksagung

An dieser Stelle möchte ich gerne meinen Dank an all jene aussprechen, die mich bei der Erstellung dieser Arbeit stets tatkräftig unterstützt haben. Gerade während des Fertigstellungsprozesses gab es doch die ein oder andere größere Hürde, ob nun gerade bei der Anfertigung dieser Arbeit, oder auch völlig unabhängig von ihr. Einige dieser Hürden zu überwinden wäre mir ohne die großartige Unterstützung meiner Freunde und auch meiner Kollegen nicht möglich gewesen. Die Freunde waren es, die sich um alltägliche Aufgaben und Schwierigkeiten kümmerten, ob nun Umzug, Malerarbeiten oder Sonstiges, die mir also zumindest den Teil an Arbeit abgenommen haben, der sonst zu viel gewesen wäre, die zu den richtigen Zeitpunkten ein offenes Ohr und zu den passenden Momenten auch die passenden Worte hatten. Mein besonderer Dank gilt deshalb Herrn Thomas Seyfarth, Frau Stefanie Zierdt, Herrn Paul Wegfraß, Herrn Philipp Götze, Herrn Thomas Schnürer, meinem Bruder Robert sowie meiner Mutter Christine Katzmann.

Meine Kollegen wiederum waren es, die mir immer dann fachlich aushelfen konnten, wenn sich die dringende Notwendigkeit zur säckeweisen Beschaffung neuen Wissens zeigte, wenn sich ein kleiner oder weniger kleiner Formfehler in die Arbeit eingeschlichen hatte, oder schließlich wenn am Roboter etwas zu reparieren war, um die Frist eines Demonstrationstermins auch einhalten zu können. Mein Dank gilt deshalb besonders Herrn Dipl.-Inf. Steffen Müller, Herrn Dr.-Ing. Christof Schröter, Herrn Thomas Schmiedel, MSc., Herrn Dipl.-Inf. Erik Einhorn, Herrn Dipl.-Inf. Ronny Stricker, Herrn Univ.-Prof. Dr.-Ing. Horst-Michael Groß sowie Frau Dr.-Ing. Andrea Scheidig.

Mein allergrößter Dank jedoch, insbesondere für alltägliche Mitarbeit wenn ich da und Vorarbeit wenn ich nicht da war, für Umzugsarbeiten trotz temporär dicken Bauches, für Gegenlesen, fachliche und vor Allem menschliche Unterstützung, für alles was war und alles was noch kommt, gilt an dieser Stelle Frau Antonia Scholz.

Erklärung: „Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe. Alle von mir aus anderen Veröffentlichungen übernommenen Passagen sind als solche gekennzeichnet.“

Ilmenau, 02.07.2015 . . . . . .

Alexander Katzmann

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 2

1.2 Zielstellung und Abgrenzung 3

1.3 Vorgehensweise und Lösungsansatz 4

2 Grundlagen und Theorie **7**

2.1 Begriffsklärung 7

2.2 Präemptives Scheduling 8

3 State of the Art **11**

3.1 Gedächtnislose und gedächtnisbehaftete Systeme 11

3.2 Speziallösungen 13

3.2.1 Ad-hoc-Methode 14

3.2.2 Ablaufsteuerung über State-Machines 14

3.3 Kombinierte und hybride Systeme 17

3.3.1 Multi-State-Machines 17

3.3.2 Hierarchische State-Machines 19

3.3.3 Stack-basierte Systeme 22

3.4 Adaptive und lernende Systeme 25

3.5 Robotics in Concert (ROS - Rocon) 29

3.5.1 Zielstellung und Herangehensweise 29

3.5.2 Gemeinsamkeiten und Abgrenzung 32

3.6 Zwischenfazit 32

4 Dynamische rekonfigurierbare Ablaufsteuerung **35**

4.1 Grundbegriffe und Konzepte 35

4.1.1 Aufbau und Informationsfluss 37

4.1.2 „Gebundene Kooperativität" als Grundgedanke 38

4.1.3 Abgrenzung zu bisherigen Ansätzen 39

4.2 Komponentenaufbau 41

4.2.1 Intuitive Definition 41

4.2.2 Services vs. Ressourcen 44

4.2.3 Hierarchische Service-Definitionen 46

4.3 Konfigurierbarkeit und Integration 52

4.4 Zusätzliche Problemstellung durch Dynamik 54

4.4.1 Verklemmungen und Verklemmungsvermeidung 54

4.4.2 Prioritätsinversion 59

4.5 Verwendete Heuristiken 61

4.5.1 Earliest Deadline First 65

4.5.2 Prioritätengesteuertes Scheduling 68

4.5.3 Hybrides Scheduling 69

4.6 Sicherheit und sicherheitskritische Anwendungen 77

4.6.1 Zwangsentzug von Ressourcen 79

4.6.2 Unerfüllbarkeit von Schedules 80

5 Test und Validierung **85**

5.1 Bewertungsmaßstab 85

5.2 Funktionale Tests 87

5.2.1 Continious Integration 87

5.2.2 Stress-Test 88

5.3 Anwendungstest 91

5.3.1 Testszenario 91

5.3.2 Systemverhalten 94

6 Zusammenfassung und Fazit **97**

6.1 Erreichung der Zielstellung 98

6.2 Weitere Ansätze und Optimierungsmöglichkeiten 99

A Ergänzende Unterlagen **103**

Literaturverzeichnis **110**

Kapitel 1

Einleitung

Insbesondere innerhalb der letzten zwei Jahrzehnte wurden große Fortschritte in den Bereichen interaktive Robotik und Service-Robotik verzeichnet. Dies spiegelt sich insbesondere in der Anzahl veröffentlichter Publikationen und Zeitschriften, veranstalteter Konferenzen und „Robotics Challenges" sowie in der Aufarbeitung der Teilbereiche robotischer Systeme (bspw. *vision-based feature extraction*) wider. Zum einen liegt dies darin begründet, dass Rechenleistung bisher einem exponentiellen Wachstum zu unterliegen scheint (vgl. *Moore's Law* [Moore, 1965]) und damit vorher wenig effiziente Modelle (bspw. *Hidden Markov Models*) für praktische Anwendungsszenarien eingesetzt werden können. Ein anderer Grund liegt darin, dass seit Überschreiten der neuronalen Eiszeit in der Mitte der 1970er Jahre neuartige Modelle (bspw. *Active Appearance Models* [Cootes et al., 1998], Multi-Cue-Ansätze [Boehme et al., 1998], *Deep-Belief-Networks* [Hinton et al., 2006], ...) entwickelt und für die Robotik nutzbar gemacht wurden. Durch beide Faktoren wurde der Einsatz derartiger Verfahren vor allem ökonomischer, die Forschungsbestrebungen entsprechend größer. Weiterhin existieren mittlerweile einige marktreife Lösungen für mehr oder weniger einfache Aufgaben, beispielsweise Rasenmäh-Roboter [Oh et al., 1988].

Im Rahmen komplexer, multimodaler Mensch-Maschine-Interaktion müssen gleich mehrere Aufgaben simultan bearbeitet werden. Hierzu zählen zum Beispiel: die Detektion von Personen, die Schätzung des Interaktionsinteresses, die visuelle Verfolgung von Interaktionspartnern (Tracking), die Kartierung der Umgebung, die Navigation

innerhalb der so erstellten Karten, etc. Diese werden hier als robotische Basisleistungen bezeichnet. Möchte man noch komplexere Verhaltensmuster wie das Lotsen oder Verfolgen einer Person umsetzen, so erfordert dies bereits die gleichzeitige Nutzung mehrerer dieser Basisleistungen. Allein für eine sozial akzeptable Navigation mit Beachtung der sozialen Distanz (siehe auch [Hall, 1966], [Weinrich, 2015]) ist gleichzeitig folgendes nötig:

- das Erkennen von Personen,
- daraus das Bilden und schließlich das Vereinen von verschiedenen Personenhypothesen,
- die Berechnung von aktuellen und prädizierten Belegungskarten, sowie
- die eigentliche Navigationsleistung.

Diese bereits komplexen Abläufe werden hier als Behaviors, also als robotische Verhaltensweisen bezeichnet.

1.1 Motivation

Plant man im Rahmen der Entwicklung eines noch komplexeren Systems, beispielsweise als Realisierung eines Ambient-Assisted-Living (**AAL**), oder gar eines Personal Robots die Nutzung des Roboters als Sekretär, Lotse, Terminkalender, o. Ä., so teilen sich diese verschiedenen Anwendungsszenarien ein Gros der zugrundeliegenden Verhaltensweisen. Sie nutzen diese also gemeinschaftlich. Eine solche multifunktionale Umsetzung findet sich beispielsweise in den Projekten ROREAS und SERROGA des Fachgebietes Neuroinformatik und Kognitive Robotik der Technischen Universität Ilmenau [Gross et al., 2014], [Gross et al., 2015]. Konsequenterweise sollten auch solche komplexen Abläufe in einem größeren Kontext untergebracht werden können. Wenn nun aber zwei verschiedene Anwendungsszenarien gleichzeitig aktiv sind (bspw. Lotse und Sekretär) und dieselben oder verschiedene Verhaltensweisen nutzen möchten, entstehen Konflikte auf den unter ihnen liegenden Ebenen der Softwarearchitektur

(Basisleistungen/Verhaltensweisen) (eine genauere Darstellung derartiger Konflikte erfolgt in Kap. 4.2.3). Damit diese aufgelöst werden können, bedarf es einer zentralen Schaltstelle, die für die jeweiligen Anwendungszenarien festlegt, wann auf welche Basisfunktionalität und wann auf welches Verhaltensmuster zurückgegriffen werden darf. Diese Schaltstelle bezeichnen wir hier als **zentrale Ablaufsteuerung**.
Da dieses Problem mehr oder weniger präsent in allen Robotikapplikationen auftaucht, gibt es bereits verschiedene Lösungsansätze. Diese basieren mehrheitlich auf Spezialllösungen (siehe Kap. 3.2), z.B. auf Ad-Hoc-Ansätzen oder Zustandsautomaten. Beide bieten den großen Vorteil, ideal an die jeweilige Problemstellung angepasst zu sein. Hinzu kommt eine besonders gute, intuitive Nachvollziehbarkeit, Validierbarkeit und Robustheit. Demgegenüber stehen jedoch dieselben Nachteile, die sich allgemein im Vergleich von Individual- und Standardsoftware abzeichnen:

- der hohe Anpassungsgrad sorgt für eine geringere Portierbarkeit
- der Wiederverwendungsgrad bei Beginn neuer Projekte ist verhältnismäßig klein
- für wiederverwendete Komponenten entsteht jeweils ein hoher Aufwand für Integration und Konfiguration

Die eigentlich relevante Entwicklung wird somit verzögert. Wünschenswert wäre also ein Konzept, welches den Aufbau wiederverwendbarer Komponenten für oben genannte Zwecke gestattet, welche ohne großen Integrations- oder Konfigurationsaufwand innerhalb verschiedener Projekte und für verschiedene Einsatzzwecke benutzt werden können. Hierdurch würden Entwicklungskosten reduziert und somit eine weitere Konzentration der Arbeitsleistung auf robotische Anwendungsgebiete ermöglicht.

1.2 Zielstellung und Abgrenzung

Im Rahmen dieser Arbeit soll nun ein generisches Konzept zur Realisierung einer solchen zentralen Ablaufsteuerung entwickelt werden. Es soll sowohl eine geteilte, als auch eine exklusive Nutzung der bereits unter Kapitel 1 genannten Roboterfähigkeiten erfolgen. Die Anpassung oder gar Reimplementierung dieser Komponenten jedoch bedeutet

einen sehr hohen Aufwand. Es bedarf somit eines Ressourcen-Frameworks, welches als Wrapper für die bereits existierenden Basisleistungen und Verhaltensweisen fungiert, statt diese vollständig an ein vollkommen neues Konzept anzupassen. Auf Basis dieses Ressourcen-Frameworks erfolgt die eigentliche Implementierung der Ablaufsteuerung (im Folgenden **Scheduler** genannt). Aus den unter Kapitel 1.1 genannten Gründen ergeben sich für die Implementierung folgende softwaretechnische Ziele:

- Nebenläufigkeit
- Modularität
- Wiederverwendbarkeit
- Nachvollziehbarkeit

Ebenfalls soll eine direkte Integration in die Robotik-Middleware MIRA erfolgen. Die Aufgabe der Implementierung einer zentralen Ressourcenverwaltung mit verteilten Ressourcen ist dabei ein ausgesprochen komplexes Problem. In Kapitel 6.2 wird deshalb noch einmal genauer auf weiteres Optimierungspotential eingegangen. Eine Portierung zu anderen Robotik-Middlewares (bspw. ROS) ist derzeit nicht geplant.

1.3 Vorgehensweise und Lösungsansatz

Nach der Klärung einiger Grundbegriffe in Kapitel 2 werden in Kapitel 3 zunächst bereits existierende Konzepte unter die Lupe genommen und untereinander verglichen. Die Ergebnisse hieraus sollen die Basis für einen eigenen, dynamischen Ansatz darstellen, welcher in Kapitel 4 vorgestellt wird. Nach der Implementierung soll ein Vergleich mit bisherigen Konzepten erstellt und die Software gegebenenfalls weiter optimiert werden. Da es sich um ein weitestgehend unerforschtes Themengebiet handelt, stellt es eine schwierige Aufgabe dar, allgemeingültige Validierungsmaßstäbe für eine dynamische Ablaufsteuerung festzulegen. Es existieren nur sehr wenige Arbeiten mit Bezug zu dieser Thematik. Hinzu kommt, dass die Kriterien für eine „gelungene Ablaufsteuerung" in hohem Maße von der jeweiligen Problemstellung abhängen. Dennoch

wird durch theoretische Überlegungen und die praktischen Erfahrungen, die im Laufe der Anwendungstests aus Kapitel 5 gesammelt wurden, versucht, einen angemessenen Vergleichsmaßstab zu finden.

Kapitel 2

Grundlagen und Theorie

Zur Schaffung eines gemeinsamen Verständnisses bietet es sich an, wiederkehrende, elementare Begriffe bereits früh zu erläutern. Einige von ihnen sind notwendig, um den State-of-the-Art hinreichend genau zu schildern. Eine erste Definition verwendeter Bezeichnungen und grundsätzlicher Abläufe erfolgt deshalb bereits an dieser Stelle.

2.1 Begriffsklärung

Der Begriff *Scheduler* wurde in Analogie zur den Zugriff auf den Prozessor steuernden Einheit eines Betriebssystems gewählt. Er ist jedoch nur insofern zutreffend, als die Steuereinheit des hier vorgestellten Konzeptes ebenfalls die Koordination von Zugriffen auf Systemkomponenten vornimmt. Dies geschieht durch Errechnung eines so genannten *Ressourcenplans* (auch: *Schedule*). Ein wesentlicher Unterschied zum Betriebssystem-Scheduler besteht jedoch darin, dass dieser die Zuteilung von Prozessorrechenzeit, und eben nicht von anderen Systemressourcen, vornimmt. Er muss dabei zwar ebenfalls die Abhängigkeiten zwischen den einen Zugriff anfordernden Prozessen kennen, nimmt die Zuteilung aber im Wesentlichen zeitgesteuert vor. In der Robotik jedoch ist die Zuteilung auf Basis konkreter Zeitangaben deutlich weniger hilfreich als im Falle eines Betriebssystems. Stattdessen erfolgen Zuteilungen hier vorwiegend Event-gesteuert. Plant man, die verschiedenen Module (genannt: *Applets*) derart zu koordinieren, dass sie ihre eigentliche Funktion je nach zur Verfügung ste-

henden Systemressourcen ausführen, so ist es im Sinne eines modular erweiterbaren Roboters ebenfalls nicht relevant, ob früher oder später alle Prozesse (*Applets*) Rechenzeit zugeteilt bekommen. Auch dies stellt einen wesentlichen Unterschied zum Betriebssystem-Scheduler dar, der insbesondere darauf achten muss, dass kein Prozess chronisch verdrängt wird und somit aushungert. Der hier vorgestellte Scheduler sowie alle von ihm verwalteten Services (entsprechen den Systemressourcen des Betriebssystem-Schedulers) und Applets unterliegen tatsächlich selbst dem Scheduling des Betriebssystems, bekommen also Prozessorrechenzeit zugewiesen. Das Gleiche gilt für das *MIRA-Framework*, auf welchem das hier gezeigte Konzept aufbaut. Da die Zuweisung von Zeitschlitzen auf dem Systemprozessor nur schwer prädizierbar ist, sollten kritische, zeitgesteuerte Prozesse nach Möglichkeit **nicht** auf Basis des hier vorgestellten Schedulers realisiert werden (hierauf wird in Kap. 4.5.1 genauer eingegangen).
Die jeweils kursiv geschriebenen Begriffe werden im Verlauf dieser Arbeit noch einige Male benutzt werden. Die hier gegebenen vorläufigen Definition sind für die folgenden Kapitel jedoch vorerst ausreichend. Eine genauere Beschreibung erfolgt in Kapitel 4.1.

2.2 Präemptives Scheduling

Scheduling-Algorithmen für Multitasking-fähige Systeme lassen sich allgemein in zwei unterschiedliche Kategorien einteilen: präemptives, also unterbrechendes, und nicht-präemptives Scheduling (vgl. präemptiv: „einer sich abzeichnenden Entwicklung zuvorkommen" [dud, 2015]). Im nicht-präemptiven Fall wird eine Ressource nach der Zuteilung an eine Anwendung durch diese selbstständig wieder freigegeben, sobald sie die Ressource nicht weiter benötigt. In keinem Falle wird sie ihr vorher innerhalb des zugeteilten Zeitschlitzes wieder entzogen. Gebe es also eine solche Ressourcenzuteilung an eine Anwendung und kurz darauf benötigt eine zweite Anwendung ebenfalls Zugriff auf die Ressource, so hätte die zweite Anwendung ausschließlich die Möglichkeit, die erste nach der Freigabe der Ressource zu fragen oder aber auf eine Freigabe zu warten. Die endgültige Entscheidung obliegt allerdings allein der Anwendung, die die Ressource hält, völlig unabhängig von der tatsächlichen (globalen) Dringlichkeit

einer Umverteilung. Eine Unterbrechung des Zugriffes von außen ist ausgeschlossen. Aus diesem Grund bezeichnet man nicht-präemptives Scheduling auch als „kooperatives Scheduling", da ein Zusammenarbeiten zwischen verschiedenen Akteuren für einen reibungslosen Ablauf gewissermaßen unerlässlich ist.
Demgegenüber steht das so genannte präemptive Scheduling. Hierbei erfolgt die eigentliche Ressourcenzuteilung ausschließlich durch eine übergeordnete Instanz „Scheduler". Es ist durchaus möglich, dass auch vor der abschließenden Abarbeitung einer Anwendung deren Ressourcen entzogen und sie selbst unterbrochen wird. Im Gegenzug ist es ihr möglich, beim Scheduler Ressourcen zu erfragen, *Deadlines* (späteste Abarbeitungszeitpunkte) zu setzen oder *Interrupt-Routinen* (abzuarbeitende Methoden bei vorübergehender Unterbrechung des aktuellen Programms) bei Entzug einer Ressource zu definieren. Die endgültige Entscheidungshoheit steht jedoch alleine dem Scheduler zu. Zu beachten ist, dass die Existenz eines Schedulers keinerlei Aussage über die Präemptivität des verwendeten Scheduling-Verfahrens zulässt. Auch beim kooperativen Scheduling ist es möglich, dass Anwendungen ihre Ressourcenzuteilungen aktiv von einem Scheduler erfragen. Der wesentliche Unterschied besteht darin, ob die endgültige Kontrolle über eine Systemressource an den benutzenden Prozess abgegeben wird, oder aber, ob sie beim Scheduler verbleibt.
In Bezug auf Betriebssysteme stellt der Begriff „präemptibler Kernel" insofern eine Mischform dar, als auch während der Abarbeitung nicht-atomarer Betriebssystemfunktionen, also auch Scheduling-Prozessen und Hardware-Zugriffen, eine Unterbrechung zugunsten eines Benutzerthreads oder einer Anwendungsaktion stattfinden kann [Eykholt et al., 1992]. Dies wird hier jedoch nicht weiter betrachtet.

Kapitel 3

State of the Art

In Kapitel 1.3 wurde bereits darauf eingegangen, dass aufgrund der Dominanz nicht-generischer Eigenentwicklungen derzeit nur wenige Ansätze zu allgemeinen Herangehensweisen oder Methodiken existieren. Bei diesen wird sich dann aber häufig abstrakter Konzepte der Automatentheorie bedient. Es existieren jedoch auch Ansätze zu adaptiv-lernenden (siehe Kap. 3.4) sowie dynamischen Konzepten (siehe Kap. 3.5).

3.1 Gedächtnislose und gedächtnisbehaftete Systeme

Im Folgenden ist ebenfalls eine Unterscheidung in gedächtnislose und gedächtnisbehaftete Systeme möglich. Ein Gedächtnis wird im Falle einer Ablaufsteuerung immer in Szenarien benötigt, in denen gilt:

- durch die Abarbeitung einer Aktion wird eine andere Aktion ausgelöst,
- zum Fortsetzen der ersten Aktion muss die zweite zunächst bearbeitet werden,
- nach der Unterbrechung soll die ursprüngliche Aktion wieder an der Stelle fortgesetzt werden, an der unterbrochen wurde.

Ein System wird dann als gedächtnislos bezeichnet, wenn über die modellierten Zustände keine verdeckten Variablen existieren, die eine Auswirkung auf die Abläufe des Systems haben (analog: gedächtnisbehaftet). Formell stellt ein Gedächtnis der Länge

h zum Zeitpunkt t die Menge zuvor durchlaufener Zustände $|Z_m| = Z_{t-1}, ..., Z_{t-h}$ dar. Grundsätzlich lässt sich jede endliche Menge durchlaufener Zustände auch durch die Einführung einer neuen Zustandsmenge Z' abbilden (z. B. bei Zustandsautomaten, siehe Kap. 3.2.2). Dies erfolgt durch Kreuzproduktbildung auf Basis der ursprünglichen Zustandsmenge. Die sonst verdeckten Variablen werden so durch Zustände realisiert. Die Zustandsmenge wächst hierdurch jedoch exponentiell (siehe Kap. 3.1):

$$Z' = Z \underbrace{\times ... \times Z}_{h \text{ mal}} \tag{3.1}$$

Besitzt ein gedächtnisloses System eine Menge von $|Z| = n$ Zuständen und muss nun der jeweils letzte durchlaufene Zustand bekannt sein, so erzeugt dies bereits eine neue Zustandsmenge mit $|Z \times Z| = n^2$ Zuständen. Ist der vorletzte Zustand ebenfalls relevant, ergibt dies bereits $|Z \times Z \times Z| = n^3$ Zustände, usw.
Dies stellt unter anderem ein wesentliches Problem bei der Benutzung von State-Machines zur Ablaufsteuerung dar (genauer beschrieben in Kap. 3.2.2), ist jedoch auch darüber hinaus relevant, da ein Gedächtnis grundsätzlich Auswirkungen auf die Validierbarkeit und auch die Modifizierbarkeit des Systems hat. Bei Ad-hoc-Lösungen (siehe Kap. 3.2.1) beispielsweise stellt sich dieses Problem durch die Einführung zusätzlicher Zustandsvariablen, bei Multi-State-Machines (siehe Kap. 3.3.1) durch die Einführung neuer Superstates, etc. Hierauf wird in den entsprechenden Unterkapiteln noch einmal genauer eingegangen.
Zu beachten ist, dass das exponentielle Wachstum der Zustandsmenge ausschließlich dann auftritt, wenn stets das gesamte Gedächtnis benötigt wird. In der Praxis reicht es jedoch häufig, wenn wenige Zustandsteilmengen ein Gedächtnis besitzen. Eine andere Herangehensweise ist die Modellierung des Gedächtnisses als Stack. Hierauf wird in Kapitel 3.3.3 noch einmal genauer eingegangen. Im Folgenden gehen wir davon aus, dass alle Systeme ein entweder implizit oder explizit modelliertes Gedächtnis aufweisen.

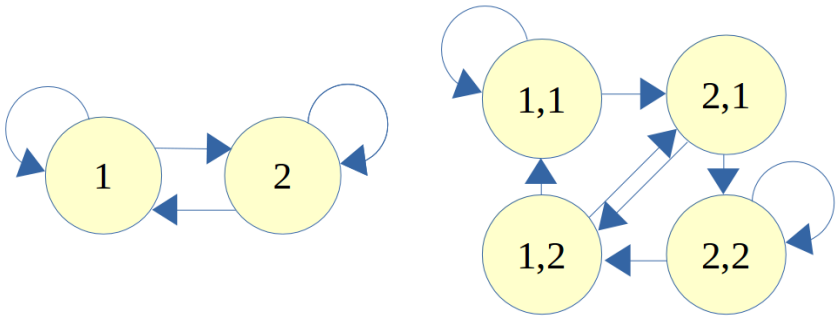


Abbildung 3.1: Anwachsen der Zustandsmenge durch Gedächtnis
Das nachträgliche Modellieren eines endlichen Gedächtnisses erhöht die Mächtigkeit der Zustandsmenge exponentiell mit der Größe des Gedächtnisses. Links: Zustandsgraph ohne Gedächtnis, Rechts: Gedächtnis der Größe eins mit Zuständen z=(aktueller Zustand, vorheriger Zustand)

.

3.2 Speziallösungen

Wie bereits in Kapitel 3 genannt, existieren auf dem Markt viele Speziallösungen, welche genau einem gewissen Zweck dienen und bei denen sich jeder Ablauf in einer Programmstruktur oder einem Zustand wiederfindet. Speziallösungen sind deshalb selten portierbar. Sie weisen üblicherweise eine sehr hohe innere Kopplung sowie einen hohen Grad an Spezifik auf. Häufig finden sich auch Speziallösungen ohne konkrete abstrakte Modellierung von Zuständen (s.g. *Ad-hoc-Lösungen*; siehe Kap. 3.2.1). Im Gegensatz zu abstrakten Modellen stellen hier interne Variablen statt formaler Modelle die Grundlage der Abläufe dar. State-Machine-Ansätze zählen ebenfalls zum Bereich der Speziallösungen (siehe Kap. 3.2.2), da sich alle für die Ablaufsteuerung relevanten Elemente jeweils in Zuständen der State-Machine wiederfinden. Da Speziallösungen jedoch auch eine Reihe von weiteren Problemen aufweisen, werden ab Kapitel 3.3 generischere Konzepte vorgestellt.

3.2.1 Ad-hoc-Methode

Generell lässt sich für **jedes** mithilfe eines Computers berechenbares Anwendungsszenario eine einfache Ablaufsteuerung realisieren („Structured Program Theory", [Böhm und Jacopini, 1966]). Die jeweilige softwaretechnische Umsetzung sieht hierfür meist eine oder mehrere Komponenten vor, welche basierend auf ihren jeweiligen inneren Zuständen im üblicherweise immer gleichen Hauptfunktionsaufruf jeweils unterschiedliche Unterprogramme oder Programmzweige aufrufen, welche die jeweilige Unterfunktionalität realisieren. Da derartige Umsetzungen sich mit dem informatischen Begriff „Programm" decken und ihre Ablaufsteuerungsmechanismen sogar eine direkte Verwirklichung in Programmiersprachenanweisungen gefunden haben (*for*-Schleifen, *switch-case*-Anweisungen, etc.; auch als „Control Flow Structures" bezeichnet), wird auf ihren Aufbau hier nicht weiter eingegangen. Ad-hoc-Ansätze stehen bezüglich der direkt realisierten Mächtigkeit zwar auf sehr niedriger Stufe, werden aufgrund ihres sehr niedrigen Inititialaufwandes jedoch trotz der dadurch in späteren Entwicklungsstadien resultierenden Mehraufwand bei Modifikation und überhandnehmender Komplexität ausgesprochen häufig auch im makroskopischen Bereich, also auch für nichttriviale Programmteile, verwendet. Nichtsdestotrotz lassen auch sie sich durchaus in formale Modelle überführen, beispielsweise durch Kreuzproduktbildung der jeweiligen Variablenbelegungsmengen, sie unterscheiden sich also funktional - und des Öfteren auch implementierungstechnisch - nur marginal von anderen Implementierungen. Dennoch verspricht eine Transformation in abstraktere Modelle Vorteile bei der späteren Handhabung. Bezogen auf die Ablaufsteuerung in der Robotik finden Ad-hoc-Ansätze vor allem im Hobbybereich, bei der Programmierung einfacher Mikrocontroller oder bei wenig umfangreichen Projekten Anwendung. In den folgenden Kapitel finden sich deutlich abstraktere Ansätze, welche versuchen die Übersichtlichkeit auch für größer dimensionierte Anwendungen wie die der Service-Robotik zu gewährleisten.

3.2.2 Ablaufsteuerung über State-Machines

Ein *Zustandsautomat* (*endlicher Automat, engl.* ***Finite-State-Machine*** *oder kurz* ***FSM***) ist ein einfacher gerichteter Graph, wobei *Knoten* den aktuellen Zustand und

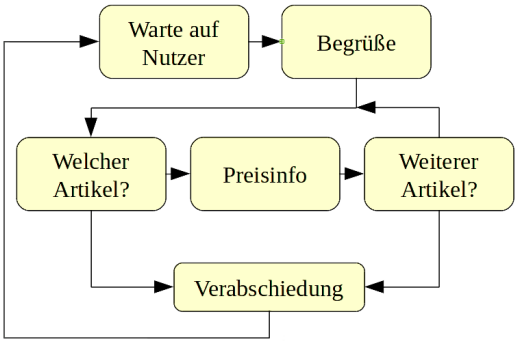


Abbildung 3.2: Finite-State-Machine zur Ansteuerung eines Roboters
Dies ist ein einfaches Beispiel für die grundsätzliche Herangehensweise bei der Umsetzung einer Ablaufsteuerung als State-Machine. Zu sehen ist ein Roboter zur Preisauskunft, beispielsweise in einem Baumarkt (Transitionsbedingungen fehlen).

Kanten die Transitionen zwischen den Zuständen angeben. Ebenfalls existieren *Kantenmarkierungen*, welche das auslösende Event (und ggf. Transitionsbedingungen) angeben [Harel, 1984]. Zustandsautomaten sind eine einfache Form der Ablaufsteuerung, sowohl im Allgemeinen für Ablaufketten in allen Bereichen der EDV, als auch im Speziellen für grafische Benutzerschnittstellen. In der Industrie wird üblicherweise auf den *GRAFCET*-Standard nach [DIN EN 60848:2014-12, 2014] (ehemals DIN 40719-6) zurückgegriffen, welcher sich (unter Gestattung von Transitions-, Zustandseintritts- und Zustandsaustrittsfunktionen) ebenfalls eindeutig auf Zustandsautomaten reduzieren lässt[1]. Insofern stellt der Zustandsautomat die Norm für einfache Ablaufsteuerungen dar. Hinzu kommt, dass Zustandsautomaten eine Reihe von Vorteilen in der Entwicklung bieten, da ... :

- ... sie sich einfach konstruieren lassen,
- ... sie intuitiv nachvollziehbar sind,
- ... sie automatisch validierbar sind,

[1] Da für praktische Anwendungen generell eine Interpretation des Zustandsautomaten nötig ist, wird im Folgenden davon ausgegangen, dass für Zustandseintritt, -austritt und -übergang jeweils eine solche interpretierende Funktion existiert.

- ... es eine Reihe von vollautomatischen Verifikationstools für verschiedenste Zwecke, einschl. grafischer Benutzerschnittstellen, sowie Entwicklungstools gibt.

Trotz alledem haben Zustandsautomaten auch eine Reihe von Nachteilen, welche insbesondere bei größeren Projekten die Entwicklung verzögern können. So lassen sich Zustandsautomaten zwar lokal durch Hinzufügen neuer Zustände und Kanten ohne größeren Aufwand abwandeln und erweitern, die Erweiterung um globale Zustände, welche also von jedem anderen Zustand (oder zumindest einer größeren Teilmenge), erreichbar sein sollen, stellt sich jedoch als deutlich problematischer dar.
Existieren in einem Zustandsautomaten $|Z| = n$ Zustände mit $|E| = m$ Kanten und ist die Einführung eines zusätzlichen Fehlerzustandes z_E geplant, welcher von allen Zuständen $z \in Z$ erreicht werden können soll, so ist die Einführung von $|(z, z_E)|_{z \in Z} = |Z| = n$ neuen Kanten nötig. Ist nach Behandlung des Fehlerzustandes zusätzlich die Rückkehr zum Herkunftszustand nötig, erfordert dies sogar das Einführen von $|z_{i_E}|_{i \in \{1,\ldots,n\}} = |Z| = n$ neuen Zuständen und $|(z_i, z_{i_E})| + |(z_{i_E}, z_i)| = 2n$ neuen Kanten (siehe Kap. 3.1). Hinzu kommt, dass sich die grafische Visualisierung der entstehenden Zustandsautomaten mit zunehmender Größe schwierig gestaltet, wodurch der ursprüngliche Vorteil, intuitiv verständlich zu sein, immer weiter in den Hintergrund rückt. Ferner müssen neue Module zur Ablaufsteuerung unmittelbar in die State-Machine eingebunden werden. Dies erzeugt eine zentrale Konzentration von Wissen und damit eine hohe innere Kopplung. Sollte also ein hochmodulares, sich kontinuierlich entwickelndes Gesamtsystem gefordert sein, stellt dies einen bedeutenden Nachteil und einen immensen Mehraufwand dar. Gleiches gilt für die Integration in vorhandene oder auch neue Systeme (siehe Abb. 3.3).

Unter der Hinzunahme komplexerer Methodiken werden diese Nachteile teils drastisch reduziert (auch zulasten von Validier- und Verifizierbarkeit), allerdings nicht restlos eliminiert (siehe folgende Kapitel, insbesondere Kap. 3.3.3).

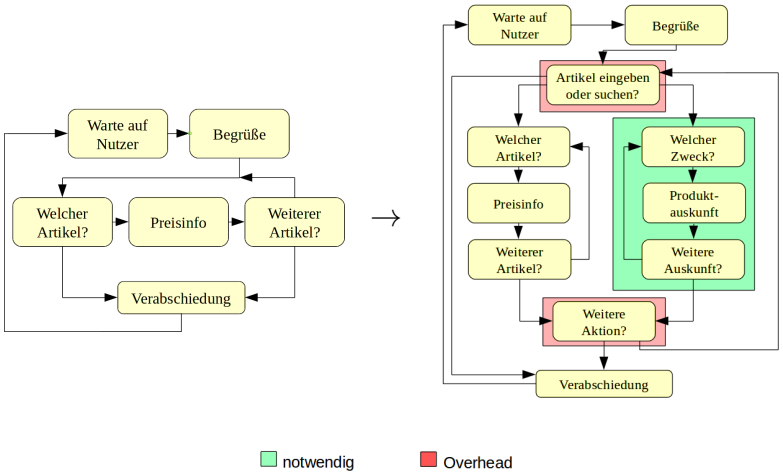


Abbildung 3.3: Darstellung der Probleme bei der Erweiterung einer Ablaufsteuerung auf Basis von State-Machines.
Zu sehen ist die Erweiterung eines einfachen Roboters zur Preisauskunft um eine zusätzliche Funktion zur Produktübersicht. Hervorgehoben ist der dabei entstehende Overhead.

.

3.3 Kombinierte und hybride Systeme

Reine Zustandsautomaten besitzen eine Reihe von Nachteilen (u.A.: Gedächtnislosigkeit, Komplexität, Kopplung; siehe Kap. 3.2.2). Um diesen zu begegnen, wurden unterschiedliche Lösungsansätze entwickelt. Hierzu zählen vor allem die Komposition einer großen State-Machine durch Aufteilung in mehrere State-Machine-Komponenten, die hierarchische Gliederung von State-Machines, sowie die Nutzung mehrerer, miteinander interagierender State-Machines. Die Konzepte sind keineswegs divergent, sondern lassen sich durchaus miteinander kombinieren.

3.3.1 Multi-State-Machines

Eine Variante, die Komplexität der entstehenden State-Machine zu reduzieren, ist die Zerteilung in mehrere, zunächst voneinander unabhängige Zustandsmaschinen. Jede ausgegliederte Zustandsmaschine repräsentiert jeweils eine funktionale Einheit des an-

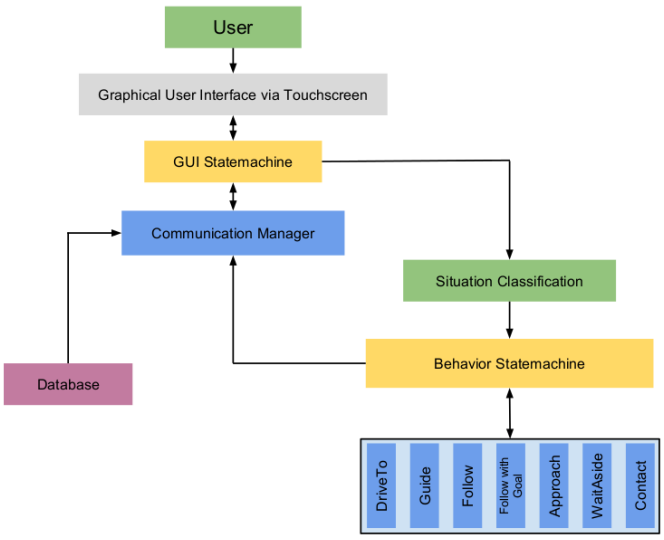


Abbildung 3.4: Funktionale Struktur der Anwendungsschicht (ROREAS)
*Eine Trennung der Ablaufsteuerung von Benutzeroberfläche und Roboterverhalten führt zu deutlich geringerem Wartungs- sowie Anpassungsaufwand und steigert gleichzeitig die Flexibilität (vgl. [*Gross *et al., 2014]).*

zusteuernden Roboters. So wurde beispielsweise beim Projekt ROREAS des Fachgebietes Neuroinformatik und Kognitive Robotik der TU Ilmenau das physische Verhalten des Roboters vollständig von komplexeren Abläufen sowie der grafischen Benutzeroberfläche getrennt (siehe Abb. 3.4).
Ein wesentlicher Vorteil dieser Art der Modellierung ist eine direkte Zustandsrepräsentation jeweils aller Komponenten, die durch eine eigene State-Machine modelliert sind. Dies erzeugt eine gewisse Nähe zur objektorientierten Programmierung. Der Zustand einer Komponente lässt sich somit also vom Modell „ablesen". Die Implementierung der eigentlichen Funktionalität gestaltet sich entsprechend einfacher, da jede der entstehenden State-Machines für sich genommen deutlich intuitiver und übersichtlicher ist. Ebenfalls vereinfacht dies Wartung, Modifikation und Erweiterung. Formal lässt sich ein solches Modell (innere, nicht modellierte Zustände ausgeschlossen) auf ein binäres Petri-Netz reduzieren, was die Nutzung einer Reihe von Algorithmen zur statischen und dynamischen Analyse gestattet. Jeder aller n Zustände der ursprünglichen State-Machines wird hierbei durch einen Platz im Petri-Netz repräsentiert, Kanten

und Interaktionen zwischen den Modellen werden durch Transitionen dargestellt. Der Zustand des Petri-Netzes mit Plätzen $Z = \{z_1, ..., z_n\}$ zum Zeitpunkt t ist nun durch seine aktuelle Markierung $M_t = (M_t(z_1), ..., M_t(z_n))$ gegeben, wobei $M_t(z) : Z \to \mathcal{N}$ die Anzahl der Token im Platz z angibt. Für ein binäres Petri-Netz gilt nun:

$$\forall t \forall z \in Z : M_t(z) \leq 1 \tag{3.2}$$

Ein Platz (ehemals Zustand) ist also entweder aktiv oder inaktiv. Entspricht $Z_a \subseteq Z$ darüber hinaus allen Zuständen der ursprünglichen State-Machine a gilt weiterhin:

$$\forall a \forall t : \sum_{z_i \in Z_a} M_t(z_i) = 1 \tag{3.3}$$

Jeder Teilautomat hat also genau einen aktiven Zustand. Ist ein Implementierender also geübt in der Modellierung von Petri-Netzen, so beschleunigt dies auch die Wartung, Modifikation und Erweiterung von Multi-State-Machines, da die zugrundeliegenden Vorgehensweisen die gleichen sind. Relevante Arbeiten zur automatischen Validierung von Petri-Netzen stammen unter anderem von Jensen und Kristensen (z. B. [Jensen et al., 2007]).

3.3.2 Hierarchische State-Machines

Eine weitere Möglichkeit ist die Umsetzung der Ablaufsteuerung als *hierarchische State-Machine*, bzw. genauer als *State-Chart* (deutsch: Zustands(übergangs)diagramm) nach [Harel, 1984]. State-Charts stellen eine andere Visualisierungsform von Zustandsautomaten mit zusätzlichen Modellierungselementen dar, sind bezüglich ihrer Mächtigkeit mit State-Machines nach der Definition aus Kapitel 3.2.2 jedoch äquivalent. Ursprünglich 1987 von David Harel eingeführt, haben Zustandsübergangsdiagramme eine weitere Normierung durch die Aufnahme in den UML-Standard (als *Zustandsdiagramm*) erfahren [Rumbaugh et al., 2004]. Sie bieten sich somit als weit verbreiteter Standard für die Modellierung von Abläufen an. Für ihre softwaretechnische Umsetzung hat das W3-Konsortium den so genannten **SCXML**-Standard eingeführt [W3C, 2005].

State-Charts bieten eine Reihe von direkten Repräsentation für Funktionen (siehe Abb.

3.5), welche sich in State-Machines nur durch zusätzliche Zustände, Zustandseintritts-, Zustandsaustritts- oder Zustandsübergangsfunktionen realisieren lassen:

- Zustandshierarchien - State-Charts bieten die Möglichkeit, so genannte Superstates einzuführen, welche aus mehreren Zustandsautomaten komponiert werden und selbst jeweils wieder einen vollständigen Zustandsautomaten darstellen
- Zustandssynchronisation - Sowohl das gleichzeitige Eintreten in, als auch das gleichzeitige Austreten aus mehreren Unterzustandsautomaten ist möglich. Zur nebenläufigen Abarbeitung stehen AND- und OR-Semantik (synchrones/abwechselndes Schalten) zur Verfügung.
- History-Konnektoren - speichern beim Austritt aus einem Superstate den aktuellen inneren Zustand und erlauben eine Rückkehr zu diesem bei Wiedereintritt
- Condition-Konnektoren - wählen bei Eintritt in einen Superstate je nach erfüllter Bedingung den entsprechenden Substate
- Selection-Konnektoren - Sind alle Substates eines Superstates eindeutig benannt, kann bei Eintritt in diesen der zu wählende Substate direkt adressiert werden.

Diese zusätzlichen Komponenten gestatten eine deutlich kompaktere Beschreibung, als dies mit der alleinigen Umsetzung durch eine State-Machine der Fall wäre. Als Erweiterungen wurden weiterhin vorgeschlagen:

- Parametrisierte Zustände - Repräsentation von mehreren Zuständen, welche sich lediglich in einem Parameter unterscheiden
- Überlappende Zustände - hierbei besitzen mehrere Superstates eine gemeinsame Schnittmenge an Unterzuständen
- Temporale Logik - zeitvariantes Verhalten von Zuständen
- Rekursive und probabilistische State-Charts

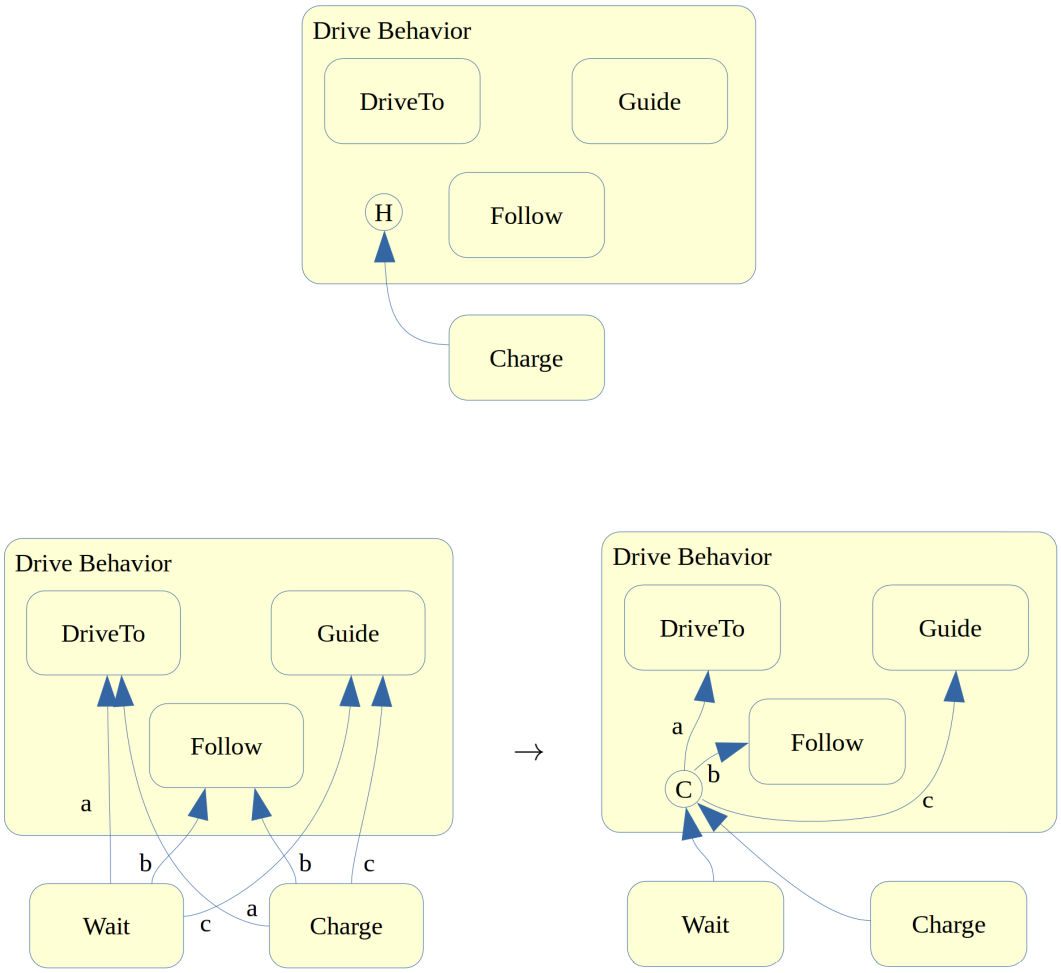


Abbildung 3.5: Darstellung der verschiedenen State-Chart-Konnektoren

Zu sehen sind die in Kap. 3.3.2 ausführlicher diskutierten Konnektoren, die im State-Chart-Konzept zur Vereinfachung von häufig in State-Machines auftauchenden Probleme eingeführt wurden.

oben: *Einsatz eines History-Konnektors zum Wiedereintritt in den letzten Unterzustand nach Austritt aus Superstate „Drive Behavior"*

unten: *Vereinfachung des State-Chart-Aufbaus durch Einsatz eines Condition-Konnektors*

.

Es existieren für diese Funktionen jedoch keine allgemeingültigen Überführungen in State-Machines, so dass sie weder bei der Normierung von State-Charts, noch bei der Aufnahme im SCXML-Standard genannt wurden. Die Modellierung einer Ablaufsteuerung auf Basis eines State-Charts stellt einen wesentlichen Fortschritt zur ausschließlichen Modellierung als State-Machine dar, jedoch lassen sich auch State-Charts nicht beliebig erweitern, ohne an Übersichtlichkeit zu verlieren. Eine Beispielumsetzung auf Basis von SCXML, welches dies verdeutlich, findet sich in Abb. 3.6.
Die darin dargestellten Superstates stellen jedoch durchaus einen großen Mehrwert dar. Durch sie ist es möglich, Funktionen auch bei Eintritt in oder Austritt aus Superstates aufzurufen. Ohne sie müsste für jeden aller adressierbaren Substates ebenfalls eine entsprechende Funktion implementiert werden. Die Einführung von Superstates führt also zu einer deutlichen Reduktion der Redundanz entstehender Abläufe und somit zu geringerem Aufbau-, Modifikations- und Wartungsaufwand. Für mittelgroße Projekte ohne häufigere Modifikation stellen State-Charts ein geradezu ideales Modellierungswerkzeug dar.
Anm.: Die in Abb. 3.5 dargestellten History-Konnektoren entsprechen funktional einem einelementigen Stack. Sie stellen insofern eine Vorstufe zum in Kapitel 3.3.3 vorgestellten Konzept dar.

3.3.3 Stack-basierte Systeme

Die Modellierung eines Gedächtnisses bringt, wie bereits unter Kapitel 3.1 gezeigt, für eine Ablaufsteuerung einige Vorteile mit sich. Aus diesem Grund wurde ebenfalls ein Ansatz zur direkten Modellierung dieses Gedächtnisses auf Basis eines Stacks entwickelt [Mueller et al., 2014a]. Auf ihm basiert das später vorgestellte System aus Kapitel 3.4. Zur Ansteuerung des Roboters wird hierbei, analog zu Abb. 3.4, ebenfalls eine Behavior-State-Machine verwendet (im Gegensatz zur hierarchischen Ablaufsteuerung aus Kap. 3.3.2 ist sie implizit modelliert, funktional jedoch äquivalent).
Das Stack-basierte Konzept gründet sich, ähnlich wie das später vorgestellte, auf der Entwicklung einzelner *Applets* (oder *Apps*) für verschiedene Anwendungsszena-

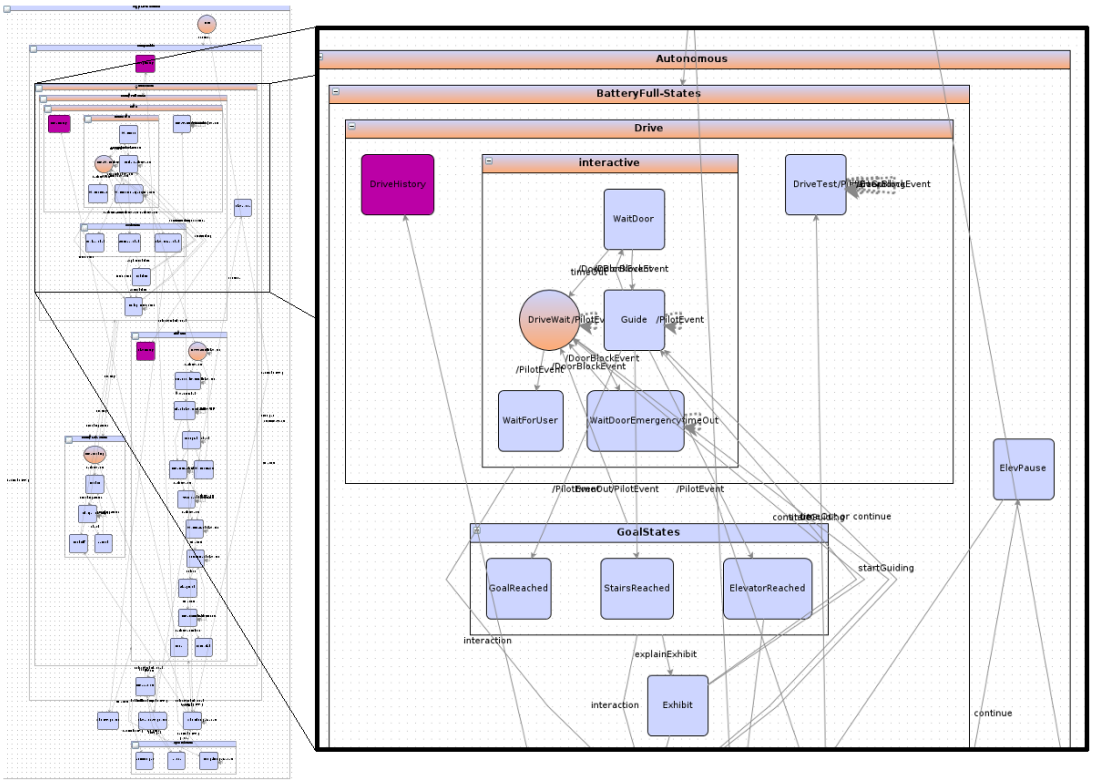


Abbildung 3.6: Ausschnitt Zuse-Guide-SCXML

*Der Zuse-Guide des Fachgebietes Neuroinformatik und Kognitive Robotik realisiert seine Ablaufsteuerung auf Basis von SCXML. Hier zu sehen ist das gesamte State-Chart (links) sowie ein Ausschnitt davon (rechts). Dargestellt sind Substates (Kästchen), Superstates (jeweils mit Kopfzeile) und Zustandsübergänge (Pfeile). Zu sehen ist auch, dass die Übersichtlichkeit der State-Charts mit steigender Komplexität signifikant abnimmt. Die Zustandseintritts, -austritts und -übergangsfunktionen sind in den Zustands- und Transitionseigenschaften modelliert, hier jedoch nicht dargestellt. Genauere Informationen finden sich in [*Stricker *et al., 2012].*

rien. Dabei ist eine strikte Trennung von grafischer Benutzeroberfläche, Funktionalität und robotischem Verhalten vorgesehen. Trotz allem wird die direkte Verwendung von Service-Funktionen innerhalb der GUI-Implementation gestattet. Jede App hält ihre eigene GUI-Definition, was eine direkte Kopplung von Steuerelementen zu App-Funktionen erlaubt. Die Implementation der grafischen Benutzeroberfläche erfolgt auf Basis der Qt-QML-Bibliothek des finnischen Softwareunternehmens *Digia*. Da QML-Oberflächen grundsätzlich zustandsbehaftet sind, behalten Apps - im Gegensatz zur direkten Implementierung der Benutzeroberfläche in einer einzigen GUI-State-Machine - ihren eigenen Zustand stets bei, selbst wenn sie gerade nicht aktiv sind.
Der Zustand der *Application*, also der entstehenden Anwendung inklusive aller Apps, findet sich hingegen in einem Stack wieder. Der Stack dient dabei als Informationsspeicher, das oberste Element des Stacks repräsentiert die derzeit aktive App. Ein Wechsel der aktiven App kann entweder durch einen Nutzer initiiert sein, beispielsweise indem dieser auf einen Menüpunkt drückt, oder aber durch ein Applet selbst ausgelöst werden. Aus Sicht der Application drückt sich dies durch eine Verschiebung innerhalb des Stacks aus. Die neu aktivierte oder aber in den Vordergrund gebrachte (*activate/raise*) App wird auf dem Stack abgelegt, bzw. an oberste Position verschoben. Sollte eine App inaktiv werden, z. B. weil eine andere App ein *raise* ausführt, so kann sie vor ihrer Inaktivierung noch letzte Sicherungsaktionen ausführen[2]. Die App wird darüber informiert, sobald sie sich im Stack wieder ganz oben befindet.
Aufgrund der ursprünglichen Konzeption für nutzergetriebene Anwendungsszenarien wurde sich hier für einen rundenbasierten Ansatz entschieden. Eine Runde („Turn") repräsentiert dabei jeweils denjenigen Interaktionspartner, von dem gerade eine Kommunikation ausgehen soll. Es gibt also sowohl „System-Turns" (das System ist an der Reihe) als auch „User-Turns" (der Benutzer ist an der Reihe). Reagiert der Nutzer eine längere Zeit nicht und es findet auch keine anderweitige Interaktion statt, so kann ein Turn ohne Ausführung einer Aktion verfallen. Die gleichzeitige Ansteuerung verschiedener Aktuatorik (bspw. Motorik, Monitor, Sound-Ausgabe, etc...) durch mehrere Applets ist in diesem Konzept nicht vorgesehen. Eine App bekommt während ihrer

[2]Bei konkurrierenden Zugriffen ist dies unumgänglich. In Kapitel 4.5.3 wird hierzu ein praktisches Anwendungsszenario vorgestellt.

Aktivitätsphase gewissermaßen Vollzugriff auf den Roboter eingeräumt, andere Apps können währenddessen nicht zugreifen. Auch wenn dieses Konzept eine hohe Nachvollziehbarkeit aufweist und seinen Grundzügen nach für modulare Erweiterungen ausgelegt ist, scheidet es für die eingangs genannten Anforderungen deshalb leider aus. Viele der hier eingesetzten Mechanismen (active/raise, stabile Sortierung der Apps, etc...) sind jedoch in die später vorgestellte dynamische Ablaufsteuerung übernommen wurden.

3.4 Adaptive und lernende Systeme

Den aktuellen Zustand des Systems möglichst genau beschreiben und prädizieren zu können stellt ein herausragendes Merkmal für eine Ablaufsteuerung dar, da hierdurch der Wartungsaufwand reduziert und Modifikationen transparent gestaltet werden können. In einigen Fällen ist es jedoch durchaus erstrebenswert, Abläufe nicht immer starr modellieren zu müssen, sondern je nach situativem Kontext unterschiedliche Abläufe zuzulassen. Dies ist insbesondere dann bedeutend, wenn zusätzlich eine Adaption an den jeweiligen Nutzer stattfinden soll, da sich Nutzerpräferenzen untereinander stark unterscheiden können. Ebenfalls soll ein multimodales Robotiksystem durchaus eine gewisse Humanoidität ausstrahlen, ein wesentliches Kriterium hierfür ist die Natürlichkeit der Interaktion. Eine wissenschaftliche Aufarbeitung des Themas erfolgte beispielsweise in [Mueller et al., 2014a] und [Mueller et al., 2014b]. Dort wurde ein adaptives System zur Verbesserung der Interaktivität vorgestellt, welches eine Erweiterung zum in Kapitel 3.3.3 vorgestellten Stack-basierten System darstellt. Dort wurde ebenfalls festgestellt, dass klassische Steuerarchitekturen für mobile Roboter nur einen geringen Teil der Anforderungen abdecken.
Die Basis des Systems stellt auch in diesem Fall eine Behavior-State-Machine (siehe Abb. 3.7), welche das Roboter-Verhalten ausgehend von der aktuellen Situation steuert. Eine zusätzlich eingeführte Komponente für Nutzerinteraktionen, der *„Dialog-Manager"*, kann für die Behavior-State-Machine dabei den jeweiligen Zielzustand angeben. Jeder Zustand (einschließlich der auf dem Weg zum Zielzustand durchlaufenen) wird auf ein entsprechendes Behavior gemappt. Ein Nachteil bei dieser Imple-

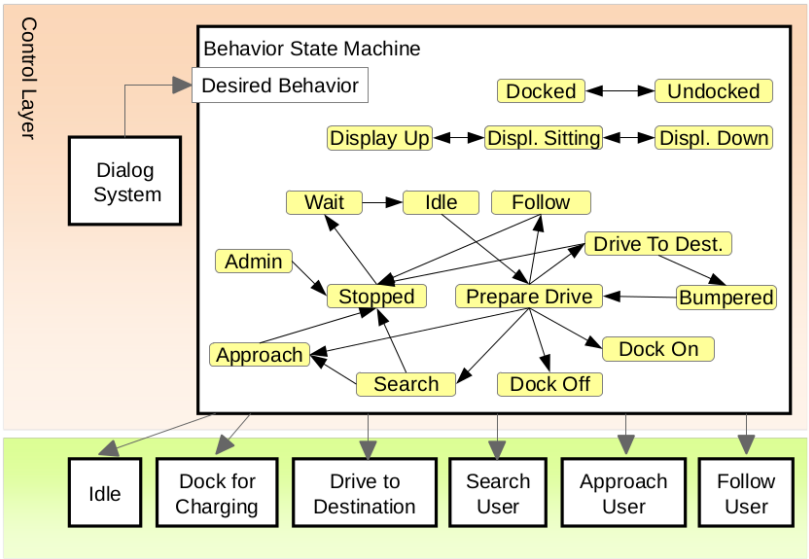


Abbildung 3.7: Behavior-State-Machine des Stack-basierten Systems

*Diese Darstellung des Konzeptes aus [*Mueller *et al., 2014a] zeigt den Aufbau der verwendeten Behavior-State-Machine. Ein Dialog-System teilt dieser das gewünschte Verhalten mit. Die jeweiligen Zustände werden dann auf ein entsprechendes Behavior (grün) abgebildet.*

mentierung ist ebenfalls, dass zumindest innerhalb der Behavior-State-Machine alle Übergangsvarianten von einem Zustand in einen anderen überprüft und gegebenenfalls modelliert werden müssen. Erweiterungen um neue Behaviors müssen deshalb explizit in der Behavior-State-Machine berücksichtigt werden. Der Fokus bei dieser Implementierung bestand jedoch weniger in der Erweiterbarkeit um neue Behaviors - also motorische robotische Verhaltensweisen - als in der Erweiterung der multi-modalen Interaktionsfähigkeiten des Gesamtsystems.

Auch bei der Umsetzung der Ablaufsteuerung als adaptives und lernendes System wird eine Erweiterung also vornehmlich durch Hinzufügen neuer Apps vorgenommen, welche dann wiederum in ein gemeinsames Dialog-System integriert werden. Eine *„Service-Applikation"* genannte App stellt dabei eine *Service-Funktionalität*, ein *GUI-Widget* sowie ein *„Dialog-Frame"*. Das Dialog-Frame stellt den Kontext der aktuellen Interaktion zwischen dieser App und dem Nutzer dar. Hierzu kennt es den aktuellen *Dialog-Zustand*, mögliche *Input-Semantiken* sowie seine zur Verfügung stehenden *Aktionen* mit einer entsprechenden *Policy*. Die Policy beschreibt dabei die Regeln zur Abbildung von Eingabe-Semantiken innerhalb des aktuellen Gesprächskontexts auf jeweilig mögliche Folgezustände. Die Zusammenhänge der einzelnen Komponenten sind noch einmal in Abb. 3.8 dargestellt.

Die Ansteuerung der Zustände wird über das Dialog-System vorgenommen, welches durch das aktuelle Frame konfiguriert wird. Es interpretiert Nutzereingaben durch einen *„Input-Interpreter"* auf Basis einer *input fusion*, also der Vereinigung verschiedener Eingaben, und führt ein *Labeling*, also eine Zuordnung zu einer Semantik, aus. Diese leitet er an einen *Dialog-Manager* weiter, welcher mithilfe eines *Dialog-Agent* auf Basis der Policy eine Auswahl einer Aktion vornimmt. Zur Verfügung stehende Aktionen werden in der Policy des Dialog-Frames festgelegt. Der so genannte *Output-Renderer* nutzt diese Auswahl dann, um die entsprechende Aktion (bspw. eine Audioausgabe, ein Umschalten der GUI etc.) auszuführen. Darüber hinaus kennt das Konzept einen Emotionsmanager, welcher die Eingabedaten zum Aufbau einer Roboteremotion nutzt und interpretiert. Die dabei entstehende Emotion kann ebenfalls als Basis für weitere Interaktionen dienen oder diese ausgestalten.

Dieser Komponentenaufbau dient zunächst zwar lediglich als Basis für adaptive Dialo-

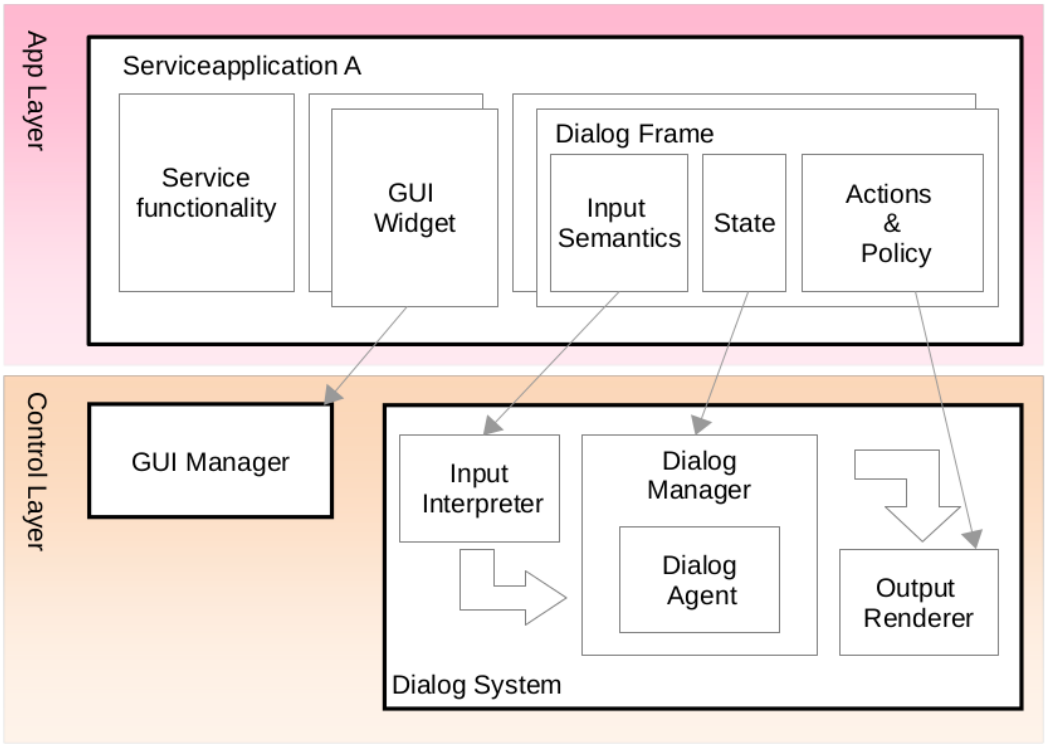


Abbildung 3.8: Grundsätzlicher Aufbau sowie Komponenteninteraktionen einer App auf Basis des Konzeptes der adaptiven, lernenden Ablaufsteuerung
Zu sehen sind die wichtigsten Interaktionen einer App (Serviceapplication A) innerhalb der Ablaufsteuerung aus Kap. 3.4. Die Hauptkomponenten des Gesamtsystems stellen die Apps, der GUI-Manager, sowie das Dialog-System. Apps wiederum bestehen aus ihrer realisierten Funktionalität, einem GUI-Widget sowie einem Dialogkontext (Dialog Frame).

ge, die multimodale Interpretation sowie die jeweiligen Policies erlauben jedoch einen weitgehenden Gestaltungsspielraum bei der Implementation von Apps, wobei Status und Emotion von Nutzer und Roboter weitgehenden Einfluss auf die jeweilige Dialoggestaltung haben können. Die Policy definiert dafür einen Entscheidungsbaum, welcher als Basis für „erlaubte" Interaktionen dient. In den Blättern des Baums existieren dann jeweils mehrere Interaktionsmöglichkeiten (Actions). Das Konzept ist dabei darauf ausgelegt, Dialogabläufe auf Basis dieser möglichen Aktionen lernen und adaptieren zu können.
Die Nutzung adaptiver Dialogsysteme oder sogar adaptiver Ressourcenzuteilungen lässt sich durchaus mit den hier gesetzten Zielen vereinen. Da lernende Dialogsysteme insbesondere für die Intuitivität große Vorteile versprechen, wird bereits an einer Übertragung des Prinzips auf Basis des hier vorgestellten Konzeptes gearbeitet (siehe auch Kap. 6.2).

3.5 Robotics in Concert (ROS - Rocon)

Bei *Robotics in Concert* (kurz **Rocon**) handelt es sich um ein Open-Source-Projekt unter der Leitung der Firma *Yujin Robots*, der *Open Source Robotics Foundation* sowie der *Learning Agents Research Group* der University of Texas, Austin. Rocon bezeichnet sich dabei selbst als *„paradigm shift"* - also als Paradigmenwechsel - im Bereich Robotik, weg von der Idee „Der Roboter ist die Lösung" hin zu „Der Roboter ist Teil der Lösung". Der Grundgedanke rührt dabei daher, dass wesentliche Erfolgsgeschichten der Robotik auf *standalone*-Lösungen basieren. Da diese nur mit hohem Entwicklungsaufwand realisierbar sind, ist es schwierig, Marktnischen zu finden, die ausreichend erfolgversprechend sind, um ein rentables Geschäftsmodell aufzubauen. Rocon selbst ist dabei als Erweiterungsmodul für das Robot Operating System (kurz **ROS**) konzipiert [Yujin-Robot et al., 2014].

3.5.1 Zielstellung und Herangehensweise

Die Leitziele von Rocon sind:

- billigere Roboter - das Benutzen (aus Sicht des Roboters) externer Ressourcen um Hardwarekosten zu reduzieren und die interne Hardware-Komplexität zu verringern

- reinstruierbare Roboter - der Roboter soll zur Lösung beitragen (aber nicht die Lösung sein), größere Anwendungsmöglichkeiten sollen dabei die Marktchancen erhöhen

- Interaktivität - Menschen sollen als Teampartner agieren können (*Anm.: z. B. als Teil einer Service-Tätigkeit, die sonst autonom erfolgt*)

- neue Märkte - Mehrrobotersysteme eröffnen ein neues Feld der Mensch-Maschine-Interaktion

Das Rocon-Projekt befasst sich dabei im Wesentlichen mit der Nutzung mehrerer Roboter zur Lösung komplexerer Probleme. Jeder einzelne Roboter agiert dabei als Mitarbeiter und führt einen mehr oder weniger spezifischen Arbeitsanteil aus. So wurde im Rahmen einer Demo im Juni 2013 und später im Mai 2014 demonstriert, wie mehrere Roboter jeweils Teile des Bestell- und Serviervorgangs in öffentlichen Cafés übernehmen könnten. Eines der wesentlichen Unterkonzepte von Rocon ist das des „Appable Robot", also eines erweiterbaren und rekonfigurierbaren Roboters. Hierfür wird ein eigenes Framework entwickelt, welches aus Sicht der Entwickler insbesondere Vorteile für:

- Software-Installation,

- Ausführen von Aufgaben,

- Neusetzung von Aufgaben,

- Konnektivität, sowie

- Schreiben von portierbarer Software

verspricht. Der Aufbau eines Appable Robot ist in Abb. 3.9 dargestellt.

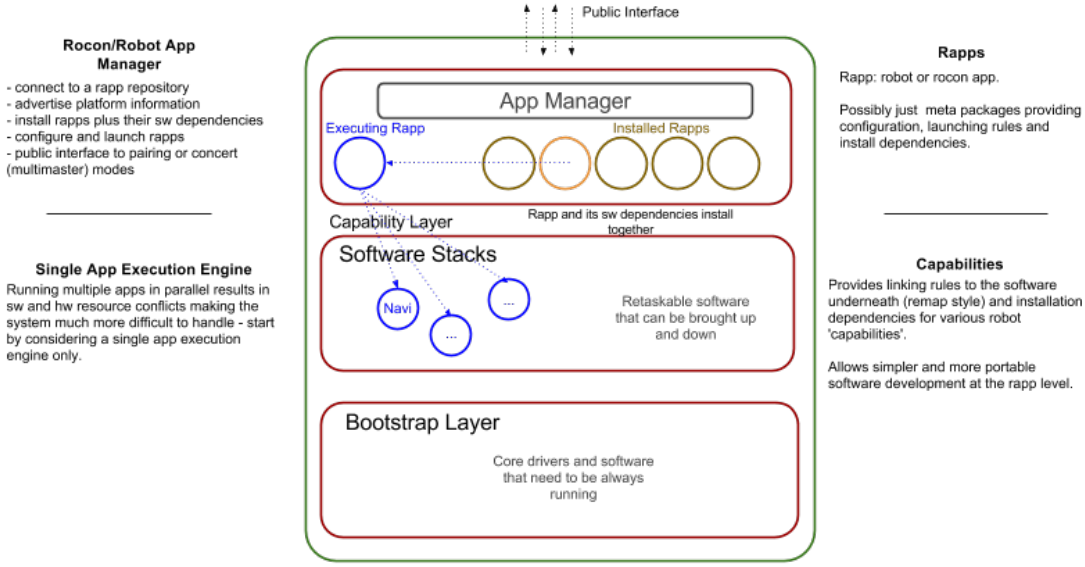


Abbildung 3.9: Robotics in Concert - Das Prinzip des Appable Robot

Unter vielen installierten Apps (Rapps) aus einem gemeinsamen Repository ist stets eine aktiv. Diese wiederum hat freien Zugriff auf alle robotereigenen Ressourcen (Software Stacks). Das Aktivieren der Rapps und Koordinieren der Aufgaben erfolgt durch den verteilten Scheduler. Quelle: http://www.robotconcert.org/index.php/Appable_Robot (abgerufen am 13. Juni 2015).

3.5.2 Gemeinsamkeiten und Abgrenzung

Das Rocon-Framework bietet ähnlich wie die hier vorgestellte Ablaufsteuerung eine eigene Realisierung eines Task Schedulers an. Dieser fungiert dabei, analog zu dem hier vorgestellten Scheduler, als Koordinator für die Zuteilung von Ressourcen. Rocon legt den Fokus dabei jedoch auf die verteilte Abarbeitung **einer gemeinsamen** Aufgabe sowie deren Unteraufgaben. Innerhalb einer Teilaufgabe agiert ein Roboter jedoch weitestgehend unabhängig von Ressourcen[3] und folgt dem Ablauf **eines** Applets, respektive Aufgabenteils. Das Ziel ist also nicht, viele verschiedene Aufgaben gleichzeitig von einem Roboter berücksichtigen, sondern stattdessen viele Roboter jeweils eine eigene Aufgabe erfüllen zu lassen. Die simultane Ausführung gleich zweier oder noch weiterer Tasks wird bewusst ausgeschlossen, worin sich Rocon also substantiell von dem hier vorgestellten Konzept unterscheidet. Der Fokus liegt bei Rocon vielmehr auf der Aufteilung der Ausführung einer Aufgabe, statt auf dem gleichzeitigen Bearbeiten mehrerer Aufgaben *innerhalb* eines Robotiksystem [Yujin-Robot et al., 2014]. Die Herangehensweise sowie verwendete Konzepte lassen sich folglich nur bedingt übertragen. Die für diese Arbeit festgelegten Zielstellung lassen sich allein durch die in Rocon verwendeten Prinzipien jedenfalls nicht erfüllen.

3.6 Zwischenfazit

Relevant für diese Arbeit sind insbesondere die Interaktionen verschiedener Applets untereinander. Diese sind zu komplex, um ohne Einschränkungen und mit geringem Aufwand allein durch eines der vorgestellten, formalen Systeme vollständig realisiert zu werden. Die gesetzten Anforderungen werden also mit keinem der bereits existenten Konzepte vollständig umgesetzt. Durchaus lassen sich jedoch viele der Teilkonzepte übertragen. Es spricht beispielsweise einiges dafür, einzelne Komponenten des Gesamtkonzeptes (z. B. Applets) weiterhin auf Basis von State-Machines zu implementieren. Die Interaktionen zwischen ihnen weisen dann einige Ähnlichkeit zum Multi-State-Machine-Konzept auf, erweitert um tatsächliche Nebenläufigkeit. Hierarchische State-

[3] diese dienen eher des wechselseitigen Ausschlusses des Zugriffes *verschiedener* Roboter

Machines finden zwar kein direktes Adäquat, das Konzept der Hierarchien kann jedoch auf Service-Ebene weiterhin umgesetzt werden und bietet dort ähnliche Vorteile. Stack-basierte Systeme weisen vermutlich die meisten Ähnlichkeiten zum hier vorgestellten Konzept auf, unterscheiden sich jedoch ebenfalls durch die nicht realisierte Nebenläufigkeit sowie das zentralisierte Know-how. Die insbesondere dem Nutzer gegenüber kommunizierte Reihenfolge von Aktivitäten wird allerdings großteilig deckungsgleich sein. Weitestgehend vollständig wird das Konzept adaptiver Systeme Einzug in die hier vorgestellte Ablaufsteuerung finden, da es sich beinahe konfliktfrei integrieren lässt (siehe auch Kap. 6.2).

Kapitel 4

Dynamische rekonfigurierbare Ablaufsteuerung

Im Folgenden soll nun ein generisches Konzept entwickelt werden, welches die wesentlichen Probleme aus Kapitel 3 adressiert und insbesondere Wert legen soll auf:

- Nebenläufigkeit,
- Modularität,
- Wiederverwendbarkeit, und
- Nachvollziehbarkeit.

Die Gründe für diese Ziele finden sich in Kapitel 1.2.

4.1 Grundbegriffe und Konzepte

Um diese Ziele zu realisieren, wurde sich für einen (teil-)hierarchischen (siehe Kap. 4.1.2) Lösungsansatz entschieden. Im Zentrum dieses Konzeptes stehen die Begriffe:

- Scheduler,
- Service,

- Ressource,
- Applet und
- Service-Applet.

Der Begriff *Scheduler* steht dabei wie bereits in Kapitel 2.2 in Analogie zum Begriff des Betriebssystem-Schedulers. Sein Zweck besteht in der Steuerung der Zugriffe auf *Services*. Er stellt gewissermaßen die höchste Instanz innerhalb des Systems dar (siehe auch Kap. 4.1.2).
Ein Service stellt dabei eine funktionale Einheit des Gesamtsystems dar, welche *einen bestimmten Zweck* erfüllt. Dieser Zweck kann beispielsweise das Erkennen von Personen oder das auditive Interpretieren von Sprachkommandos sein. Ebenso kann er darin liegen, die Ansteuerung des Roboter-Displays, oder der Roboter-Aktuatorik vorzunehmen.
Da es durchaus möglich ist, dass ein derartiger Zugriff auf einen Service einen anderen Zugriff ausschließt (z. B. Roboter-Aktuatorik), existiert das Konzept der *Ressource*. Diese stellt gewissermaßen eine Kopie des Services dar, auf welche dann wie auf den tatsächlichen Service zugegriffen werden kann (siehe Kap. 4.2.2). Sie dient dabei als Proxy und leitet die Zugriffe an den Service weiter.
Die zugreifende Instanz ist üblicherweise ein *Applet* (kurz: App). Dieses erfüllt als Teil des Gesamtsystems einen gewissen Zweck, welcher direkt oder indirekt dem Nutzer zur Verfügung steht. Sie stellt deshalb normalerweise eine Benutzeroberfläche, modelliert konkrete Abläufe und greift hierfür auf die Ressourcenobjekte zurück.
Auch existiert der Fall, dass ein Applet *keine* Benutzeroberfläche stellt. Dies gilt insbesondere dann, wenn es nicht zur direkten Interaktion mit dem Nutzer gedacht ist, sondern seine (üblicherweise nutzerzentrierten) Abläufe als Basis für die Aufgaben anderer Applets dienen. Es stellt seine Funktionalität anderen Applets zur Verfügung und wird dann als *Service-Applet* bezeichnet[1].

[1] Die Unterscheidung zwischen Service-Applet und Applet erfolgt jedoch nicht auf Basis der Tatsache, ob ein Applet eine Benutzeroberfläche hat, sondern dadurch, dass ein Applet anderen seine eigene Funktionalität zur Verfügung stellt.

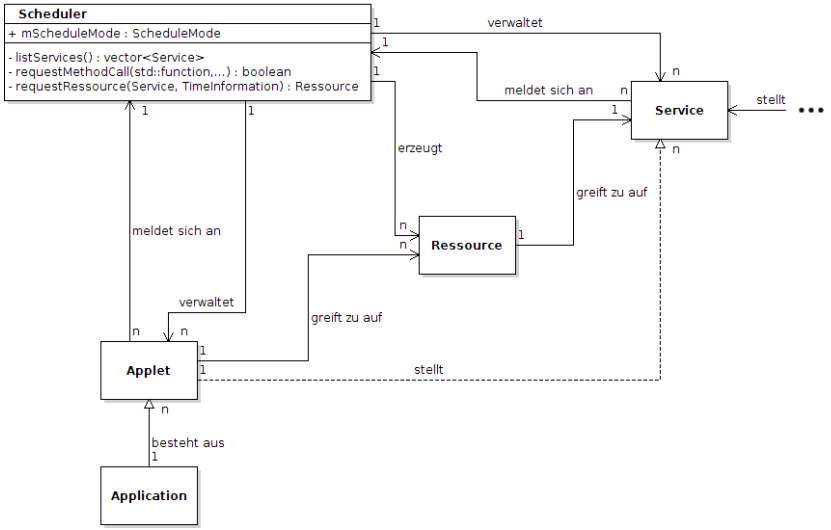


Abbildung 4.1: Komponentendiagramm zum Grundaufbau des Schedulers
Dargestellt sind alle relevanten Grundkomponenten sowie deren Zusammenspiel im Konzept der dynamischen Ablaufsteuerung.

4.1.1 Aufbau und Informationsfluss

Die dominierenden Abläufe innerhalb der Scheduler-Logik bestehen Applet-seitig im:

- Beantragen von Ressourcen (geteilt oder exklusiv), sowie im
- Benutzen von Ressourcen.

Beide Aktionen finden ausschließlich in Übereinkunft mit dem Scheduler statt. Er entscheidet dabei:

- ... welches Applet auf eine Ressource zugreifen darf,
- ... wann dieses Applet Zugriff erhält,
- ... wie lange auf die Ressource zugegriffen werden darf,
- ... wann ein Ressourcenzugriff zu beenden ist.

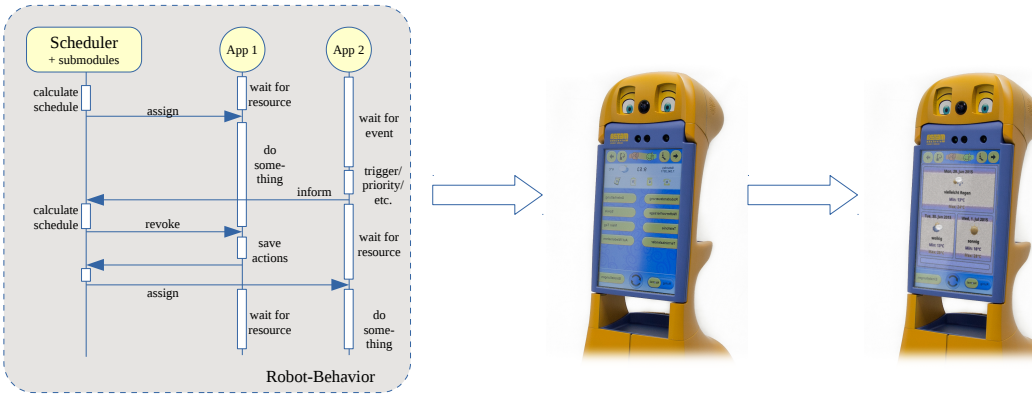


Abbildung 4.2: Ablauf der Neuzuteilung von Services

Dargestellt sind die notwendigen Schritte zur Neuzuteilung eines Services von einem Applet zu einem anderen. Der Scheduler erteilt und entzieht dabei Zugriffsgenehmigungen auf Ressourcen (hier: Monitor).

Dabei dient er als Registrar für Services und Applets, wird über deren An- oder Abwesenheit also jeweils informiert, und weiß, wie das jeweilige Applet oder der jeweilige Service zu adressieren sind und was ihr aktueller Status ist. Der Scheduler dient damit als Herz der Kommunikation auf Applet- und Application-Ebene. Die Zusammenhänge zwischen den einzelnen Komponenten sind in Abb. 4.1 dargestellt, der grundsätzliche Ablauf beim Wechsel von Ressourcenzuteilungen findet sich in Abb. 4.2.

4.1.2 „Gebundene Kooperativität" als Grundgedanke

Trotz der Entscheidung für einen zunächst einmal hierarchischen Ansatz handelt es sich bei der hier vorgestellten Ablaufsteuerung keineswegs um ein ausschließlich hierarchisch organisiertes Konzept. Vielmehr stellt der Scheduler einen Mediator dar, welcher Zugriffswünsche vieler Applets auf physisch einmalige Services koordiniert indem er virtuelle Verweise auf Service-Objekte vergibt, hier als *Ressourcen* bezeichnet (siehe Kap. 4.2.2). Dem Scheduler dient somit in erster Linie als Vermittler. Als letztes Mittel steht ihm jedoch der Ressourcenentzug und damit auch die Verhinderung

weiterer Zugriffe auf den Service durch das jeweilige Applet zur Verfügung.
Das grundsätzliche Ziel einer jeden Ressourcenzuteilung an ein Applet ist weiterhin die vollständige Abarbeitung der jeweiligen Aufgabe und die daraufhin erfolgende **appletseitige** Freigabe der Ressource. Sollte jedoch trotz allem eine Unterbrechung der aktuellen Aktion notwendig sein, beispielsweise um Nutzerwünschen nachzukommen, wichtige Servicefunktionen termingetreu auszuführen oder sogar um ernsthafte Verletzungen umstehender Personen oder Schäden am Roboter zu vermeiden, so existieren für die Kommunikation dieser Dringlichkeiten mehrere Möglichkeiten:

- Deadlines (Kapitel 4.5.1)
- Dynamische Prioritäten und Triggering (Kapitel 4.5.2)
- Callbacks, Interrupts und Verhandlungsrunden im Falle eines Ressourcenentzugs (Kapitel 4.5.3)
- Interkomponentenkommunikation (auf Basis der jeweiligen Implementierung)

Während der Umverteilung von Ressourcen berücksichtigt der Scheduler all diese Möglichkeiten. Weiterhin wird ein geplanter Ressourcenentzug dem die Ressource haltenden Applet mitgeteilt, so dass dieses durch die oben genannten Mechanismen den ursprünglich geplanten Zuweisungsplan noch weitreichend beeinflussen kann. Die Abläufe im Einzelnen sind unter Kapitel 4.1.1 beschrieben. Eine Diskussion der auftretenden Probleme erfolgt in Kapitel 4.4, Lösungsansätze zur Reduktion dieser finden sich in Kapitel 4.5.

4.1.3 Abgrenzung zu bisherigen Ansätzen

Das hier vorgestellte Konzept weicht in einigen wesentlichen Punkten von den in Kapitel 3 genannten ab: State-Machines sowie darauf aufbauende Modelle zielen zunächst auf die Modellierung eines jeweils festen und immer wieder gleichen Ablaufes ab. Im Rahmen dieser Arbeit gilt es jedoch eine Ablaufsteuerung zu finden, die in erster Linie

der **Koordination** der Zugriffe *verschiedener* Abläufe dient[2]. Dabei soll die *interne Zustandsmodellierung* jedoch dem eigentlichen Applet obliegen. Die dynamische Ablaufsteuerung modelliert deshalb lediglich die Objekte (Services bzw. Ressourcen), mit denen Applets arbeiten, nicht aber deren eigene Abläufe.
Es ergibt sich deshalb eher eine gewisse Nähe zum Konzept der Multi-State-Machine, wobei dann jedes Applet als Black-Box betrachtet würde. Jeder Service hat einen fest definierten Zustand („Welche Applets dürfen auf mich zugreifen?"). Transitionen (hier also Interaktionen zwischen Applets und Scheduler) erzeugen dann Veränderungen der Zustände von Applets und Services. Die Trennung der für den Nutzer sichtbaren Schichten, wie UI, Verhaltensweisen, Aktuatorik, etc., sowie die Kapselung der funktionalen Einheiten soll hier jedoch noch einmal deutlich konsequenter und feingranularer als bei Multi-State-Machines erfolgen, jedoch weiterhin generisch sein.
Die größten Ähnlichkeiten weist vermutlich das Konzept der Stack-basierten Systeme auf. Dies liegt natürlich auch darin begründet, dass wesentliche Erfahrungen und Lösungsansätze direkt in die Entwicklung der hier vorgestellten Ablaufsteuerung eingeflossen sind. Das vorgestellte Stack-basierte System nutzt ebenfalls eine Reihe von Mechanismen, die sich hier wiederfinden. Beispielsweise soll bei Schließen einer App stets die letzte, zuvor geöffnete App wieder aktiviert werden. Ebenfalls existiert dort ein Triggering-Mechanismus, mit welchem es möglich ist, eine App in den Vordergrund zu schalten. Während dies bei Stack-basierten Systemen über die Ablage auf einen Stack erfolgt, wird hier eine stabile Sortierung als Basis dieser Funktion benutzt (siehe auch Kap. 4.5.2).
Ein wesentlicher Unterschied zwischen den meisten vorherigen Herangehensweisen und der hier vorgestellten ist jedoch die weitestgehend explizite Modellierung aller Zustände in den State-of-the-Art-Modellen. Dies erfordert einen hohen Grad an Abstimmung zwischen den einzelnen Komponenten und ist in erster Linie für fest-konfigurierte Systeme geeignet. Für diese bringen sie entscheidende Vorteile mit sich, die mit einem dynamischen Konzept nur schwierig zu erreichen sind:

- fester Systemzustand (keine *inneren Zustände*)

[2]Ein Ablauf ist hier gleichbedeutend mit dem Begriff des Applets

- weitestgehend einfache Validierbarkeit
- weniger aufwändige Fehlerbehebung

Rocon stellt diesbezüglich das einzige System dar, welches ausgesprochen ähnliche Fehlerquellen begünstigt. Generell existieren sehr viele Gemeinsamkeiten zwischen diesem und dem in Rocon benutzten Konzept. Der wesentliche Unterschied besteht jedoch in der Ausrichtung auf die Koordination von Abläufen eines (wie in diesem Konzept) oder vieler (Rocon) Roboter. Bei Rocon wird dabei nicht versucht, die *internen* Abläufe der einzelnen Roboter-spezifischen Apps zu koordinieren. Ein Roboter insgesamt, und nicht seine funktionalen Komponenten, dienen als Service-Provider (genauer wurde hierauf bereits in Kap. 3.5.2 eingegangen).

4.2 Komponentenaufbau

Zum Aufbau der Scheduler-Logik aus Kapitel 4.1 sind verschiedene Grundkomponenten nötig. Die wichtigsten von ihnen wurden bereits in Kapitel 4.1 genannt. In diesem jedoch soll vermehrt auf deren Interaktionen untereinander, sowie auf einige erweiterte Konzepte eingegangen werden. Die grundsätzliche funktionale Architektur der neu eingeführten Komponenten findet sich in Abb. 4.1, ihre Eingliederung in die Gesamtarchitektur ist in Abb. 4.3 wiedergegeben. Die Übersichtsgrafik in Abb. 4.4 zeigt noch einmal alle Interaktionen sowie einen groben Umriss des Gesamtkonzeptes, welches in den folgenden Kapiteln genauer beschrieben wird.

4.2.1 Intuitive Definition

Hier sollen noch einmal einfache, intuitive Definitionen für die wesentlichen Begriffe aus Kapitel 4.1 sowie einige weitere Begriffe gegeben werden:

- Der *Scheduler* ist der zentrale Steuermechanismus. Er nimmt Anfragen entgegen und erlaubt oder verbietet Zugriffe auf Services.
- Ein *Service* ist, lapidar ausgedrückt: „etwas, was eine Funktion für andere erledigt", stellt also nach außen eine Funktionalität zur Verfügung.

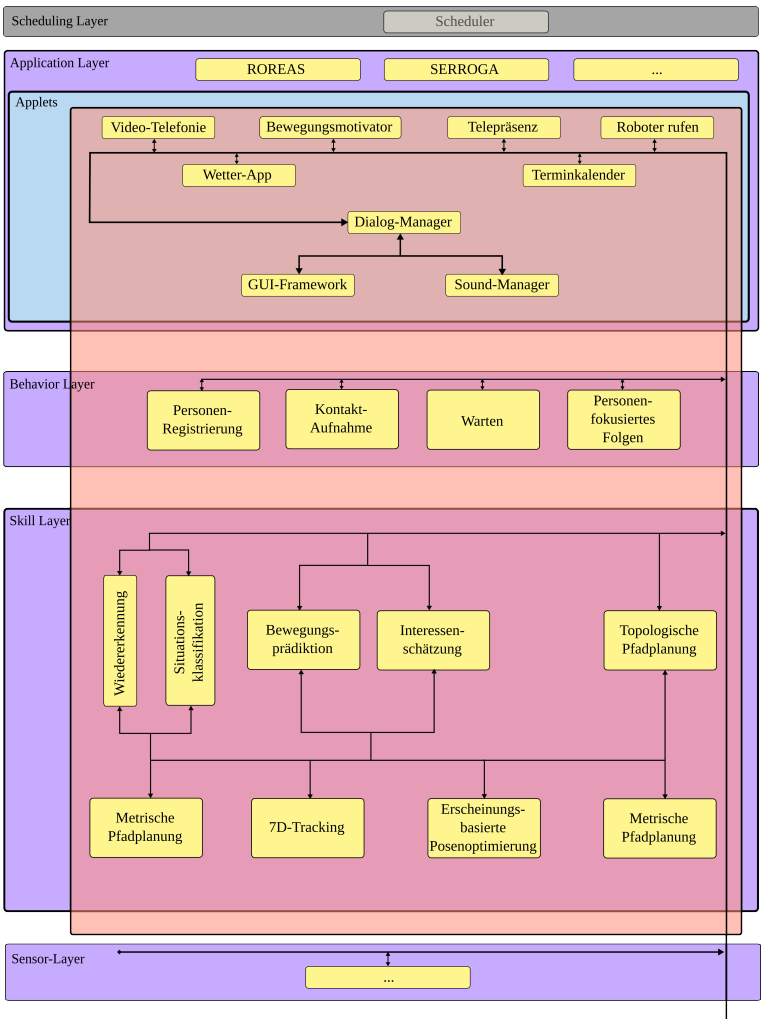


Abbildung 4.3: Funktionale Architektur der dynamischen Ablaufsteuerung

*Zu sehen ist die Aufteilung in einzelne, funktionale Layer. Da der Scheduler rein administrative Aufgaben übernimmt, steht er ausgegraut über der Komponentenhierarchie. Es existiert für ihn keine unmittelbare Unterscheidung zwischen Services, die durch in der Hierarchie oben stehende Instanzen (wie z. B. Applets) oder aber Services die durch die darunterliegenden Ebenen (Behaviors, Skills, ...) gestellt werden. Dies erlaubt die vereinfachende Definition eines Service-Layers (roter Kasten). Die dargestellten Applets, Behaviors und Skills dienen der Veranschaulichung und sind als exemplarisch zu betrachten. Die Architektur lehnt sich stark an die ROREAS- bzw. SERROGA-Architektur an (vgl. [*Gross *et al., 2014*]).*

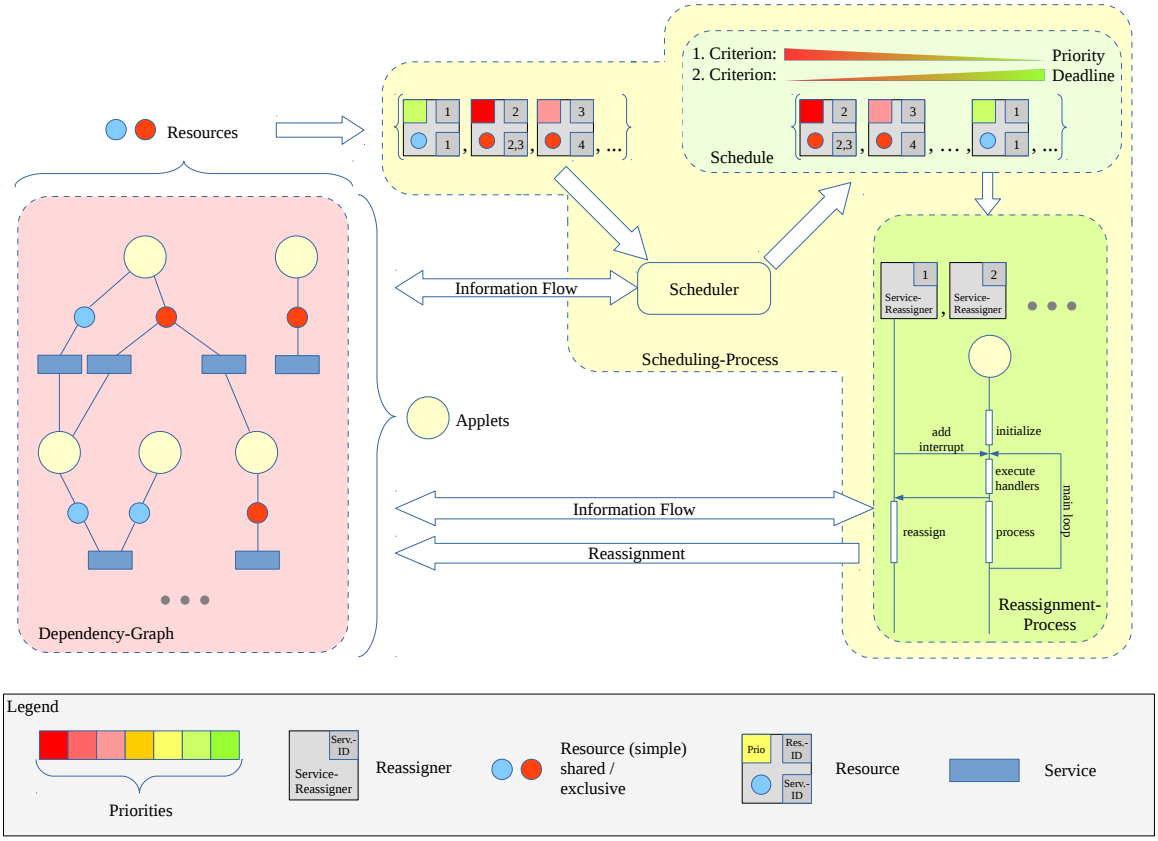


Abbildung 4.4: Übersicht Komponenteninteraktionen

Auf dieser Grafik befinden sich bereits die wichtigsten Interaktionsabfolgen zur Realisierung der eigentlichen Ablaufsteuerung. Die einzelnen Abläufe werden in den Folgekapiteln noch einmal genauer besprochen. Eine Übersichtsgrafik findet sich im Anhang unter A.1.

- Eine *Ressource* ist der Zugriffsmechanismus auf diese Funktionalität (siehe Kap. 4.2.2).

- Ein *Applet* benutzt Ressourcen und bietet eine direkte Funktion für den Nutzer (im Gegensatz zum Service-Applet).

- Ein *Service-Applet* stellt seine Funktionalität für andere Applets (siehe Kap. 4.2.3) und indirekt für den Nutzer zur Verfügung.

Für die Kommunikation und Interaktion zwischen den Komponenten werden weiterhin folgende Konstrukte benötigt:

- Ein *Interrupt* bezeichnet die vorübergehende Unterbrechung der normalen Abarbeitung eines Applets (siehe auch Kap. 4.4.1).

- Eine *Deadline* bezeichnet die Definition des spätesten Zeitpunktes, zu dem eine bestimmte Aufgabe erledigt sein muss (siehe Kap. 4.5.1).

- Eine *Priorität* entspricht der Dringlichkeit einer Aufgabe. Aufgaben höherer Priorität werden Aufgaben niedrigerer stets vorgezogen (siehe Kap. 4.5.2).

- Eine *Verhandlungsrunde* ist - z. B. bei der Neuzuteilung von Services - die Anfrage eines Applets zum Überdenkenken der Neuzuteilung (beispielsweise durch Aufschub oder Einwände, siehe Kap. 4.5.3).

- Ein *Schedule* (auch: Zuteilungs- oder Ressourcenplan) ist die aktuell vom Scheduler vorgesehene Zuweisung und Übersicht über alle Ressourcen.

4.2.2 Services vs. Ressourcen

Wie bereits zuvor angedeutet, besteht ein wesentlicher Unterschied in der Definition der Begriffe „Service" und „Ressource". Das Konzept der Ressource wurde eingeführt, um einen möglichst großen Gestaltungsspielraum bei der Implementierung von Applets zu gestatten und ist auch einigen Eigenheiten der benutzen Programmiersprache

(*C++*) geschuldet. Ist hier die Speicheradresse eines Objektes bekannt, so ist ein direkter Zugriff auf das Objekt möglich, außer, es stellt selbst Mechanismen zur Einschränkung des Zugriffes zur Verfügung. Üblicherweise unterscheidet ein Objekt jedoch nicht danach, *von wo* es benutzt wird.

Das Ziel besteht also darin, eine Entkoppelung für die physisch einmaligen Services derart zu schaffen, dass beliebige Applets, zumindest innerhalb der Zeit, in der ihnen ein Service zur Verfügung gestellt wird, so darauf zugreifen können, als würden sie den tatsächlichen Service verwenden. Es muss also zwischen der Erlaubnis eines Zugriffes und den tatsächlichen Zugriffen auf einen Service stets eine Synchronisation aller beteiligten Applets stattfinden (Applet A darf nicht zugreifen, wenn Applet B zugreift, und umgekehrt). Ohne eine geeignete Zwischenschicht muss jede Applet-Implementierung hierauf gesondert Rücksicht nehmen. Die Alternative hierzu wäre, den Synchronisationsmechanismus in jedem einzelnen Service unterzubringen. Das Konzept der Ressource entkoppelt diesen sonst redundant implementierten Mechanismus: Jedes Applet hält seine eigene Ressource und keine zwei Applets halten dieselbe Ressource. Ressourcen sind somit eine surjektive Abbildung auf die Menge der Services und stellen zusätzlich einen Synchronisationsmechanismus dar (siehe Abb. 4.5). Dies erlaubt zum einen, Zugriffe zu entkoppeln und Funktionen zur Synchronisation auch zwischen verschiedenartigen Services zu teilen, zum anderen stellt es eine einfache Implementierung von Exklusivität und Nebenläufigkeit von Zugriffen auf den gleichen Service zur Verfügung. Ist ein exklusiver Zugriff auf den Service gewünscht, werden alle ebenfalls auf diesen Service verweisenden Ressourcen blockiert. Ist ein geteilter Zugriff gewünscht, werden alle exklusiven Ressourcen blockiert und alle nebenläufigen aktiviert.

Eine weitere wichtige Eigenschaft ist, dass die eigentliche Synchronisation nun direkt auf Basis der Verfügbarkeit eines Services vorgenommen werden kann. Ist der Service frei, so wird die Ressource zugeteilt. Ist er nicht frei, bekommt dasjenige Applet mit der höher priorisierten Ressource den Zugriff auf den Service. Zugriffe, die über die andere Ressource ausgeführt werden, werden einfach blockiert.

Nur am Rande erwähnt sei an dieser Stelle der Begriff des Resource-Bundles. Dabei handelt es sich um die Komposition mehrerer Ressourcen zu einem gemeinsamen Proxy. Dies kann dann wichtig sein, wenn der Zugriff auf einen Service keinen Sinn ohne den

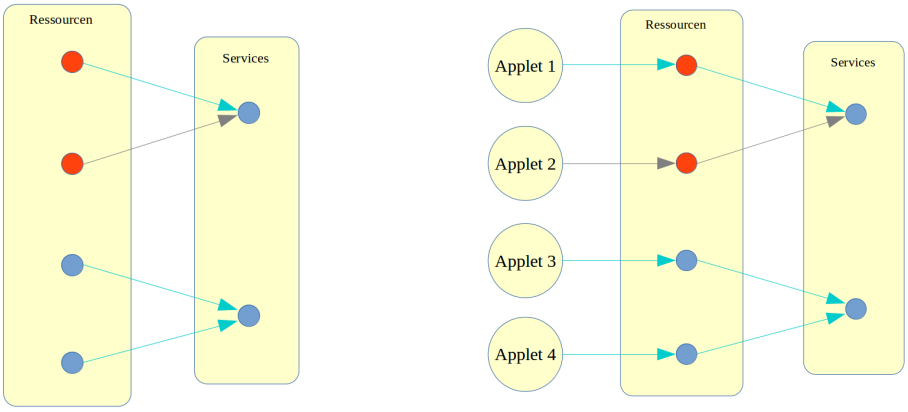


Abbildung 4.5: Aufbau von Services und Ressourcen
Links: Surjektive Abbildung der Menge der Ressourcen auf die Menge aller Services. Rechts: Benutzung der Ressourcen durch Applets und Mapping auf entsprechende Services. Jeweils dargestellt ist auch, ob ein Zugriff (Pfeil) gestattet ist (hellblau), oder nicht (ausgegraut), sowie, ob eine Ressource exklusiven (rot) oder geteilten Zugriff (blau) auf einen Service benötigt.

gleichzeitigen Zugriff auf einen anderen hätte. Ist der Roboter zum Beispiel in der Lage, ein Nutzerinterface zum Anfahren einer Position in einem Raum zur Verfügung zu stellen (siehe *Fahre-Zu-Applet* in Kap. 4.2.3), so muss er hierfür sowohl das *User Interface*, als auch die Aktuatorik benutzen können. Nur eine der beiden Ressourcen reicht für das Applet nicht aus, um für den Nutzer eine sinnvolle Funktionalität zur Verfügung zu stellen. Resource-Bundles verhalten sich ansonsten jedoch wie einzelne Ressourcen. Der Unterschied liegt darin, dass alle Synchronisationsmechanismen auf jeweils mehrere Services wirken (siehe Abb. 4.6).

4.2.3 Hierarchische Service-Definitionen

Ein einfacher Konfliktfall zwischen zwei Ressourcen ist immer dann gegeben, wenn sie gemeinsame Serviceabhängigkeiten besitzen. Existieren beispielsweise die folgenden beiden Applets:

- Fahre-Zu-Applet: Dieses Applet bietet dem Nutzer auf Basis der Navigationskarte eine Vogelperspektive der Wohnung. Der Nutzer kann nun einen Punkt auf

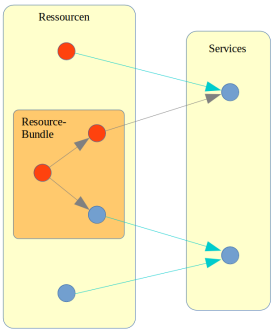


Abbildung 4.6: Gegenüberstellung Ressource und Resource-Bundle

Das Resource-Bundle verhält sich zunächst wie eine normale Ressource. Synchronisationsmechanismen sind jedoch so erweitert, dass das Resource-Bundle nur dann Zugriff erhält, wenn alle untergeordneten Ressourcen ebenfalls Zugriff erhalten. Da einer der beiden Services des Resource-Bundles nicht zur Verfügung steht, wird das gesamte Bundle blockiert. Die verwendeten Farben entsprechen denen aus Abb. 4.5.

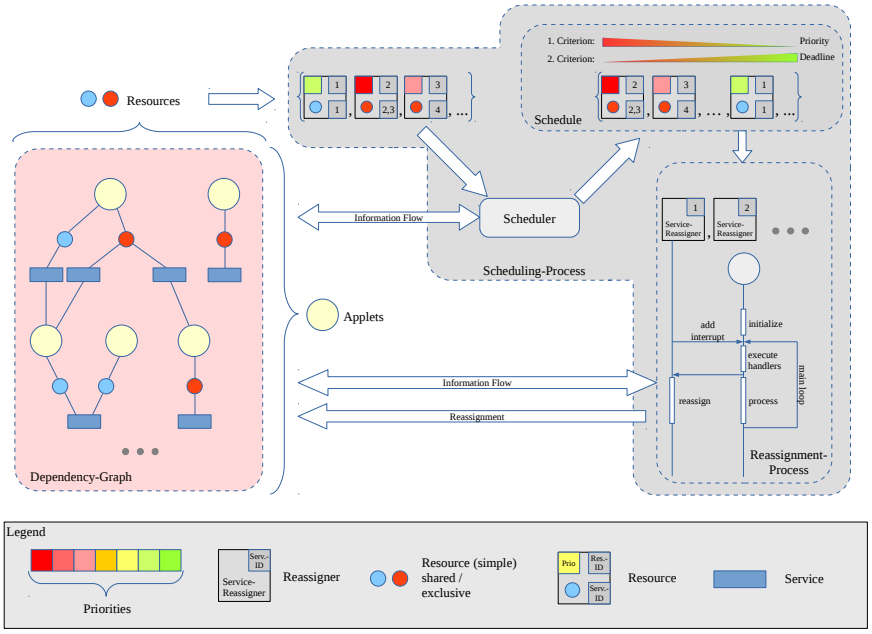


Abbildung 4.7: Einordnung hierarchischer Service-Definitionen in das Gesamtkonzept

Hierarchische Service-Definitionen stellen die Basis für die spätere Wiederverwendbarkeit von Komponenten.

der Karte selektieren, woraufhin der Roboter zu diesem Punkt fährt.

- Batterie-Lade-Applet: Dieses Applet beobachtet den aktuellen Batterie-Ladezustand und kann bei Absinken unter einen festen oder dynamischen Schwellwert automatisch die Roboter-Ladestation anfahren.

Beide Applets benötigen direkten Zugriff auf die Motorsteuerung („*Pilot*"). Ein Nutzer trägt dem Roboter nun auf, über das Fahre-Zu-Applet eine bestimmte Position innerhalb einer Wohnung anzufahren. Dieses schickt daraufhin seiner Motor-Ressource (*sprich: der Proxy dieses Applets zur Motorsteuerung*) das Kommando zum Anfahren der selektierten Position. Die Ressource wiederum leitet dies einfach an den Pilot-Service weiter. Bis zu diesem Punkt entstehen keine Probleme.
Wir gehen nun jedoch davon aus, dass zur selben Zeit der Batterieladezustand unter eine festgelegte Schwelle T fällt. Die Pflicht des Batterie-Lade-Applets ist es nun, wiederum über seine Ressourcen-Repräsentation des Pilot-Services (*seinen Proxy*) diesem das Kommando zum Anfahren der Ladestation zu geben. Es entsteht ein direkter Konflikt zwischen den Wünschen der beiden Applets - hier als **Konflikt ersten Grades** bezeichnet (*direkter Konflikt*; siehe Abb. 4.8). Dieser muss zunächst behoben werden. Konflikte ersten Grades lassen sich durch eine Priorisierung der Applets - bzw. der ihnen zugeordneten Ressourcen - durchaus vermeiden. Dynamischere Techniken sind jedoch unter Kapitel 4.5 vorgestellt.
Aufgrund der Gliederung der Architektur in **mehrere** funktionale Schichten (siehe Abb. 4.3) ist es möglich, dass Services ihrerseits selbst wieder von anderen Services abhängen. Dies kann in einem der folgenden beiden Fälle auftreten:

- Ein Applet stellt einen oder mehrere Services nach außen zur Verfügung, hängt seinerseits aber von anderen Services ab

- Ein Service wird aus einem Skill oder einer Behavior (siehe Kap. 4.3) gebildet, welches selbst wieder auf andere Skills zugreift

Der Ablauf des Batterie-Lade-Applets aus dem oben angeführten Beispiel besteht in der Praxis selbst wiederum aus mehreren Schritten:

- Fahre vor die Ladestation
- Schalte um zu genauerer Lokalisierung[3]
- Fahre langsam an Ladekontakte heran

Es stellt somit ein Behavior dar, welches den zweiten der oben genannten Fälle erfüllt: Ein als MIRA-Unit implementierter Service **Docking-Controller** steuert alle notwendigen Schritte, die für ein automatisches Anfahren der Ladestation nötig sind. Das Batterie-Lade-Applet überprüft deshalb lediglich den Batteriestatus und teilt dem Docking-Controller (über seine Docking-Controller-Ressource) gegebenenfalls mit, er solle die Ladestation anfahren. Der Docking-Controller leitet daraufhin alle notwendigen Schritte ein. Er fällt innerhalb seiner Implementierung jedoch selbst wieder auf den Pilot zurück, welcher ja bereits vom Fahre-Zu-Applet benötigt wird (siehe Abb. 4.8). Existiert hier also keine Modellierung der Service-Abhängigkeiten käme es zwar zu keinem Konflikt ersten Grades (da beide Applets unterschiedliche Services beantragen), sehr wohl jedoch gäbe es aber auf der Service-Ebene konkurrierende Zugriffe auf die Motorsteuerung (**Konflikt zweiten Grades**).
Umgehen könnte man dieses Problem natürlich durch eine flache Architektur, jedoch ginge damit ein Großteil der Erweiterbarkeit - eines der wesentlichen Kriterien für die Verwendung einer dynamischen Ablaufsteuerung - verloren. Wird also aus einem Service eine Ressource gebildet, muss bei der Zurverfügungstellung dieser jeder andere Ressourcenzugriff mit überschneidenden Abhängigkeiten blockiert werden. Eine Bekanntmachung der direkten - nicht jedoch der indirekten - Beziehungen untereinander ist also zwingend erforderlich. Da jedes Applet selbst wieder einen Service bereitstellen kann, sind durchaus auch Konflikte dritten, vierten und höheren Grades denkbar.

Spezifische Probleme

Durch die so entstehenden Abhängigkeiten erwachsen einige zusätzliche Schwierigkeiten. Dies spiegelt sich zum einen in sehr klassischen (siehe Kap. 4.4.2), zum anderen

[3]Um die Ladekontakte zu treffen wird eine deutlich genauere Lokalisierung benötigt, als dies beim normalen Fahren der Fall ist. Sie basiert auf der Erkennung eines Schachbrett-Musters an der Ladestation.

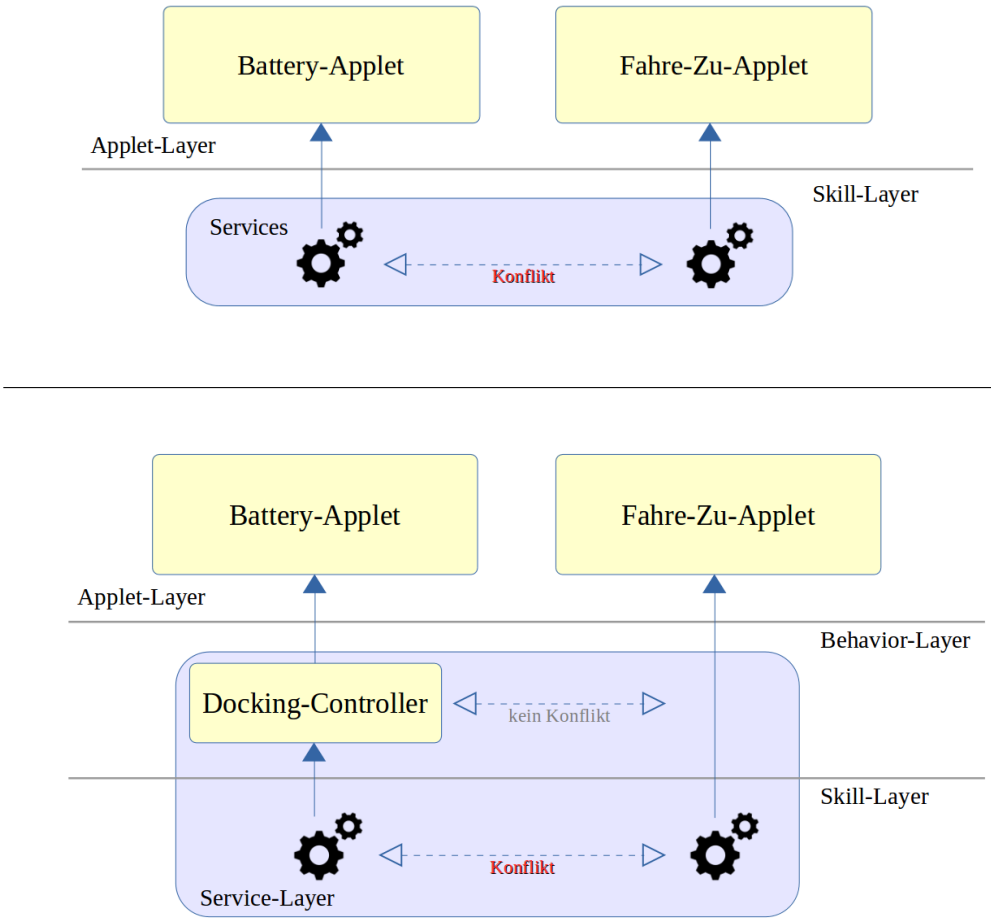


Abbildung 4.8: Verdeutlichung eines Konfliktes ersten bzw. zweiten Grades
oben: *Batterie- und Fahre-Zu-Applet benötigen beide exklusiven Zugriff auf die Motorsteuerung („Pilot") des Roboters (**Konflikt 1. Grades**);*
unten: *Batterie- und Fahre-Zu-Applet benötigen jeweils unterschiedliche Services, haben jedoch in den unteren Layern gemeinsame Abhängigkeiten (**Konflikt 2. Grades**). Die Einteilung in Layer erfolgt analog zu Abb. 4.3.*

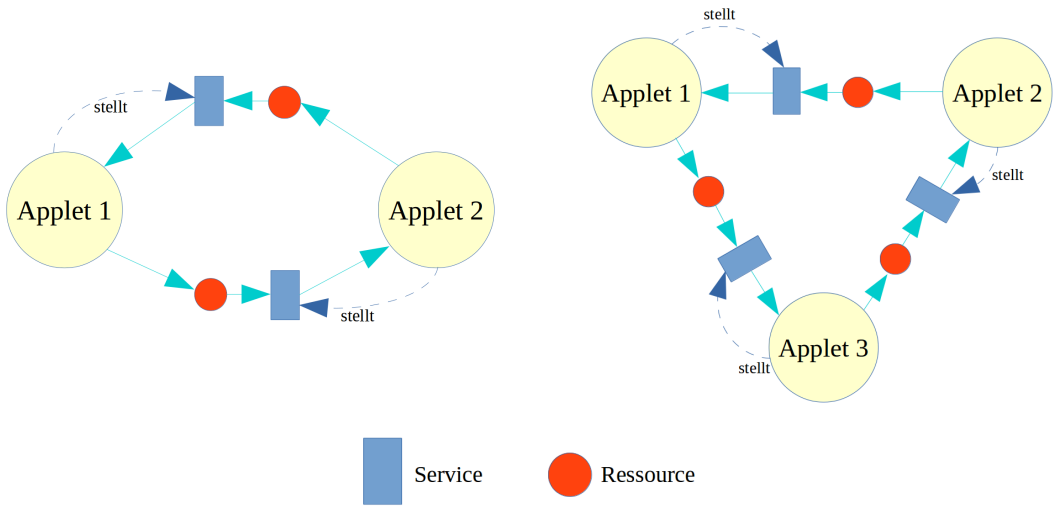


Abbildung 4.9: Darstellung einer zirkulären Abhängigkeit
Links zu sehen ist eine einfache zirkuläre Abhängigkeit, z. B. zur Kommunikation zweier Komponenten. Rechts zu sehen ist der Fall eines komplexeren Zyklus. In jedem Falle müssen entweder alle Ressourcenzugriffe gestattet, oder aber verboten werden, da alle anderen Fälle zu unerfüllten Abhängigkeiten führen.

in ausgesprochen spezifischen Problemfällen (siehe auch Kap. 4.5) wider. Ein wesentliches dieser charakteristischen Probleme ist der Aufbau zirkulärer Abhängigkeiten, auch wenn die funktionale Architektur zunächst einem reinen Top-Down-Ansatz zu folgen scheint. Wie bereits in Kapitel 4.2 genannt, ist es durchaus möglich, dass ein Applet einen Service zur Verfügung stellt. Ein anderes Applet hat somit also die Möglichkeit, diesen Service als Ressource zu beantragen und baut damit eine Abhängigkeit zum Services des erstgenannten Applets auf. Stellt das zweite Applet nun selbst einen Service, so gilt diese Abhängigkeit für ihn ebenso. Es ist jedoch durchaus möglich, dass ebendieses erstgenannte Applet nun den Service des zweiten Applets beantragt, wodurch im Sinne der Graphentheorie ein Zyklus entsteht (siehe Abb. 4.9). Natürlich lässt sich dieses Szenario auch um weitere Applets erweitern. Solche Abhängigkeitskreise gilt es zu erkennen, so dass gleichzeitig entweder alle Zugriffe auf Ressourcen innerhalb des Zyklus gestattet werden, oder aber keine. Heuristiken, um derartige Probleme zu vermeiden, sind in Kapitel 4.5 genauer beschrieben.

4.3 Konfigurierbarkeit und Integration

Im Rahmen dieser Arbeit erfolgte eine vollständige Implementierung des Konzeptes sowie aller darin vorgestellten Elemente. Auch wenn das Konzept an sich variabel einsetzbar ist, eröffnen sich durch die Benutzung eines Frameworks, welches bereits Mechanismen zur Interkomponentenkommunikation sowie Grundgerüste für den Komponentenaufbau bietet, viele Vorteile. Hierbei wurde sich für das MIRA-Framework entschieden, welches eine ähnliche Funktionalität und Erweiterbarkeit wie das *Robot Operating System* (**ROS**) anbietet. Die Basis stellen dabei so genannte Units. Diese sind gekapselte Komponenten, welche jeweils eine eigene Funktionalität implementieren und innerhalb ihrer jeweiligen Implementierung gegebenenfalls (aber nicht zwangsläufig) mit anderen Komponenten kommunizieren. Units stellen hier die Basis für den Aufbau von Applets. Eine Unit selbst ist dabei als *shared library* umgesetzt und kann zur Laufzeit dynamisch geladen und instantiiert werden [MetraLabs GmbH et al., 2012c]. Applets auf Basis von Units zu definieren ist insofern vorteilhaft, dass Units derzeit ohnehin die grundlegenden architektonischen Bauelemente für neue Anwendungen darstellen, ein Wechsel von der Implementierung von Units zu der Implementierung von Applets für einen Entwickler also nur eine geringe Einarbeitungszeit erfordert. Alle gängigen Konzepte zur Kommunikation zwischen Komponenten sowie zum Ansprechen von Hardware, Nutzung von verteilten Nachrichten, etc. lassen sich also wie zuvor nutzen.

Darüber hinaus erfolgte eine Integration zur Nutzung des XML-Serialisierungsmechanismus des MIRA-Frameworks, sowohl vom Scheduler, als auch von verwendeten Services und Applets (siehe auch [MetraLabs GmbH et al., 2012b]). Die Bekanntmachung sonst nicht vorhersehbarer Abhängigkeiten - beispielsweise, weil diese ausschließlich innerhalb der Implementierung sichtbar werden und als solche für den Scheduler nicht zu erkennen sind - ist über das XML-Format ebenfalls möglich. Weiterhin können XML-Dateien zur Konfiguration von Scheduler, Applets und Services genutzt sowie von anderen XML-Dateien inkludiert werden. Eine Beispiel-XML findet sich in Abb. 4.10.

Alle grundsätzlichen Komponenten lassen sich damit bereits direkt über das MIRA-

```xml
<unit id="SchedulerLoader" class="DynamicAppSchedulerLoader">

    <Scheduler class="PreemptiveAppletScheduler" />

    <ServiceDependencies>
        <item>
            <Service>/docking/DockingController</Service>
            <Dependencies>
                <item>/navigation/Pilot</item>
            </Dependencies>
        </item>
    </ServiceDependencies>

    <Applets>
        <item class="UIServiceApplet" AppletName="UIServiceApplet">
            <MenuAppletName>/Scheduler/MenuApplet</MenuAppletName>
            <MenuTriggerMethod>trigger</MenuTriggerMethod>
        </item>
        <item class="MenuApplet" AppletName="MenuApplet" />
        <item class="DateApplet" AppletName="DateApplet">
            <MenuApplet>/Scheduler/MenuApplet</MenuApplet>
            <PilotServiceName>/navigation/Pilot</PilotServiceName>
            <DeliverX>79.006</DeliverX>
            <DeliverY>-21.078</DeliverY>
            <DeliverPhi>44.614</DeliverPhi>
        </item>
        <item class="UIDriveToApplet" AppletName="DriveToApplet">
            <PilotServiceName>/navigation/Pilot</PilotServiceName>
            <MapChannel>/maps/localization/Map</MapChannel>
        </item>
        <item class="RobotBatteryApplet" AppletName="BatteryApplet">
            <BatteryServiceName>/BatteryService</BatteryServiceName>
            <DockingControllerServiceName>/docking/DockingController</DockingControllerServiceName>
            <DockingStateChannel>/docking/State</DockingStateChannel>
        </item>
        <item class="MonitorRequireApplet" AppletName="MonitorRequireApplet" />
    </Applets>
    <Services>
        <item class="MonitorService">
            <Width>700</Width>
            <Height>1100</Height>
        </item>
    </Services>
</unit>
```

Abbildung 4.10: Beispiel-XML-Datei zum Laden einer Applet-Grundkonfiguration

Wie in der Grafik zu sehen, ist die Bekanntmachung von zu verwendendem Scheduler, benutzten Applets, Services, deren jeweiliger Konfiguration sowie evtl. vorhandene unsichtbare Abhängigkeiten über die Integration in das MIRA-Serialisierungsframework möglich. Ebenfalls ist eine Umbenennung der Komponenten sowie das Setzen derer Eigenschaften möglich. Bei dem dargestellten Beispiel handelt es sich um das Realweltszenario aus Kapitel 5.3.

Framework instantiieren, laden, konfigurieren und ansprechen. Da es sich bei dem Konzept der Ablaufsteuerung jedoch um ein verhältnismäßig komplexes handelt, ist es überdies sinnvoll, eine geeignete Visualisierung zur Vermeidung und Analyse von Fehlern, sowohl im Scheduling-Prozess, als auch im Aufbau von Applets, zu implementieren. Auf Basis von AT&T's Open-Source-Toolkit **Graphviz** (siehe [Bilgin et al., 2012]) wurde deshalb versucht, eine geeignete Visualisierung zu finden, welche ebendiese Probleme adressiert. Abb. 4.11 zeigt die Visualisierung eines einfachen Anwendungsfalls. Die Integration der Visualisierung in MIRA erfolgte auf Basis von MIRAs Rich Client Platform (**RCP**, siehe [MetraLabs GmbH et al., 2012a]).

4.4 Zusätzliche Problemstellung durch Dynamik

Sowohl Scheduler als auch Applets werden jeweils in einem separaten Thread ausgeführt. Dies stellt eine einfache Variante dar, die Entwicklung eigenen Codes vollständig von der Ausführung des Schedulers zu entkoppeln. Weiterhin ist dies eine elegante Lösung um ein Blockieren des Scheduler aufgrund langer Rechenzeit der Applets zu vermeiden. Ebenso wird hierdurch die Fehlertoleranz der entstehenden Ablaufsteuerung erhöht. Hinzu kommt, dass auch das MIRA-Framework und im Normalfall[4] wiederum jede darin enthaltene Unit in einem eigenen Thread ausgeführt wird. Da aus MIRA-Units direkt für den Scheduler und die von ihm verwalteten Apps nutzbare Services erstellt werden können (siehe Kap. 4.3), betrachten wir sie im Folgenden analog zu lokalen Services.

4.4.1 Verklemmungen und Verklemmungsvermeidung

Sowohl jedes Applet, der Scheduler selbst, sowie ein Großteil aller Services (da als MIRA-Unit implementiert, siehe [4]) werden in jeweils einem eigenen Thread ausgeführt. Die gegenseitige Synchronisation stellt deshalb sowohl auf Funktions- als auch auf

[4] Es ist de facto möglich, die Ausführung von MIRA-Units einem bestimmten Thread zuzuweisen. Eine Unit kann unabhängig davon auch mehrere Threads, z. B. für die Ausführung von Remote-Procedure-Calls, zugewiesen bekommen.

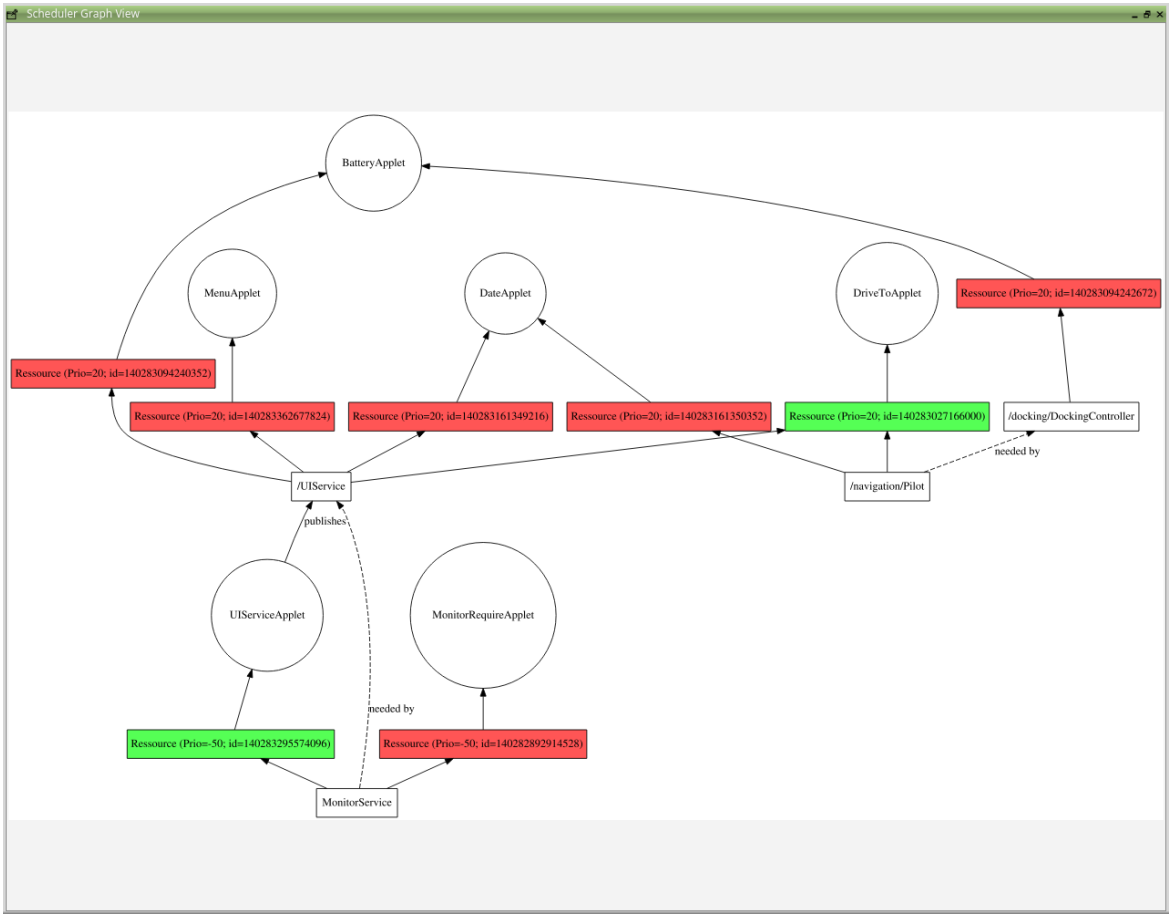


Abbildung 4.11: Graphviz-Visualisierung bekannter Applets, Services, Ressourcen
Dieser Screenshot zeigt die auf Graphviz basierende Visualisierung, welche eine Übersicht über alle zur Verfügung stehenden Applets, Services und Ressourcen bietet und dabei die jeweiligen Zustände, Prioritäten, etc., sowie alle Abhängigkeiten darstellt. Auch hier handelt es sich um das Realweltszenario aus Kapitel 5.3.

Algorithmus

Initialisierung: //

```
1       Registriere Applet bei Scheduler;
2       Beantrage eventuell benötigte Ressourcen;
3       Stelle ggf. einen oder mehrere Services;          // sofern Service-Applet
```

Ausführung: //

```
4       Führe alle Interrupt-Handler in Warteschlange aus;   // Neuerteilung,
Zuweisung, Wegnahme von Ressourcen
5       ...;                                                // implementierungsabhängig
6       Warte auf Zeitpunkt nächster Ausführung;
7       Gehe zu Ausführung;
```

Abbildung 4.12: Grundaufbau eines Applets

Der hier dargestellte Algorithmus zeigt das Grundprinzip des Ablaufs eines Applets. Hervorstechend ist die Ausführung von Interrupt-Händlern vor der eigentlichen Funktionalität des Applet-Main-Loops.

Interaktionsebene eine große Herausforderung dar. Da die funktionale Synchronisation wesentlich von der jeweiligen Implementierung abhängt, wird hier besonders auf die Synchronisation der Komponenteninteraktionen eingegangen. Der Grundaufbau jedes Applets entspricht dabei den Schritten des Algorithmus aus Abb. 4.12.

Wie darin zu sehen ist, existiert in jedem Applet eine Warteschlange für zusätzliche *Handler*. Ein Handler ist dabei ein Callback, das vom Scheduler oder einer seiner Unterkomponenten in die Warteschlange der Applets eingefügt wird. Die Implementierung eines Applets verkompliziert sich ungemein, sofern davon ausgegangen werden muss, dass dem Applet jederzeit seine zugeteilten Ressourcen entzogen werden können. Jederzeit bedeutet dabei, dass ein Entzug ohne derartige Synchronisationsmechanismen auch *zwischen* zwei Anweisungen innerhalb des eigentlichen Applet-Main-Loops („Ausführung") oder sogar während der Ausführung einer Anweisung erfolgen kann. Um dies zu vermeiden, wurde das Konzept des *Service-Reassigners* eingeführt.

Ein Service-Reassigner wird immer dann benötigt, wenn ein Service einem anderen Applet zugewiesen werden soll. Dies ist normalerweise nach jeder Berechnung eines

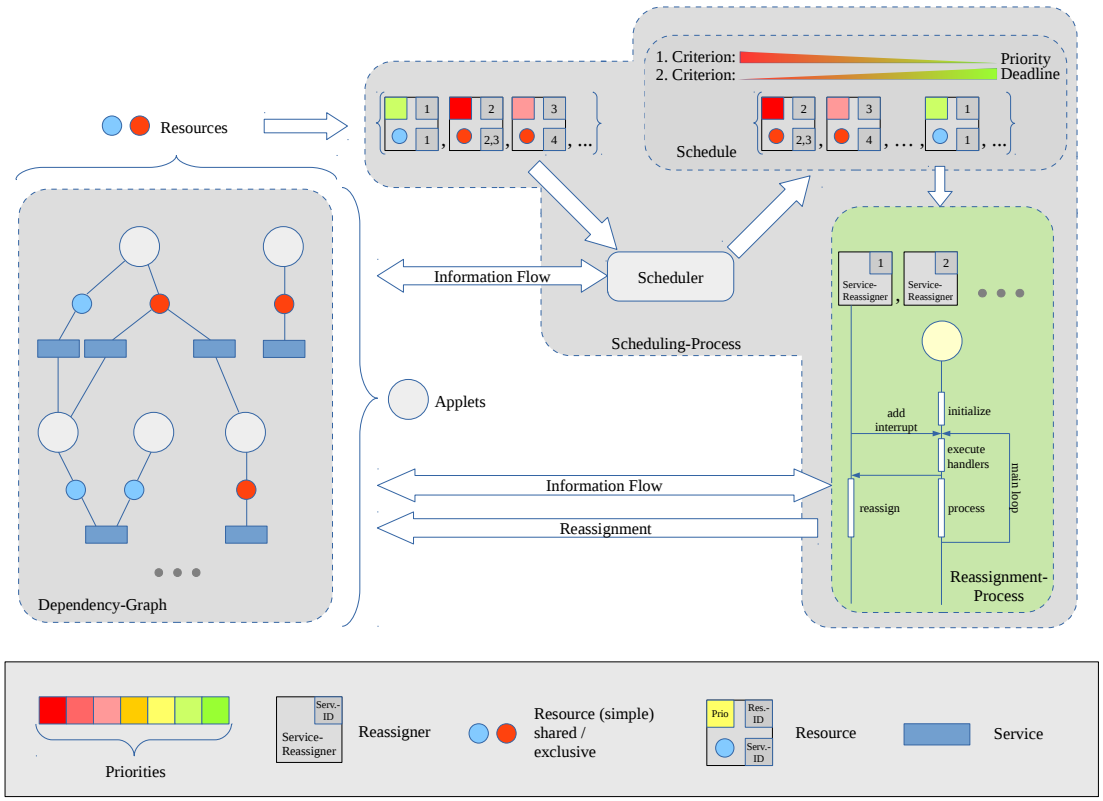


Abbildung 4.13: Einordnung des Service-Reassigners in das Gesamtkonzept
Service-Reassigner stellen Unterkomponenten des Schedulers dar und sind direkt im Scheduling-Prozess eingebunden.

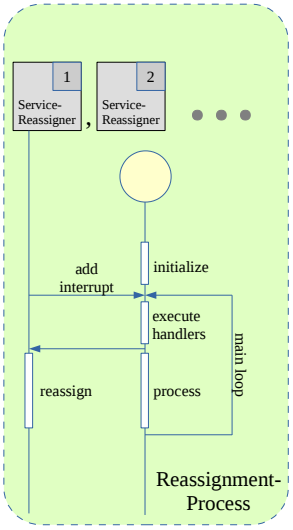


Abbildung 4.14: Einfügen eines Interrupt-Handlers in den Ablauf eines Applets durch Service-Reassigner

Ressourcenplans der Fall. Der Scheduler erstellt am Ende der Berechnung eines Ressourcenplans für jeden Service je einen Service-Reassigner. Der Service-Reassigner ist Teil des Scheduling-Prozesses und stellt selbst eine Unterkomponente des Schedulers dar. Seine individuelle Umsetzung ist jedoch stark implementierungsabhängig. Bei der Implementierung eines eigenen Scheduling-Algorithmus innerhalb des Frameworks zur dynamischen Ablaufsteuerung kann er jederzeit ausgetauscht werden, weshalb hier nur kurz darauf eingegangen wird. Der Service-Reassigner dient als Synchronisationsmechanismus zwischen den einzelnen Applets. Er realisiert folgende Funktionen:

- Applets, die den Service halten, über den geplanten Entzug informieren (Interrupt, siehe Kap. 4.2.1)
- Verhandlungsinformationen an den Scheduler weiterreichen, wenn nötig (siehe Kap. 4.5.3)
- Warten, bis alle Applets die jeweilige Ressource freigeben
- Zwangsentzug der Ressource, wenn nötig (siehe Kap. 4.6.1)
- Neuzuteilen der Ressource
- Informieren der Applets über Fortsetzung

Applets können von ihm entweder über die Zuteilung oder aber über den Entzug einer Ressource informiert werden. Auf beide Varianten können sie entsprechend reagieren, beispielsweise, indem sie eine Verhandlungsrunde (siehe Kap. 4.5.3) starten. Eine solche Möglichkeit wird den Applets über das Ausführen eines Interrupts eingeräumt. Dieser Interrupt wird dafür in die Warteschlange des Applets eingefügt und vor dem nächsten Main-Loop ausgeführt (siehe Abb. 4.14). Ein Interrupt kann also nur an fest definierten Stellen des Codes in Erscheinung treten. Dies ist insofern ein ausgesprochen großer Vorteil, als ein Implementierender sich somit also nicht um die Neuzuteilung von Ressourcen innerhalb des Main-Loops sorgen muss [5]. Sollte die Abarbeitung des

[5] Es ist, sofern der Entwickler dies festlegt, innerhalb des Applets durchaus möglich, ein hiervon abweichendes Verhalten zu definieren.

Main-Loops länger als eine festgelegte maximale Wartezeit auf eine Ressource (aktuell 5 Sekunden) dauern, wodurch der Interrupt nicht rechtzeitig ausgeführt werden kann, so wird die Ressource dem Applet schließlich zwangsentzogen (siehe Kap. 4.6.1). Auf diesen Fall sollte bei der Implementierung Rücksicht genommen werden.

4.4.2 Prioritätsinversion

Im Sinne des Schedulings in Betriebssystemen handelt es sich bei der Prioritätsinversion um das Phänomen, dass bei entsprechenden Belegungen von (System-)Ressourcen ein mittelpriorisierter Task gegenüber einem höher priorisierten Task bevorzugt werden kann. Hierzu erfordert es mindestens drei Tasks jeweils unterschiedlicher Prioritäten. Dem am niedrigsten priorisierten Task ist dabei eine Ressource zugewiesen, auf die der höchstpriorisierte Task Zugriff verlangt. Ein mittelpriorisierter Task benötigt diese Ressource jedoch nicht, wodurch er den niedrigpriorisierten vom Prozessor verdrängen kann. Der niedrigpriorisierte Task kann somit die Ressource nicht wieder freigeben, womit der höchstpriorisierte Task ausgesperrt wird. Die Prioritäten von mittel- und hochpriorisiertem Task werden somit invertiert.

Dieses Problem ist aufgrund der vollständigen Exklusivität von Zugriffen auf den Prozessor im Betriebssystemumfeld natürlich nur teilweise übertragbar, findet sein Adäquat in der dynamischen Ablaufsteuerung jedoch in einem ähnlichen Kontext. Bedingt wird dies durch den hierarchischen Aufbau der Service-Definitionen (siehe Kap. 4.2.3). Beantragt ein Applet mit höchster Priorität Zugriff auf den Services eines Service-Applets, so kann es durchaus sein, dass ebendieses Service-Applet wiederum selbst auf einem Service aufbaut. Wird dieser zweite Service nun jedoch von einer Applet-Ressource mittlerer Priorität belegt, so würde dieser stets der Vorzug auf den zweiten Service gegeben. Das Applet mit der Ressource mittlerer Priorität blockiert also die niedrigpriorisierte Ressource des Service-Applets. Hieraus folgt, dass ebenfalls die höchstpriorisierte Ressource des ersten Applets blockiert wird, es werden also auch hier die Prioritäten der beiden Ressourcen invertiert (siehe Abb. 4.15).

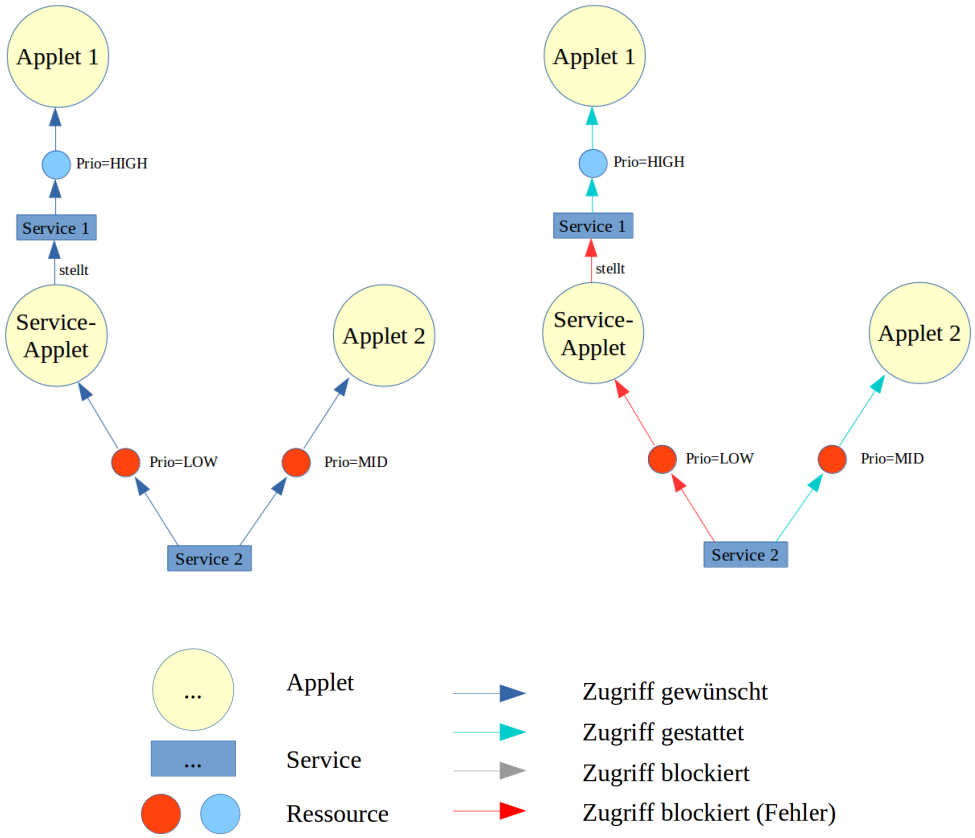


Abbildung 4.15: Prioritätsinversion durch hierarchische Service-Abhängigkeiten

Dargestellt ist die fehlerhafte Zuweisung von Service 2 an Applet 2. Die eigentlich höher priorisierte Ressource von Applet 1 wird so am Zugriff auf das Service-Applet gehindert.

Prioritätsvererbung

Ein gängiges Konzept zur Vermeidung der Prioritätsinversion, welches ebenfalls aus der Betriebssystemumgebung stammt, besteht im Vererben von Prioritäten [SHA et al., 1990]. Dies erfolgt, indem dem niedrig priorisierten Task, welcher dort die Ressource des höher priorisierten Tasks belegt, bis zur vollständigen Abarbeitung und Freigabe der Ressource vorübergehend die Priorität ebendieses höher priorisierten Tasks eingeräumt bekommt. Hierdurch wird ein Verdrängen durch einen mittelpriorisierten Task verhindert.
Auch dieses Prinzip lässt sich unmittelbar übertragen. Erfordert ein Applet höchster Priorität Zugriff auf die Ressource des Services eines Service-Applets niedrigerer Priorität, so erbt dieses Service-Applet wiederum für diejenigen Ressourcen, die für seine Funktionalität notwendig sind, ebenfalls die Priorität des höchstpriorisierten Applets. Hierdurch wird das Applet mittlerer Priorität daran gehindert, das Service-Applet niedriger Priorität auszuschließen und so die Funktionalität des höchstpriorisierten Applets zu beeinflussen. Abb. 4.16 verdeutlicht diesen Sachverhalt noch einmal. Eine interessante Eigenschaft der Prioritätsvererbung ist, sich ohne Weiteres mit Greedy-Algorithmen kombinieren zu lassen. Diese Gegebenheit wird sich unter Kapitel 4.5.1 sowie im Algorithmus 4.19 zunutze gemacht.

4.5 Verwendete Heuristiken

Die Optimalität der hier vorgestellten Methoden oder der durch sie berechneten Zeitpläne stellt für das resultierende Gesamtsystem einen enormen Mehrwert dar. Insbesondere wichtig wird dies durch die konkurrierenden, teils vollständig ausdefinierten Ansätze aus Kapitel 3, wie dem der Single-State-Machine (siehe Kap. 3.2.2). Tatsächlich beschränkt sich deren Verifizierbarkeit jedoch ebenfalls auf das sie definierende formale System. Die Einführung von (beliebigen) Interpretierfunktionen für Zustandseintritt, -austritt oder -übergang führt auch bei Ihnen zum Verlust der Vorhersagbarkeit und somit formaler Verifizierbarkeit. Deshalb an dieser Stelle ein kleiner Ausflug in die Algorithmentheorie:

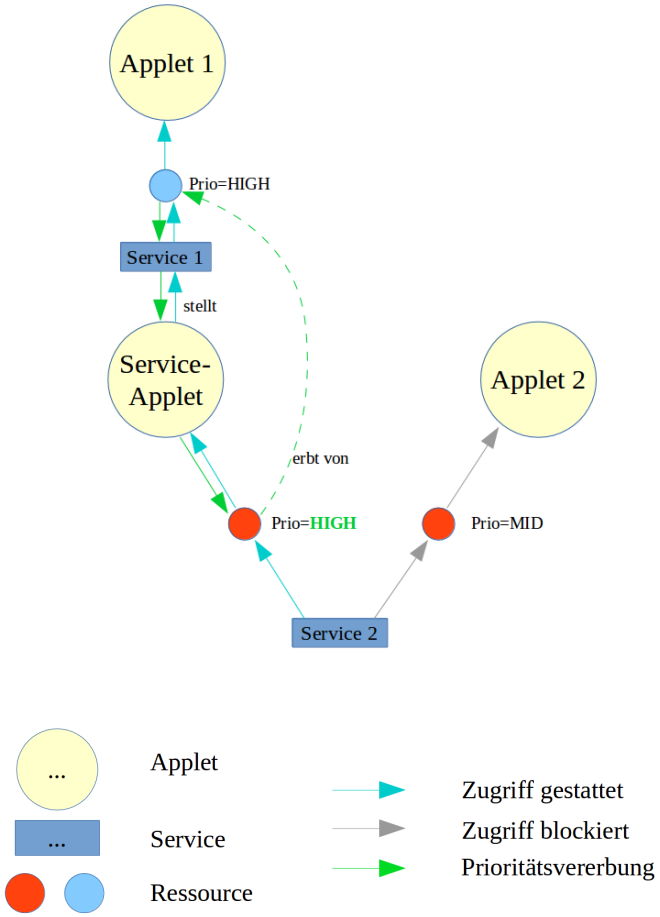


Abbildung 4.16: Lösung der Prioritätsinversion durch Prioritätsvererbung

Durch Vererbung der Ressourcenpriorität von Applet 1 wird die Funktionalität des Service-Applets sichergestellt und alle Ressourcen entsprechend ihrer Prioritäten freigeschaltet.

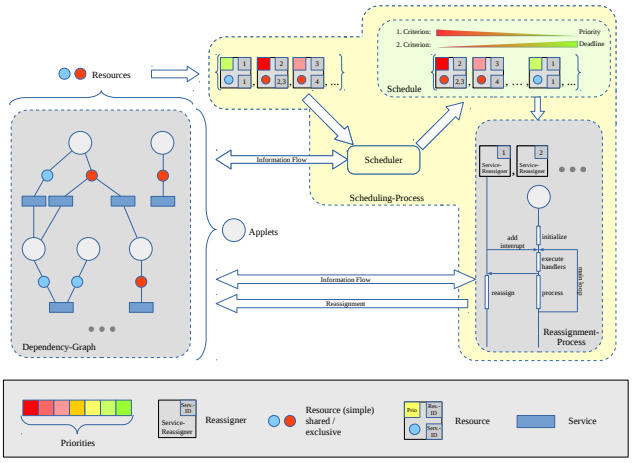


Abbildung 4.17: Einordnung des Schedulings auf Basis von Heuristiken
Der Scheduler übernimmt im Wesentlichen die Aufgabe des Sortierens der Zugriffe. Die beiden relevanten Ordnungskriterien sind dabei Priorität und eventuelle Deadlines.

Das Halteproblem, ein Begriff der theoretischen Informatik, sagt aus, dass ein Programm im Allgemeinen nicht die Terminierung eines anderen Programms vorhersagen kann. Die Gödel'schen Unvollständigkeitssätze, eigentlich der modernen Logik entsprungen, besagen weiterhin, dass ein hinreichend komplexes, widerspruchsfreies System:

- ... stets unbeweisbare Aussagen beinhaltet, und
- ... seine eigene Widerspruchsfreiheit nicht selbst beweisen kann

Aus jedem der beiden Probleme lässt sich unmittelbar das Korrektheitsproblem schlussfolgern[6], welches wiederum besagt, dass ein Algorithmus niemals die Korrektheit eines anderen Programms vorhersagen kann. Genau wie die Zustandseingans-, Zustandsausgangs- und Zustandsübergangsfunktionen des State-Machine-Konzeptes unterliegen auch Applets nach dem hier vorgestellten Konzept formal keinen algorith-

[6]Dieses lässt sich sogar auf das Halteproblem reduzieren und umgekehrt. Beide Probleme sind somit äquivalent. Der Gödel'sche Unvollständigkeitssatz stellt eine Verallgemeinerung dar.

mischen Einschränkungen und fallen somit unter diese Begrifflichkeit. Da ein Applet somit einen unendlichen Zustandsraum aufweisen kann (zumindest theoretisch, siehe [8]) und es ebenfalls weder zeitliche, prioritäre noch mengenmäßige Beschränkung für die Beantragung von Ressourcen oder den Umgang mit diesen einzuhalten hat, ist das zukünftige Zeitverhalten eines Applets weder durch die Vergangenheit noch durch Kenntnis über verwendete Ressourcen im Allgemeinen bestimmbar[7][8].
Da also selbst im allgemeinen Fall keine Lösung für die oben genannte Problematik existiert (und auch nicht existieren kann), folgt für den hier vorgestellten Scheduler, dass es ebenfalls nicht möglich ist, einen optimalen Zeitplan für zukünftige Ressourcenzugriffe zu berechnen. Es ist somit nützlich für das Gesamtverhalten, einige zusätzliche Einschränkungen einzuführen und sich die im Folgenden vorgestellten Heuristiken zunutze zu machen. Wie bereits in Kapitel 4.1.2 kurz erläutert, wurden hierfür als zusätzliche Unterkonzepte eingeführt:

- Deadlines
- Dynamische Prioritäten und Triggering
- Callbacks, Interrupts und Verhandlungsrunden im Falle eines Ressourcenentzugs

Zusätzlich sei angemerkt, dass Ressourcenplanungen in der aktuellen Implementierung **nicht zeitgesteuert**, sondern ausschließlich zeitlokal (wenn auch mehr oder weniger periodisch) erfolgen. Zeitbehaftete Eigenschaften von Ressourcen wie Deadlines werden dabei berücksichtigt. Eine Neuberechnung erfolgt mit jeder Ressourcenbeantragung oder Prioritäts-/Deadline-Modifikation. Aufgrund dieser zeitvarianten Dynamik könnte man die entstehende Abfolge von Zugriffen als Zeitplan bezeichnen. Ein ähnlicher Ansatz wird beim Betriebssystemscheduling verwendet. In Kapitel 4.6.2 wird auf diesen Umstand noch einmal genauer eingegangen.

[7] Unter bestimmten Einschränkungen der Bedingungen für Ankunftszeiten, beantragbare Ressourcen oder maximale Dauer eines Ressourcenzugriffes lässt sich das Problem auch algorithmisch lösen.

[8] Tatsächlich lässt sich aufgrund der Beschränktheit der verwendeten Hardware in diesem Fall stets ein zukünftiges Verhalten berechnen - die Anzahl voneinander verschiedener Konfigurationen ist aufgrund des begrenzten Speichers stets endlich. Die Lösung lässt sich somit durch Durchprobieren aller Konfigurationen berechnen, ist jedoch offensichtlich nicht praktikabel.

4.5.1 Earliest Deadline First

Bei der Beantragung einer Ressource ist es möglich, dem Scheduler eine Frist zur Zuweisung zu setzen, also einen Zeitpunkt zu definieren, ab dem eine Ressource spätestens verfügbar sein muss, um die Funktionalität des beantragenden Applets zu gewährleisten. Ist diese bereits zugewiesen, so kann ein Applet auch die Zusicherung der Ressource zu einem definierten, späteren Zeitpunkt verlangen.
Der Earliest-Deadline-First-Algorithmus (kurz **EDF**; auch: *least time to go*) gehört zur Gruppe der dynamischen Scheduling-Algorithmen. Er existiert sowohl in einer präemptiven, als auch in einer nicht-präemptiven Variante. EDF-Scheduling folgt, wie der Name bereits verrät, einem ausgesprochen einfachen Grundprinzip: Es wird stets der Job abgearbeitet, der die kürzeste Deadline aufweist. Vorausgesetzt, es handelt sich um eine **prioritätsfreie** Ausführung von Jobs auf einem präemptiven **Einprozessorsystem**, so bietet das insbesondere in Echtzeitbetriebssystemen benutzte EDF-Scheduling einen großen Vorteil: Theoretische Informatik und Betriebssystemtheorie konnten formal beweisen, dass EDF unter den oben genannten Bedingungen stets einen optimalen Zeitplan liefert, sofern ein solcher existiert. Weder handelt es sich bei dem hier vorgestellten Konzept jedoch um ein ausschließlich auf einer Ressource operierenden, noch um einen prioritätsfreien Algorithmus. Tatsächlich stellen Prioritäten in der dynamischen Ablaufsteuerung sogar eines der wesentlichsten Kriterien dar. Deadlines sind insofern eher zweitrangig, als sie in robotischen Anwendungen auf Anwendungslevel, aufbauend auf einem nicht-echtzeitfähigen Betriebssystem, eher sparsam eingesetzt werden sollten (siehe Kap. 4.6). Bedeutung bekommen Deadlines in der robotischen Anwendung immer dann, wenn mit Ablaufen der Deadline:

- ... Applets eine erhebliche Beeinträchtigung ihrer Funktionalität droht,
- ... diese Beeinträchtigung das Gesamtverhalten des Systems beeinflussen kann, oder
- ... eine Beschädigung von Hardware-Komponenten droht[9].

[9] Da das Verhalten der Ablaufsteuerung aufgrund der Systemdynamik nicht vollständig prädiziert werden kann, sollte eine Vermeidung der Beschädigung von Hardware vorzugsweise auf einem niedrige-

Trotzdem findet der EDF-Algorithmus einen sinnvollen Anwendungszweck im hier vorgestellten Szenario. Zunächst sei dafür als Prämisse gegeben, dass Prioritäten stets höher gewertet werden sollen als Deadlines (siehe Kap. 4.5.2). Sie stellen somit das primäre Ordnungskriterium bei der (endgültigen) Priorisierung von Ressourcenzugriffen dar. Diese Simplifikation erlaubt es, die Gesamtmenge aller Eigenschaftstupel $e_r = (p_r, d_r)$ von Ressourcen $r \in R$ mit Priorität $p_r \in P \subset \mathcal{R}$ und Deadline $d_r = (t_{early,r}, t_{duration,r}, t_{finish,r}) \in D$ mit $t_{early,r}$ = frühestem Startzeitpunkt, $t_{duration,r}$ = Dauer und $t_{finish,r}$ = spätestem Endzeitpunkt eines Zugriffes auf Ressource r derart zu sortieren, dass gilt:

$$\begin{aligned} \forall e_i, e_j \in E = \{(p_x, d_x)\} : \quad & i \leq j \equiv p_i \leq p_j \\ 1 \leq x, i, j \leq |R| & \\ x, i, j \in \mathcal{N}_+ & \end{aligned} \tag{4.1}$$

Diese Menge ist nun ohne weiteres in eine Partition überführbar:

$$E_P = \{U_1, U_2, ..., U_{|P|}\} \tag{4.2}$$

$$E = \bigcup_{1 \leq k \leq |P|} U_k \tag{4.3}$$

$$\forall k, l \in \{1, ..., |P|\} : U_k \cap U_l = \emptyset \tag{4.4}$$

Alle Untermengen U_k mit $1 \leq k \leq |P|$ enthalten jeweils ausschließlich gleichprioritäre Eigenschaftstupel, wobei ein höheres k einer höheren Priorität entspricht:

$$\forall k \in \{1, ..., |P|\}, \forall i, j \in \{1, ..., |R|\} : i \neq j : \qquad e_i, e_j \in U_k \equiv p_i = p_j \tag{4.5}$$

$$\forall i, j \in \{1, ..., |R|\}, \forall x, y \in \{1, ..., |P|\} : x < y : \qquad e_i \in U_x \wedge e_j \in U_y \equiv p_i < p_j \tag{4.6}$$

Innerhalb jeweils einer dieser Partitionen ist es ohne Weiteres möglich, das Problem als ein prioritätsfreies zu betrachten. Weiterhin sind alle Abhängigkeiten zwischen

rem Systemlevel als dem der Ablaufsteuerung (bedeutet: Behavior-, Skill- oder sogar Hardware-Layer) stattfinden (siehe auch Kap. 4.4 sowie Kap. 4.5)

Services und Applets ausschließlich unidirektional. Dies erlaubt zwar durchaus Zyklen, welche normalerweise für die folgende Betrachtung störend wären, kann aber umgangen werden, indem als Einschränkung eingeführt wird, dass Ressourcenzugriffe innerhalb von Zyklen stets vollständig gesperrt oder aber gestattet werden. Maßgeblich ist hierbei jeweils das Zykluselement mit der höchsten Priorität, diese wird auf die darunterliegenden Stufen propagiert (siehe Kap. 4.4.2). Zyklen lassen sich somit auf jeweils ein einziges Element reduzieren, welches die Ressourcenzugriffe aller innerhalb des Zyklus verbundenen Applets und Services repräsentiert [10]. Graphentheoretisch betrachtet konvergiert jeder einzelne Teilgraph aller Ressourcenzugriffe somit zu einem gewurzelten Baum, wobei die Wurzel des Baumes stets das Element mit der höchsten im Teilgraphen existierende Priorität ist. Es gilt die früheste *relative Deadline* des gesamten Teilgraphen. Diese entspricht dem spätesten Startzeitpunkt eines Zugriffes mit Deadline, berechnet sich für Ressource r also zu:

$$t_{rel,r} = t_{finish,r} - t_{duration,r} \tag{4.7}$$

Hierdurch ist es möglich, die Ressourcen als frei von Abhängigkeiten zu betrachten, womit - innerhalb der hier beschlossenen Einschränkungen - EDF als Scheduling-Algorithmus für optimale Zeitpläne eingesetzt werden kann.
Einen guter Einsatzzweck für Deadlines und damit das Scheduling durch EDF stellt das Batterie-Lade-Applet aus Kapitel 4.2.3 dar. Sollte es einen Abfall des Ladezustandes feststellen, kann es recht frühzeitig erste Prognosen darüber liefern, wann die Batterie geladen werden muss. Dabei kann es zum einen abschätzen, wann die Batterie vollständig entladen sein wird ($t_{finish,r}$), zum anderen aber auch wie lange ein Anfahren der Ladestation dauern wird ($t_{duration,r}$). Der späteste Zeitpunkt zum Anfahren der Ladestation, um sicherzustellen, dass diese also noch erreichbar ist, berechnet sich somit nach Formel 4.7, wobei in der Praxis ein (möglichst großzügiger) Puffer eingeplant werden sollte.

[10] In Kapitel 4.2.3 wird hierauf genauer eingegangen.

4.5.2 Prioritätengesteuertes Scheduling

Wie bereits in den vorherigen Kapiteln erläutert, stellen Prioritäten das wesentlichste aller Kriterien für die Steuerung der Reihenfolge von Ressourcenzugriffen dar. Einem Applet steht es frei, für seine Ressourcen selbst Prioritäten festzulegen und diese sogar zur Laufzeit zu verändern. Drei hierfür verbreitete Ansätze sind folgende:

- Festlegen einer statisch definierten Anzahl und Wertigkeit von Prioritäten
- Erlauben beliebiger Prioritäten (z. B. reel)
- Vordefinieren von Prioritätsstufen mit gleichzeitiger Gestattung von Zwischenstufen

Jedes dieser Prinzipien hat dabei seine eigenen Vor- und Nachteile. Während die Verwendung statisch definierter Prioritäten zwar den Vorteil eines begrenzten Wertebereichs hat, somit also bereits im Vorhinein abschätzbar ist, welche Prioritäten auftreten können und wie diese zu berücksichtigen sind, ist die Granularität häufig nicht ausreichend, um komplexen Systemen hinreichend feine Abstufungen zu ermöglichen. Das Erlauben beliebiger Prioritäten löst dieses Problem zwar, kann aber dagegen zu Prioritätskriegen führen, also die Überpriorisierung neu entwickelter Applets zuungunsten bereits vorhandener fördern, welche später ihrerseits wieder von neuen Applets depriviligiert werden können. Bei Verwendung beliebiger Prioritäten ohne entsprechenden Referenzrahmen muss deshalb stets die Priorisierung vorhandener Applets mitberücksichtigt werden, was aber dem Ziel erhöhter Modularität zuwiderläuft. Zwar ergibt sich die gleiche Problemstellung ebenfalls bei der Vordefinition von Prioritätsstufen mit Erlauben von Zwischenstufen, durch die Verwendung eines Triggering-Mechanismus kann jedoch die Anzahl der tatsächlich verwendeten Prioritäten drastisch reduziert werden, weshalb dies die hier favorisierte Lösung darstellt. Ein Triggering-Mechanismus erhöht dabei temporär die Priorität eines Ressourcenzugriffes (Überpriorisierung) um einen Wechsel der Ressourcenzuweisung zu bezwecken. Ist dieser erfolgt, wird die ursprüngliche Priorität wiederhergestellt. Da Ressourcenzuweisungen ohne Veränderung von Prioritäten oder Deadlines als statisch gelten (siehe Kap. 4.5.3), erzeugt dies einen

stabilen Wechsel ohne oszillierendes Verhalten[11].

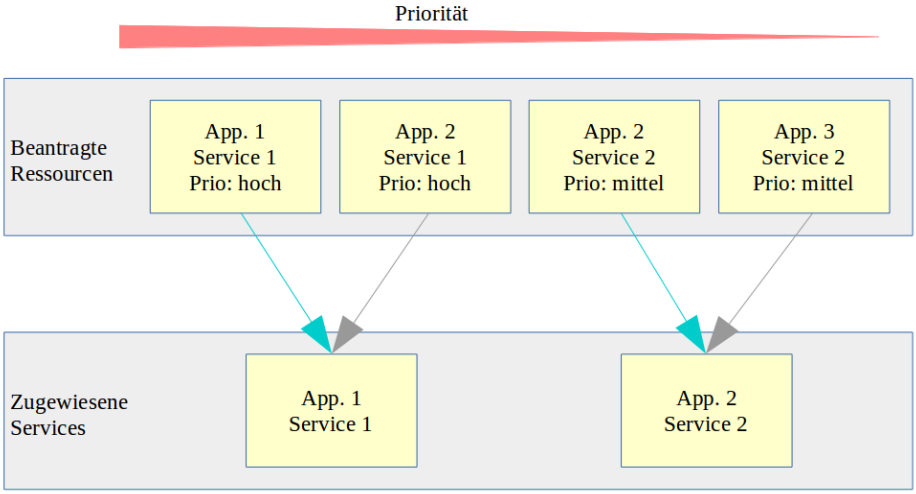
Ein sehr ähnlicher Ansatz findet sich bereits in Kap. 3.3.3. Diesen Triggering-Mechanismus nun mit vordefinierten Prioritätsstufen zu kombinieren, gestattet es dem Entwickler, auf Basis fester Prioritäten ein dynamisches Systemverhalten auch zwischen gleichpriorisierten Applets zu erzeugen, Prioritätsstufen jedoch zur Priviligierung wichtigerer Service-Applets nutzen zu können. Vorzugsweise dienen sich Applets dabei gegenseitig als Trigger (wie z. B. bei User-Interfaces), sie können jedoch unter definierten Umständen auch selbstständig aktiv werden. Das oben vorgestellte Batterie-Lade-Applet beispielsweise würde bei Unterschreiten einer festgelegten Schwelle für den Batteriezustand den Nutzer über einen anstehenden Ladevorgang informieren, würde dazu also die Ressource zum Anzeigen ebendieser Information triggern. Das Risiko, bereits vorhandene Applets dauerhaft zu überblenden (und damit aushungern zu lassen), wird deutlich reduziert.

Die Verwendung eines Priorisierungsansatzes im Allgemeinen leitet sich aus den Anforderungen an das Gesamtsystem ab. Aufgrund der einem robotischen System innewohnenden Autonomie stellt Robustheit eines der wesentlichsten Kriterien dar. Wäre es einem Applet durch die Setzung von Deadlines also implizit möglich, eine höhere Priorisierung als die der sonst deutlich wichtigeren - also auch höher priorisierten - Services oder Service-Applets zu erreichen, kann dies nicht nur Auswirkungen auf Application-, Applet- und Scheduler-Layer (siehe Kap. 4.2), sondern durch Fehlansteuerung der verwendeten Komponenten auch auf Behavior- und Skilllayer sowie das darunterliegende MIRA-Framework haben.

4.5.3 Hybrides Scheduling

Aus den beiden vorgestellten Prinzipien lässt sich ein einfacher, performanter Algorithmus ableiten, der einen Zeitplan erstellt, welcher:

[11] Ein oszillierendes Verhalten ist erreichbar, sofern eine Ressourcen haltende App auf einen Entzug dieser mit einem Triggering ebenjener Ressourcen reagiert. Dies gilt es zu vermeiden. Generell sollte auf einen Entzug hin niemals eine Neupriorisierung erfolgen. Auf die Systemsicherheit wird noch einmal genauer in Kapitel 4.6 eingegangen.



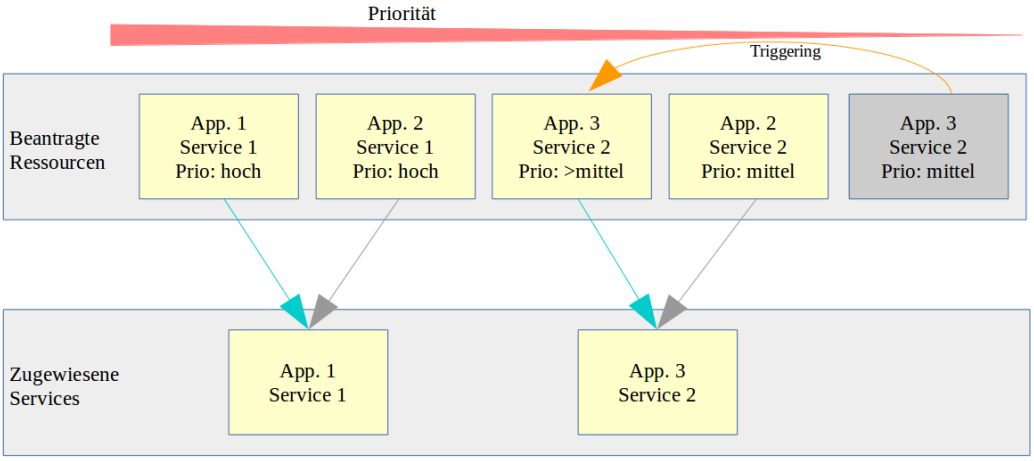


Abbildung 4.18: Reihenfolge der Zuweisung mit und ohne Triggering

Dargestellt ist prioritätengesteuerte Zuweisung von Ressourcen an Applets, sowie der Triggering-Mechanismus. Türkise Pfeile entsprechen zugewiesenen, ausgegraute entsprechen blockierten Ressourcen. Da eine stabile Sortierung genutzt wird, erzeugt der Triggering-Mechanismus kein oszillierendes Verhalten (siehe auch Kap. 4.5.3).

.

- ... exklusive Zugriffe gestattet
- ... Service-Applets berücksichtigt (Kap. 4.2)
- ... Mehrfachressourcen (Resource-Bundles) berücksichtigt (Kap. 4.2.2)
- ... hierarchische Service-Definitionen gestattet (Kap. 4.2.3)
- ... Deadlines innerhalb einer Prioritätsstufe einhält (Kap. 4.5.1)

  ... und hierfür den EDF-Algorithmus implementiert
- ... die Ressourcen-Zuweisung auf Basis von Prioritäten vornimmt (Kap. 4.5.2)

  ... den Applets dabei ein Ressourcen-Triggering gestattet

  ... oszillierendes Verhalten verhindert

Hierbei wird sich zunutze gemacht, dass der EDF-Algorithmus auch als Greedy-Implementierung eine optimale Lösung in $\mathcal{O}(n \log n)$ findet, wobei n der Anzahl an einzuplanenden Prozessen entspricht. Auch im Folgenden wird eine Greedy-Strategie verwendet.
Der folgende Algorithmus berechnet einen Ressourcenplan, also eine Übersicht über alle zuzuweisenden bzw. zu blockierenden Ressourcen, die dem Scheduler bekannt sind (siehe Kap. 2.1). Dies entspricht also der Zuteilung von Zugriffsberechtigungen auf Services durch Applets (vgl. Kap. 4.2.2). Es wird weiterhin die Notation aus Kapitel 4.5.1 verwendet, Deadlines zeichnen sich also durch frühesten Startzeitpunkt, Dauer des Zugriffs und spätesten Endzeitpunkt aus, Ressourceneigenschaften setzen sich aus Priorität und Deadline zusammen. Hat ein Ressourcenzugriff keine Deadline, wird der späteste Endzeitpunkt als unendlich angenommen. Als relative Deadline wird weiterhin die Differenz aus spätestem Endzeitpunkt und Dauer bezeichnet. Zusätzlich werden folgende Funktionen benötigt:

- **needed**(r) - gibt für eine Ressource r an, welche Services sie benötigt (inkl. hierarchischer Abhängigkeiten)
- **neededServices**(r) - gibt an, welche Services eine Ressource unmittelbar benötigt (direkte Abhängigkeit)

- **neededServices**(s) - stellt für Services das Analog zu neededServices(r) (siehe zweiter Fall in Kap. 4.2.3)

Die Ergebnisse dieser Operationen ändern sich zur Laufzeit nicht und verursachen nur eine sehr geringe Speicherlast, dürfen also im Rahmen der Berechnung des **Ressourcenplans** als konstant angenommen werden, da sie sich ohne Weiteres in effiziente Datenstrukturen wie HashMaps eintragen lassen[12]. Die Laufzeit des entstehenden Algorithmus beträgt somit ebenfalls $\mathcal{O}(n \log n)$[13], wobei n der Anzahl an Ressourcen entspricht.

Im Algorithmus aus Abb. 4.19 finden sich alle oben geforderten Kriterien wieder: Die Liste aller Ressourcen wird von höchster zu niedrigster Priorität abgelaufen. Dabei werden nur so viele Ressourcenzugriffe genehmigt, wie von diesem Standpunkt aus möglich. Es handelt sich also um einen Greedy-Algorithmus. Wird innerhalb der Planung einem Applet eine Ressource zugewiesen, kann dies durchaus spätere Zuweisungen verhindern. Da alle Ressourcen der von dieser Ressource benötigten Services (inklusive Service-Applets) jeweils mit dem Schritt der Zuweisung ebenfalls zugelassen werden, wird implizit eine Prioritätsvererbung implementiert (siehe auch Kap. 4.4.2). Der rekursive Aufruf dient ebenfalls der Umsetzung von Service-Hierarchien. Exklusive Zugriffe schließen spätere Zuweisungen aus und können nur dann zugelassen werden, wenn nicht bereits ein Zugriff auf den Service gestattet wurde, egal ob exklusiv oder geteilt, da die Zuweisung nur an eine höherpriorisierte Ressource erfolgen konnte. Oszillierendes Verhalten wird dadurch verhindert, dass die Ressourcen auch danach sortiert werden, ob sie bereits zugewiesen sind und ein Triggering-Mechanismus wird ebenfalls

[12] Bei Degenerierung der Hash-Map ist es möglich, dass ein Zugriff eine Worst-Case-Laufzeit von $\mathcal{O}(n)$ aufweist. Wird jedoch eine ausreichend große Hash-Funktion benutzt, so liegt die Average-Case-Laufzeit bei $\mathcal{O}(1)$. In der hier vorgestellten Implementierung werden Hash-Maps von 64 bit Länge verwendet. Die Kollisionswahrscheinlichkeit einer einzelnen Ressource liegt somit für eine vollständige Hash-Funktion bei $(1 - 2^{64})^{|R|}$, ist also selbst bei mehr als einer Million Ressourcen noch über acht Magnituden kleiner als 1%.

[13] Aufgrund der in unseren Szenarien üblicherweise verhältnismäßig geringen Anzahl von Applets und Ressourcen sollte angemerkt werden, dass es sich hierbei ausschließlich um eine asymptotische Schranke für $n \to \infty$ handelt. Bei niedriger Anzahl von Ressourcen und Applets wird der Linearzeitanteil den Algorithmus deutlich dominieren.

Eingaben

1 $R = \{r_1, ..., r_n\}$ // *Ressourcen*

2 $P = \{p_1, ..., p_n\}$ // *Prioritäten*

3 $D = \{d_1, ..., d_n\} = \{(t_{early,1}, t_{duration,1}, t_{finish,1}), ...\}), (t_{early,2}, ...), ...$ // *Deadlines*

4 $E \in P \times D = \{e_1, ..., e_n\} = \{(p_1, d_1), (p_2, d_2), ..., (p_n, d_n)\}$ // *Eigenschaftstupel*

5 $W_{t-1} = \{w_{t-1,1}, w_{t-1,2}, ..., w_{t-1,m}\}$ // *Working-Set (bekannte Ressourcen) mit* $W_0 = \emptyset$

Initialisierung

6 $W_t \leftarrow W_{t-1} \cup R$; // *Working-Set um fehlende Elemente aus R erweitern (anhängen)*

7 $R' \leftarrow \emptyset$; // *Zu gestattende Ressourcenzugriffe*

8 $services(R') \leftarrow \emptyset$; // *Zugewiesene Services nach* R'

9 $X \leftarrow \emptyset$; // *Exklusiv zugewiesene Services*

Algorithmus

Schritt 1: // **Entferne Ressourcen mit Parent (Unterelement von Resource-Bundle)**

10 `Für alle` i `mit` $i \in \{1, ..., |W_t|\}$;

11 `Wenn hasParent(`$w_{t,i}$`):` $W_t = W_t \setminus \{w_{t,i}\}$;

Schritt 2: // **Sortiere Working-Set**

12 `Sortiere` W_t `nach` $p_{w_{t,i}} < p_{w_{t,j}}$;

Schritt 3: // **Liste von höchster zu niedrigster Priorität ablaufen**

13 `Für alle` i `mit` $i \in \{1, ..., |W_t|\}$;

14 `Wenn` $w_{t,i} \notin R'$; // *Durch Hierarchie bereits zugewiesen?*

15 `Wenn allowed(`$w_{t,i}$`);` // *Prüfe ob gestattet (siehe Algorithmus 4.20)*

16 $R' = R' \cup \{w_{t,i}\}$; // *Gestatte der Ressource Zugriff*

17 `Wenn` $w_{t,i}$ `exklusiv;`

18 $X \cup \{$ `service(`$w_{t,i}$`)` $\}$; // *Markiere Service als exklusiv*

19 `needed` $\leftarrow$ `needed(`$w_{t,i}$`);`

20 `Für alle` $m_i \in$ `needed;`

21 $R' = R' \cup \{m_i\}$; // *Gestatte benötigten Ressourcen Zugriff*

22 $services(R') = services(R') \cup \{m_i\}$; // *Markiere Service als zugewiesen*

23 `Wenn` m_i `exklusiv;`

24 $X \cup \{$ `service(`m_i`)` $\}$; // *Markiere Service als exklusiv*

Rückgabe

25 R' // *Liste aller zuzulassenden Ressourcenzugriffe*

Abbildung 4.19: Algorithmus calculateAssignments

Hauptalgorithmus zur Berechnung eines Ressourcenplans

Eingaben

1 $R = \{r_1, ..., r_n\}$ // *Ressourcen*

Initialisierung

2 $X \leftarrow \emptyset$; // *Exklusiv zugewiesene Services*

Algorithmus

Schritt 1: // **Ressourcenzugriff bereits gestattet?**

```
3      Wenn w ∈ R';
4         return true;
5      dependencies ← neededServices(w);
6      Für alle s ∈ dependencies;
7         Wenn allowed(s) = false;
8            return false;
9      return true;
```

Rückgabe

10 R' // *Liste aller zuzulassenden Ressourcenzugriffe*

Abbildung 4.20: Algorithmus allowed(r)

Dieser Algorithmus berechnet für eine Ressource, ob ihre Freigabe auf Basis der aktuell bereits einkalkulierten Ressourcen noch gestattet werden kann.

Eingaben

1 $R = \{r_1, ..., r_n\}$ // *Ressourcen*

Initialisierung

2 $X \leftarrow \emptyset$; // *Exklusiv zugewiesene Services*

Algorithmus

Schritt 1: // **Service bereits exklusiv vergeben?**

```
3    Wenn s ∈ X;
4       return false;
```

Schritt 2: // **Service geteilt vergeben aber exklusiv gewünscht?**

```
5    Wenn nicht exklusiv;
6    Wenn s ∈ services(R');
7       return false;
```

Schritt 3: // **Abhängigkeiten vom Scheduler erfragen**

```
8    dependencies ← neededServices(s);
```

Schritt 4: // **Handelt es sich um ein Service-Applet?**

```
9    Wenn fromServiceApplet(s);
10      applet ← serviceApplet(s);
11      found[ 1 ... |dependencies| ] ← false;
12      Für alle r ∈ Ressourcen(applet);
13         Wenn allowed(r);
14            Für alle s_a ∈ services(r);
15               found(s_d) = true;
16      Für alle s_d ∈ dependencies;
17         Wenn nicht found[s_d];
18            return false;
19   Sonst;
20      Für alle s_d ∈ dependencies;
21         Wenn nicht allowed(s_d);
22            return false;
23      Wenn exklusiv;
24         X = X ∪ s_d;
25   return true;
```

Abbildung 4.21: Algorithmus allowed(s)

Dieser Algorithmus berechnet auf Basis der aktuell bereits eingeplanten Ressourcen, ob ein weiterer Zugriff auf einen Service s noch gestattet werden kann.

implizit beachtet, da eine temporär erhöhte Priorität (siehe Kap. 4.5.2) dafür sorgt, dass eine Ressource in Richtung Listenanfang verschoben wird.

Verhandlungsrunden bei Ressourcenentzug

Ergibt ein so berechneter Ressourcenplan nun, dass ein Applet, welches zuvor eine Ressource gehalten hat, diese vorübergehend entzogen bekommt, so bedeutet dies für das Applet im schlimmsten Falle eine direkte Beeinträchtigung der Funktionalität, üblicherweise aber zumindest eine Einschränkung. Es ist somit notwendig, ein Applet über den geplanten Entzug dieser Ressource zu informieren und ihm gegebenenfalls noch Zeit für Sicherungsaktionen einzuräumen. Ressourcenzugriffe sollten grundsätzlich so transparent wie möglich sein, demnach also keine Rückschlüsse auf innere Zustände eines Services zulassen. Besonders deutlich wird dieses Problem, wenn der Roboter einen Hardware-Zustand einnimmt, der einen solchen transparenten Servicezugriff erst gar nicht ermöglicht:
Der bereits zuvor vorgestellte *Docking-Controller* (siehe Kap. 4.2.3) erfordert beispielsweise ein geradliniges Anfahren der Ladestation von einer räumlich direkt vor ihr befindlichen, fest definierten Position. Notwendig wird dies durch die Tatsache, dass die Ladekontakte möglichst senkrecht in die Ladebuchse eingeführt werden müssen um Schäden an der Hardware zu vermeiden. Ist der Roboter nun in einem solchen Ladezustand und die Ladekontakte befinden sich in der Ladebuchse, so ist es natürlich ebenso notwendig, die Ladestation wieder geradlinig zu verlassen.
Anschaulich wird das Problem, wenn nun das Batterie-Lade-Applet den Roboter in einen solchen Zustand versetzt hat. Der Scheduler kalkuliert einen Ressourcenplan, der dem ebenfalls in Kapitel 4.2.3 definierten Drive-To-Applet den *Pilot*, also das Motorsteuerungsmodul des Roboters, übergibt. Der Docking-Controller baut jedoch auf ebendiesem Service (dem Pilot) auf, weshalb die Ressource „Docking-Controller" dem Batterie-Lade-Applet vorübergehend entzogen werden muss. Das Drive-To-Applet hält selbstverständlich keine Informationen darüber, in welchem Ladezustand der Roboter sich gerade befindet und ob er an der Ladestation angedockt ist. Folglich berücksichtigt er dies bei der Ausführung von Fahrkommandos auch nicht. Eine Drehung des Roboters

könnte nun jedoch zum Abreißen der Ladekontakte an Ladestation und Roboter führen. Vor dem finalen Entzug des Docking-Controllers muss dem Batterie-Lade-Applet somit ausreichend Gelegenheit geboten werden, den Roboter wieder abzudocken, damit ernsthafte Schäden an der Hardware vermieden werden können (siehe auch Kap. 4.6). Für ebendiesen Zweck dienen Verhandlungsrunden. Sie stellen das Format für die Kommunikation von derartigen Anfragen. Eine Verhandlungsrunde besteht dabei aus der Applet-seitigen Definition von:

- der Mindestdauer, die das Applet die Ressource vor Entzug noch benötigen wird
- der Maximaldauer, die es benötigt
- ein Vote, also eine Zustimmung oder Ablehnung des Ressourcenentzugs
- einem Code-Teil, der zuvor ausgeführt wird
- einer Verhaltensdefinition für den Fall der Zustimmung des Schedulers
- eine Verhaltensdefinition für den Fall der Ablehnung dieser Definition durch den Scheduler

Die beiden Verhaltensdefinitionen entsprechen dabei jeweils der Definition einer weiteren, vollständigen Verhandlungsrunde. Die Maximaldauer einer Verhandlungsrunde oder Ausführung ist derzeit auf 20 Sekunden beschränkt, jedoch variabel einstellbar. Die maximale Anzahl an Verhandlungsrunden legt der Scheduler fest. Ist eine Verhandlung nicht erfolgreich, oder das definierte Verhalten konnte vollständig ausgeführt werden, wird die Ressource entzogen (siehe auch Kap. 4.6.1). Für die Ausführung der Verhandlungsrunden werden auch hier die bereits in Kapitel 4.4.1 vorgestellten Interrupt-Handler verwendet.

4.6 Sicherheit und sicherheitskritische Anwendungen

Wie bereits in Kapitel 4.5.3 veranschaulicht, gibt es durchaus Situationen in denen ein Versagen der Scheduler-, vor allem aber der Applet-internen Logik ernsthafte Schäden

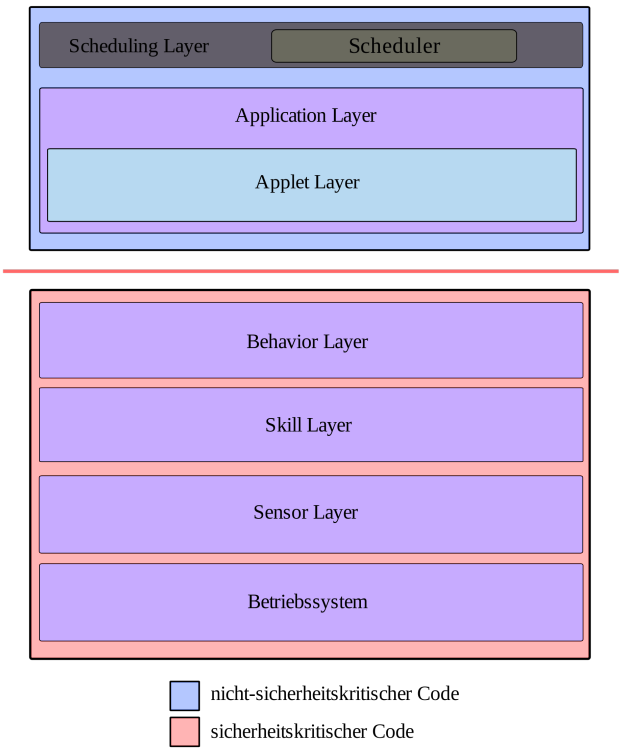


Abbildung 4.22: Vorgeschlagene Trennung von sicherheitskritischen und nicht-sicherheitskritischen Anwendungen
Die dargestellte Struktur basiert auf der funktionalen Architektur aus Kap. 4.3.

an der Hardware des Roboters verursachen kann. Dies gilt umso mehr, je komplexer die realisierten Abläufe sind. Spezialfälle derartiger Konflikte sind der Zwangsentzug von Ressourcen nicht mehr reaktiver Applets (Kap. 4.6.1) sowie die Behandlung von unerfüllbaren Ressourcenplänen bei kollidierenden oder aufgrund einer niedrigeren Priorität überschatteten Deadlines (Kap. 4.6.2). Diese werden deshalb separat behandelt.
In jedem Falle sollte beachtet werden, dass der hier vorgestellte Scheduler in erster Linie als Backend der Roboter-Anwendungsebene, und eben nicht als *vollständige* Robotik-Plattform entwickelt wurde. Hinzu kommt, dass der Scheduler weder harte noch weiche Echtzeit-Garantien vollständig implementiert. Weder die Ablaufsteuerung selbst, noch das im Rahmen der Entwicklung verwendete MIRA-Framework sind für ein Echtzeitbetriebssystem ausgelegt oder könnten dessen Funktionalität vollständig ausschöpfen.

Da die Scheduler-Implementierung direkt auf MIRA-Funktionalitäten aufsetzt, steht ein direkterer und damit performanterer Zugriff zumindest über das Framework selbst zur Verfügung. Sicherheitskritische Anwendungsszenarien oder Anwendungen sollten deshalb wenigstens auf Behavior- oder Skill-Level der funktionalen Architektur verschoben werden und keiner Ablaufsteuerung durch den Scheduler unterliegen. Im Zweifel gestattet die Architektur weiterhin eine Trennung von sicherheitskritischen und nicht-sicherheitskritischen Anwendungsteilen, so dass, wie in Abb. 4.22 dargestellt, die Benutzeroberfläche sowie unkritische Abläufe auf Applet-Layer, sicherheitsrelevante jedoch auf Behavior-Layer oder sogar darunter umgesetzt werden sollten. Darüber hinaus empfehlen sich für derartige Anwendungsbereiche grundsätzlich Robotik-Middlewares oder -Betriebssysteme, beispielsweise ROS und MIRA. Sind harte Echtzeitanforderungen zu erfüllen, so sollten Echtzeitbetriebssysteme oder sogar Hardware-Implementierungen bevorzugt werden. Dies gilt selbstverständlich auch für die nach diesem Konzept festgelegten Deadlines. Diese werden zwar nach Möglichkeit erfüllt, weder gibt es aber vollständige Garantien auf die rechtzeitige Ausführung, noch ist die Genauigkeit des Timings ausreichend für kritische Anwendungsbereiche (siehe Kap. 4.5.1).

4.6.1 Zwangsentzug von Ressourcen

Wie bereits in Kapitel 4.5.3 beschrieben, steht es einem Applet frei, auf einen geplanten Ressourcenentzug entsprechend zu reagieren. Diese Reaktion schließt mit ein:

- ... ob eine Ressource überhaupt entzogen werden darf
- ... wie viel Zeit ein Applet vor einem Ressourcenentzug noch benötigt
- ... welche Aktionen durch das Applet zuvor ausgeführt werden müssen

Die Definition des Entzugsverhaltens sowie dessen Ausführung erfolgen im Applet selbst. Sollte jedoch selbst nach mehreren Verhandlungsrunden noch keine Einigung zwischen Scheduler und Applet erfolgen, oder aber ein Konflikt mit den Beantragungsfristen der anderen Applets auftreten, so wird der ursprünglich errechnete Ressourcenplan zur Gewährleistung der Restfunktionalität weiterverfolgt. In diesem Falle wird

dem Applet die Ressource zwangsentzogen und das Applet hierüber benachrichtigt. Weitere Methodenaufrufe zu dieser Ressource sorgen auf Seite des aufrufenden Applets zu einem Blockieren des Threads bis zur Wiedererteilung der Ressource, um konkurrierende Zugriffe auf den dahinterliegenden Service zu unterbinden[14]. Um ein innerhalb des Applets stets eindeutig bestimmtes Verhalten zu gewährleisten, sollte ein Implementierender deshalb darauf achten, dass ein Applet diesen Konventionen folgt. Dies vorausgesetzt, befinden sich alle Applets, Services und Ressourcen stets in einem wohldefinierten Zustand. Interferenzen werden ausgeschlossen. Da weitere Zugriffe bei Abweichungen des Verhaltens eines Applets zum Blockieren des entsprechenden Applet-Threads führen, schlägt sich undefiniertes Verhalten innerhalb des Applets somit ausschließlich auf dieses, nicht jedoch auf andere, oder gar den Scheduler nieder. Das System als solches bleibt weiterhin valide. Nicht auszuschließen sind selbstverständlich kaskadierende Effekte, sofern andere Applets direkte oder indirekte Abhängigkeiten zu diesem aufweisen (z.B. durch Services). Die Einhaltung der Konventionen bei Service-Applets ist also umso wichtiger.

4.6.2 Unerfüllbarkeit von Schedules

Es gibt prinzipiell auch die Möglichkeit, dass Anforderungen gar nicht erfüllbar sind, beispielsweise, weil sie mit entsprechenden Deadlines versehen sind. Auch diesen Fall gilt es also zu betrachten. Da es sich bei der vorgestellten Ablaufsteuerung im Wesentlichen um eine Implementierung für die Koordinierung - und eben nicht um die vollständige Erfüllung - aller Zugriffswünsche handelt, ist den berechneten Ressourcenplänen ebenfalls inhärent, auch nicht die Erfüllung aller Anforderungen zu garantieren. Einige der üblichen Ziele eines Betriebssystem-Schedulers wie das Verhindern des Aushungerns von Prozessen werden also weder erfüllt noch überhaupt angestrebt (siehe Kap. 2.2). Die Berechnung eines entsprechenden Zuweisungsplans entspricht also - wie bereits genannt - in erster Linie einer Momentaufnahme der aktuellen Notwendigkeit

[14]Ein Applet hat ebenfalls die Möglichkeit, seine Ressource anzuweisen, statt den Thread zu blockieren, eine Exception - also einen Ausnahmefehler - zurückzugeben, welcher dann wiederum vom Applet behandelt werden kann (siehe Kap. 4.6.2).

unterschiedlicher Ressourcenzuweisungen. Wie bereits in Kapitel 4.5 begründet, basieren die Scheduler-Ziele bei der Berechnung auf zwei wesentlichen Kriterien:

- der Priorität eines Ressourcenzugriffes, und
- den Deadlines, die innerhalb einer Prioritätsstufe gesetzt werden

Das erste der beiden Kriterien wird dabei stets vor eventuell gesetzten Deadlines betrachtet (siehe Kap. 4.5.3). Ein Konflikt, der die Erstellung eines erfüllbaren Zeitplanes also gänzlich verhindert, kann somit auf folgende Arten entstehen:

- eine Ressource mit Deadline ist niedriger priorisiert als eine andere mit oder ohne Deadline (siehe Kap. 4.6.2)
- zwei Ressourcen gleicher Priorität haben überschneidende Deadlines
- nach der Zuweisung einer Ressource mit Deadline wird eine weitere mit niedrigerer relativer Deadline bekannt

Es gilt weiterhin die Definition der Deadline aus Kapitel 4.5.1: $d_r = (t_{early,r}, t_{duration,r}, t_{finish,r})$ mit $t_{early,r}$ = frühestem Startzeitpunkt, $t_{duration,r}$ = Dauer und $t_{finish,r}$ = spätestem Endzeitpunkt eines Zugriffes auf Ressource r. Die relative Deadline eines Zugriffs berechnet sich zu:

$$d_{rel,r} = t_{finish,r} - t_{duration,r} \tag{4.8}$$

Eine Deadline d_i ist außerdem nur dann gültig, wenn gilt[15]:

$$t_{early,r} + t_{duration,r} \leq t_{finish,r} \tag{4.9}$$

$$t_{duration,r} + t \leq t_{finish_r} \tag{4.10}$$

Wobei t die aktuelle Zeit bezeichnet. Dass sich nun zwei Deadlines d_i, d_j überschneiden, entspricht genau dem Fall, dass:

$$\min(t_{early,i}, t_{early,j}) + (t_{duration,i} + t_{duration,j}) > max(t_{finish,i}, t_{finish,j}) \tag{4.11}$$

[15]Dies stellt eine ausschließlich theoretische Betrachtung mit unendlich kurzer Zeit für den Kontextwechsel dar. In der Praxis benötigt dieser eine Zeit c, wobei es sich sogar um mehrere Sekunden handeln kann (siehe Kap. 4.5.3). Der Term ergibt sich somit zu $t_{early,r} + t_{duration,r} + c \leq t_{finish,r}$, bzw. $t_{duration,r} + t + c \leq t_{finish,r}$. Dies sollte bei der Implementierung beachtet werden.

Sollte der Scheduler bei Berechnung des Ressourcenplans einen solchen Konflikt feststellen, so werden beide beantragenden Applets hierüber informiert und können gegebenenfalls darauf reagieren (siehe Kap. 4.5.3).

Best-Effort

Bei Berechnung des Ressourcenplans wird ein s.g. Best-Effort-Ansatz verfolgt. Ursprünglich ein Begriff für Service-Garantien in Telekommunikationsnetzen, steht Best-Effort für einen Ansatz der Zusicherung von Services nach besten Bemühungen, also ohne feste Garantien. Auf den Scheduler übertragen bedeutet dies, dass - nach prioritärer Abwägung (siehe Kap. 4.5.3) - möglichst viele Ressourcenwünsche gleichzeitig erfüllt werden, bis keine weiteren Zusicherungen mehr möglich sind, da entweder:

- ... keine weiteren Wünsche existieren, oder
- ... neue Zuweisungen im Konflikt zu bereits vorhandenen stehen würden.

Dies gilt sowohl für die Beachtung von Deadlines (Kap. 4.5.1), als auch für das prioritätengesteuerte Scheduling (Kap. 4.5.2) im Allgemeinen. Wie weiter oben bereits erläutert, ist es möglich, dass Deadlines im Konflikt zueinander stehen, woraufhin eine Benachrichtigung der jeweils beantragenden Applets erfolgt, um eventuelle Reaktionen (z. B. Neubeantragungen) zu ermöglichen. Ein Applet *kann* nun auf diese Benachrichtigung reagieren, *muss* es allerdings nicht. Beachtet die Implementierung eines Applets eine solche Benachrichtung *nicht*, so geht es das Risiko ein, dass die entsprechende Deadline nicht eingehalten werden kann. Dies liegt allerdings in der Systematik von Deadlines und eben nicht der Ablaufsteuerung begründet. Konfligieren mehrere, so erfordert dies grundsätzlich eine Festlegung neuer Deadlines. Ein solcher Konflikt lässt sich also auf Seite der Ablaufsteuerung weder vermeiden noch beheben[16]. Best-effort bedeutet in diesem Falle, dass der Zugriff auf die Ressource im Nachhinein gewährt wird, sie also als *soft deadline*[17] betrachtet wird.

[16] Applets werden ebenfalls darüber benachrichtigt, welchem anderen Applet eine Ressource übergeben wird. Dies ermöglicht im Zweifel eine Kommunikation mit diesem.

[17] Eine *soft deadline* sollte zur Deadline erledigt sein, kann jedoch auch nachgeholt werden. Sie steht dabei im Gegensatz zur hard deadline, die den Sinn ihrer Abarbeitung mit Überschreiten der

Fail-Stop

Das hier vorgestellte System erlaubt verschiedenste Abläufe und erfordert keine besonderen Einschränkungen bezüglich der genauen Implementation, weshalb es jedoch auch notwendig wird, den Scheduler insbesondere für hohe Fehlertoleranz auszulegen. Ein Versagen eines Ressourcenzugriffs oder sogar der vollständige Absturz eines Applets darf (sofern keine direkten Abhängigkeiten zum Applet bestehen) nicht die Funktionalität der übrigen Systemkomponenten beeinflussen oder gar zu einem Systemabsturz führen. Um dies zu realisieren, wird jedes Applet innerhalb seines eigenen Threads ausgeführt (siehe Kap. 4.4). Jeder Applet-Thread wird seinerseits wiederum innerhalb eines Wrappers ausgeführt, der eventuell auftretende Fehler behandeln kann, so dass ein Absturz der restlichen Software verhindert wird. Auf Probleme systematischer Art, die also Beschränkungen für die Systemarchitektur selbst einführen, wurde in Kapitel 4.6 bereits genauer eingegangen. Im Folgenden soll nun das Verhalten des Applets bei *Zugriff trotz Nicht-Erhalt* einer Ressource eingegangen werden:
Greift ein Applet ohne die Erlaubnis des Schedulers auf eine Ressource zu, so blockiert der Applet-Thread bis zur erneuten Zugriffserteilung auf die Ressource. Es gibt jedoch einige Fälle, wo eine Reaktivität des Applets weiterhin nötig ist. Es kann zum Beispiel sein, dass ein Applet einen Ressourcenzugriff eigentlich atomar ausführen wollte (beispielsweise beantragt durch eine entsprechende Deadline), dieser jedoch aufgrund eines Konfliktes unterbrochen wurde (siehe Kap. 4.6.2 weiter oben). Hierdurch kann eine Wiederholung des gesamten (eigentlich atomaren) Ablaufes nötig werden. Wie bereits in Kapitel 4.6.1 kurz angerissen, kann ein Applet auf eine Unterbrechung eines Ressourcenzugriffs deshalb nicht nur mit dem Blockieren des entsprechenden Applet-Threads reagieren. Es ist ihm durch Beantragung dieses Verhaltens ebenfalls möglich, bei Zugriff auf die Ressource ohne Erlaubnis eine entsprechende Fehlermeldung (*Exception*) auslösen zu lassen. Diese kann dann vom Applet behandelt werden. Die Scheduler-seitige Implementierung erlaubt somit bei einem ursprünglich fehlgeschlagenen Ressourcenzugriff nicht nur das Warten auf die Validität ebendieses Zugriffes, sondern ebenfalls die Unterbrechung des gesamten Vorgangs (*Fail-Stop*). Die Wieder-

Deadline verliert.

aufnahme der ursprünglichen Tätigkeit durch das Applet wird als *Fail-Stop-Return* bezeichnet.

Kapitel 5

Test und Validierung

Im Folgenden soll untersucht werden, wie sich die dynamische Ablaufsteuerung in virtuellen sowie realistischen Anwendungsszenarien verhält. Dazu sollen zunächst Bewertungsmaßstäbe aufgestellt werden, welche daraufhin im Rahmen einiger funktionaler Tests sowie eines Realweltszenarios untersucht werden sollen.

5.1 Bewertungsmaßstab

Wie bereits in Kapitel 1.3 geschildert, ist es aufgrund mehrerer Tatsachen schwierig, einen geeigneten Maßstab für die Bewertung von Erfolg und Misserfolg zu definieren. Zum einen gibt es bisher nur wenige gleichartige Arbeiten. Es ist deshalb nicht einfach, vergleichbare Werte oder aber andere anwendbare Bewertungsmaßstäbe zu finden. Zum anderen kann eine Ablaufsteuerung je nach Anwendungszweck unterschiedlichste Zielstellungen haben. Diese wiederum haben in hohem Maße Einfluss auf den jeweiligen Systementwurf. Wesentliche Ziele können u.A. sein:

- Adaptivität
- geringe Redundanz
- hohe Generik bzw. hohe Spezifik
- Modularität und Erweiterbarkeit

- konstantes Systemverhalten
- prädizierbares Systemverhalten
- unmittelbare Nachvollziehbarkeit

Wie aus der Liste hervorgeht, stehen einige dieser Ziele im direkten Widerspruch zueinander (bspw. Generik/Spezifik). Ebenfalls könnte man sie beliebig erweitern, da sich für beinahe jede funktionale oder nicht-funktionale Anforderung ein entsprechendes Anwendungsszenario aufstellen ließe. Wie jedoch bereits unter Kapitel 1.2 begründet, wurden bei der Konzeption der hier vorgestellten Ablaufsteuerung folgende Kriterien als besonders wichtig erachtet:

- Nebenläufigkeit
- Modularität
- Wiederverwendbarkeit
- Nachvollziehbarkeit

In den folgenden Kapiteln wird deshalb insbesondere auf die Erreichung dieser Ziele eingegangen. Darüber hinaus bietet es sich jedoch an, im direkten Bezug zur Ablaufsteuerung einige spezifischere Charakteristika zu untersuchen:

- Wie hoch ist die durchschnittliche Wartezeit auf eine Ressource?
- Wie hoch ist die maximale Wartezeit auf eine Ressource?
- Welchen Overhead erzeugt das Scheduling (Verhältnis der Rechenzeiten von Scheduling-Prozess und Applets)?
- Werden die Kriterien des hybriden Schedulings (siehe Kap. 4.5.3) stets erfüllt?
- Falls nicht, welche Umstände bedingen eine Nicht-Erfüllung?

Da diese Kriterien eine genauere Abschätzung der Performanz des verwendeten Verfahren erlauben und sich ebenfalls in messbaren Zahlen angeben lassen, ist für sie eine Bewertung deutlich einfacher als für die o. g. nicht-funktionalen Eigenschaften.

5.2 Funktionale Tests

Im Rahmen der Entwicklung der dynamischen Ablaufsteuerung wurden verschiedene Test-Cases definiert, die das korrekte Verhalten des Schedulers insgesamt, sowie der einzelnen Komponenten überprüfen. Zur Ausführung und Auswertung wurde der Continious-Integration-Server „Jenkins" verwendet. Genauer wird hierauf in Kapitel 5.2.1 eingegangen. In Kapitel 5.2.2 wird weiterhin ein Stress-Test vorgestellt, der auf Basis von Randbedingungen randomisiert verschiedene Fälle testet und den Scheduler unter Volllast betreibt.

5.2.1 Continious Integration

Bei der **Continious Integration** (dt.: kontinuierliche Integration; kurz **CI**) handelt es sich um ein software-technisches Hilfsmittel zur Vermeidung von Entwicklungsfehlern sowie zur Steigerung der Produktivität und Software-Qualität. Der Entwickler definiert hierfür Testfälle zur Überprüfung der korrekten Funktionalität einzelner Komponenten und deren Interaktionen. Zur Ausführung der Testfälle wird meist ein Continious Integration Server verwendet[1]. Dieser ist mit dem Software-Repository verbunden, kompiliert jede neu hochgeladene Version und führt dann vollautomatisiert alle vorhandenen Test-Cases aus. Nach der Ausführung informiert er den Entwickler, ob die Test-Cases erfolgreich waren.
Für jede funktionale Eigenschaft des Schedulers sollte ein eigener Test-Case entwickelt werden. Interessant sind allerdings insbesondere diejenigen Test-Cases, die komplexe Interaktionen zwischen den einzelnen Komponenten untersuchen. In Anhang A ist ein Test-Case angegeben, welcher in der Lage ist, gleich mehrere Fehlerfälle abzufangen. Er untersucht unter Anderem das Timing des Schedulers, Ressourcenzuteilungen & -entzug, exklusive sowie geteilte Zugriffe auf Services, Synchronizität sowie die Ausführung nebenläufiger Applets. Continious-Integration-Tests eignen sich also inbesondere dann, wenn gleich mehrere Funktionen oder die Interaktionen verschiedener Kompo-

[1] Als CI-Server wurde, wie bereits in Kapitel 5.2 genannt, „Jenkins" verwendet. Jenkins bietet eine gute Integration für alle gängigen Versionierungstools sowie eine Reihe von Ticket- und Code-Review-Systemen

nenten untersucht werden sollen.
Ein grundsätzliches Problem der Continious Integration besteht jedoch darin, dass der Entwickler das Problem von zwei völlig unterschiedlichen Standpunkten zu betrachten hat: Der Entwickler kennt seinen Code, und weiß dementsprechend, wie eine Komponente anzusteuern ist. Er soll auf der anderen Seite aber einen Test-Case für Fälle entwickeln, bei denen sich ein Implementierender zwar an die Syntax des entstehenden Produktes hält, diese jedoch nicht im korrekten Kontext verwendet. Bei der Entwicklung von Test-Cases muss also das Funktionieren der Komponenten bei richtiger Verwendung ebenso sichergestellt werden wie das Scheitern bei einer falschen. Weitere verwendete Testfälle schließen unter anderem ein: *Benutzung von hierarchischen Service-Definitionen*, *Triggering-Mechanismus*, *MIRA-Integration*, *Zwangsentzug von Ressourcen*, *Timing bei Ressourcenentzug*, *Verhandlungsrunden*.

5.2.2 Stress-Test

Bei einem Stress-Test handelt es sich um einen groß angelegten, vollautomatisierten Test der Systemfunktionalität. Hierfür werden s. g. *constraints*, also Randbedigungen eingeführt, innerhalb derer randomisiert verschiedene Szenarien durchlaufen werden. Durch die zufällige Anwendung können verschiedenste Ablaufsequenzen getestet werden, die sonst nur aufwändig über jeweils einzelne Test-Cases zu realisieren wären. Komplexere Stress-Tests können dabei sowohl valide als auch ungültige Szenarien nutzen, in diesem Fall wurde sich jedoch ausschließlich auf Fälle beschränkt, bei denen Scheduler-Funktionen zumindest im Kontext einer für sich funktionierenden App verwendet werden. Der Stress-Tests untersucht also in erster Linie die Interaktionen zwischen den Apps. Er setzt den Scheduler durch randomisiertes Aufrufen verschiedener App-Funktionen unter Volllast und analysiert die bereits oben genannten funktionalen Eigenschaften:

- durchschnittliche Wartezeit
- maximale Wartezeit
- Erfüllung der Scheduler-Bedingungen

Sollten die Scheduler-Constraints einmal nicht eingehalten werden, gibt der Stress-Test gegebenenfalls zusätzliche Informationen zur Fehleranalyse aus. Da die untersuchten Wartezeiten sehr stark vom Verhalten der einzelnen Applets abhängen (bspw. durch Einführung von Verhandlungsrunden), ist hier in erster Linie der Overhead über dem zwingend notwendigen Delay, also die Zeit nach Ausführung von Interrupts, interessant.

Die Ergebnisse des Stress-Tests umfassen insgesamt 5268 Service(neu)zuweisungen. Die erreichte durchschnittliche Wartezeit eines Applets bis zur Verfügbarkeit einer Ressource lag dabei bei 277,45 ms, die maximale Wartezeit auf eine Ressource lag bei 601 ms. Die Ausführungszeiten der eigentlichen Applets waren verhältnismäßig kurz und lagen maximal im Bereich weniger Millisekunden. Dies spiegelte sich ebenfalls in den Ergebnissen wider (siehe Abb. 5.1). Da die Ausführungszeiten der Applets deutlich kleiner sind als die jeweiligen Zykluszeiten, wird der Ressourcenplan gewissermaßen parallel zum Warten auf die nächste Ausführung erzeugt und angewendet. Dies ist, unabhängig vom jeweiligen Verhältnis von Ausführungs- und Zykluszeit, obligatorisch, da eine Zuteilung oder ein Entzug einer Ressource nur zwischen den Ausführungsschritten durchgeführt werden können soll (vgl. Kap. 4.4.1). Die resultierenden Wartezeiten stellen damit Summen der Zykluszeiten betroffener Applets dar, je nachdem, ob mehrere Applets gleichzeitig auf die nächste Ausführung warten oder ob sie noch einen vollständigen Hauptschleifenschritt durchlaufen (siehe Abb. 5.2).

Der entstehende Overhead, also die zusätzliche Wartezeit durch Mechanismen des Scheduling-Prozesses, lag stets im einstelligen Millisekunden-Bereich (Max.: 9 ms; Avg.: $\leq$1 ms). Darüber hinaus war keine Verzögerung durch das Scheduling messbar. Unter den oben gemachten Einschränkungen der Simultanität von Erstellung des Schedules und Warten auf nächste Ausführung bedeutet dies, dass das Scheduling selbst früher beendet war als eine Zuteilung hätte erfolgen können. Setzt man also voraus, dass das Scheduling beim gleichen Abhängigkeitsgraphen auch eine vergleichbare Anzahl an Ausführungsschritten benötigt, so lag die Zeit zur Erstellung eines Ressourcenplans sogar stets unter der kleinsten Zykluszeit, in diesem Fall also unter 100 ms.

Abbildung 5.1: Plot der Wartezeiten auf eine Ressourcenzuteilung
Wie in der Grafik zu sehen, verteilen sich die Wartezeiten auf ein sehr kleines Intervall um die Zykluszeiten der Applets herum.

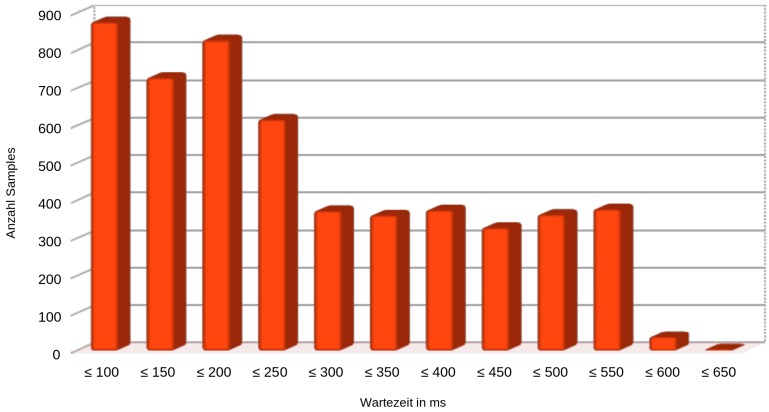


Abbildung 5.2: Histogramm der Wartezeiten auf eine Ressourcenzuteilung
Die unterschiedlichen Wartezeiten resultieren aus den Abfolgen verschiedener Applet-Zykluszeiten, die jeweils vor Neuzuteilung einer Ressource eingehalten werden muss.

5.3 Anwendungstest

Ebenfalls wurde für die Validierung der dynamischen Ablaufsteuerung ein komplexeres Testszenario aufgestellt. Dieses basiert auf dem bereits mehrfach im Rahmen dieser Arbeit verwendeten Beispiel (siehe auch Kap. 4.2.3) konkurrierender Zugriffe, wurde jedoch um einige Komponenten erweitert. Das Szenario ist dabei derart konzipiert, dass alle unter Kapitel 4 vorgestellten Komponenten und Mechanismen darin verwendet werden und eignet sich deshalb sehr gut zur Demonstration des Konzeptes. In den folgenden beiden Kapiteln soll dieses Szenario nun vorgestellt und das Verhalten der Ablaufsteuerung analysiert werden.

5.3.1 Testszenario

Die Basis des Anwendungsszenarios bilden weiterhin die bereits vorgestellten Services:

- Motorsteuerung/„Pilot" - dient der Ansteuerung der Aktuatorik, nimmt Wegpunkte entgegen und manövriert den Roboter dorthin
- Docking-Controller - dient dem Andocken des Roboters an die Ladestation (*Docking-Station*), nutzt dafür innerhalb seiner Implementierung den Pilot

Zusätzlich werden folgende Services eingeführt:

- Batterie-Status-Service - liest den aktuellen Ladezustand der Roboter-Batterie aus
- Monitor-Service - dient der Anzeige von Informationen auf einem Touchscreen am Roboter
- Mausklick-Service - nimmt Benutzereingaben über den Touchscreen entgegen und leitet sie ggf. an ein haltendes Applet weiter
- UI-Service - kombiniert Mausklick- und Monitor-Service, um einen kontextuellen Zugriff zu ermöglichen, fügt Home-Button zur Rückkehr ins Hauptmenü hinzu, wird vom UI-Service-Applet gestellt (s.u.)

Die hier verwendeten Services eignen sich, um gleich mehrere Mechanismen unter Beweis zu stellen. Während der UI-Service ein klassisches Beispiel eines ausschließlich exklusiv zu nutzenden Services ist (und auch nur exklusiv beantragt werden kann), kann im Einzelfall durchaus ein Sinn darin bestehen, die einzelnen Mausklicks- und Bewegungen völlig unabhängig vom angezeigten Kontext wahrzunehmen. Ein entsprechender Fall wäre die Erkennung von Mausgesten zum Wechsel zwischen verschiedenen Applets, die auch unabhängig vom aktuellen Inhalt angewendet werden können soll. Ähnlich wie der UI-Service sollte auch das Monitor-Service ausschließlich exklusiv genutzt werden. Das UI-Applet unterscheidet sich vor allem dadurch, dass es selbst auf anderen Services aufbaut und von einem Applet gestellt wird (s.u.).

Um einen sinnvollen Ablauf darzustellen ist selbstverständlich noch die Einführung entsprechender Applets nötig:

- Batterie-Lade-Applet - lädt, sofern nötig, den Roboter und informiert den Nutzer über das Graphical User Interface
- Drive-To-Applet - Stellt dem Nutzer eine Karte der Umgebung zur Verfügung und erlaubt ihm, den Roboter an eine selbstgewählte Position innerhalb der Karte zu schicken
- Kalender-Applet - dient der Eintragung von Terminen, fährt bei anstehende Terminen zum Nutzer um ihn zu informieren
- Menü-Applet - dient als Hauptseite des Graphical User Interfaces und gestattet dem Nutzer Zugriff auf die jeweils anderen Komponenten
- UI-Service-Applet - dient als Hauptkomponente der Nutzerinteraktion, stellt dafür den UI-Service, sammelt Informationen wie Mausklicks und leitet sie weiter, überträgt Anzeigedaten an den Monitor-Service

Eine Übersicht über verwendete Services und Ressourcen findet sich in Abb. 5.3.
Die Implementierung der einzelnen Applets sieht vor, dass bei Unterbrechungen die ursprüngliche Aufgabe stets fortgeführt werden soll. Das Testszenario besteht nun aus den folgenden Schritten:

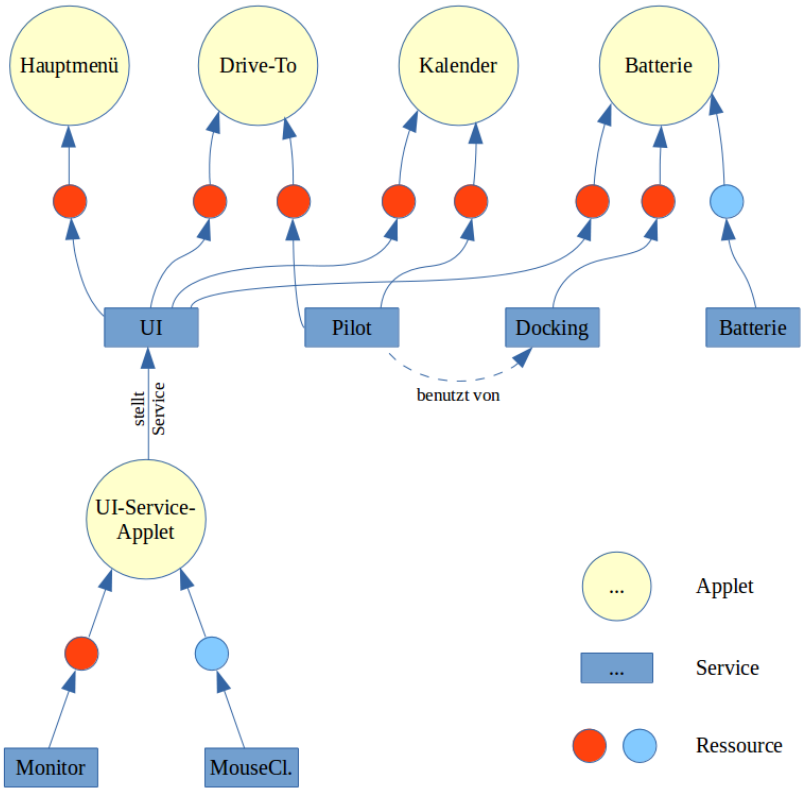


Abbildung 5.3: Komplexes Realweltszenario zur Validierung der dynamischen Ablaufsteuerung

Hierbei handelt es sich um eine Darstellung des in Kapitel 5.3.1 vorgestellten Realweltszenarios. Zu sehen sind die komplexen Abhängigkeiten verschiedener Services und Applets untereinander. Mit Ausnahme von Batterie-, Monitor- und Mausklick-Service handelt es sich dabei ausschließlich um konkurrierende Ressourcenzugriffe.

1. Starten der Applikation
2. Öffnen des Drive-To-Applet über das Hauptmenü, Roboter zu einer Position auf der Karte schicken (Drive-To-Aufgabe setzen)
3. (simuliertes) Abfallen des Batterie-Ladezustandes noch während der Drive-To-Aufgabe, Unterbrechung durch automatisches Einschreiten des Batterie-Lade-Applets
4. (simulierter) Anstieg des Batterie-Ladezustandes, Fortsetzen der Drive-To-Aufgabe aus Schritt 2
5. (simulierter) Termin noch während der (wiederaufgenommenen) Drive-To-Aufgabe, Unterbrechung und Auslieferung des Termins zum Nutzer
6. Bestätigung des Termins durch den Nutzer, Wiederaufnahme der Drive-To-Aufgabe
7. Öffnen des Hauptmenüs durch Klick auf Home-Button

Die Selbstaktivierung der Applets erfolgt hierbei über den in Kapitel 4.5.2 vorgestellten Triggering-Mechanismus. Die tatsächlichen Prioritäten aller Ressourcen sind vollständig äquivalent und bleiben über den gesamten Ablauf konstant. Einzige Ausnahme bilden dabei der UI-Service, der mit Systempriorität läuft, sowie das Hauptmenü, welches zur Sicherstellung der Zuteilung die UI zunächst mit hoher Priorität beantragt, dann jedoch auf die gleiche Priorität wie die übrigen Applets reduziert.

5.3.2 Systemverhalten

Im Folgenden soll die erwartete Zuteilung der Ressourcen bei der Ausführung des oben geschilderte Szenarios betrachtet werden. Wie Abb. 5.3 bereits zu entnehmen ist, existieren für die Services „Batterie", „Mouse-Click" und „Monitor" jeweils nur eine Ressource. Konkurrierende Zugriffe auf sie sind deshalb ausgeschlossen. Auf ihre Betrachtung wird hier also verzichtet. Dem Service „Docking-Controller" sind zwar

Aktion	UI-Service	Pilot	Docking-Controller
Starten der Applikation	Hauptmenü	undef.	undef.
Drive-To-Task setzen	Drive-To	Drive-To	blockiert
Batterie niedrig	Batterie-App	blockiert	Batterie-App
Batterie geladen	Drive-To	Drive-To	blockiert
Termin simulieren	Kalender	Kalender	blockiert
Termin bestätigen	Drive-To	Drive-To	blockiert
Hauptmenü öffnen	Hauptmenü	undef.	undef.

Abbildung 5.4: Geplantes Systemverhalten bei Ausführung des Testszenarios zur Validierung der Scheduler-Funktionalität

ebenfalls keine weiteren Ressourcen zugeordnet, da er jedoch selbst vom Pilot abhängt und dieser durchaus zugeordnete Ressourcen hat, kann der Docking-Controller blockiert werden. Er ist deshalb weiterhin relevant. Die Tabelle in Abb. 5.4 zeigt die erwartete Zuordnung der Services. Wie dort dargestellt, soll der anfangs gestartete Task des Drive-To-Applets nach jeder Unterbrechung fortgesetzt werden. Der gleichzeitige Zugriff auf Pilot und Docking-Controller durch verschiedene Applets ist nicht möglich, da, wie bereits in Abb. 5.3 gezeigt, eine Abhängigkeit von Docking-Controller zu Pilot besteht.

Das vorgestellte Anwendungsszenario wurde zur Validierung der Funktionalität viele Male durchlaufen. Während dabei aufgrund der aktiven Entwicklung in der Anfangsphase noch Probleme mit der Koordination von Zugriffen auf hierarchische Service-Definitionen entstanden, wurde bei einer abschließenden Untersuchung mit $n = 20$ Durchläufen und etwa 60 Minuten Dauereinsatz eine Erfolgsquote von 100% erreicht. Bezüglich der Reaktivität war das Verhalten mit dem aus Kapitel 5.2.2 vergleichbar.

Kapitel 6

Zusammenfassung und Fazit

Das Konzept der dynamischen Ablaufsteuerung stellt definitiv einen Schritt in Richtung eines leicht modifizier- und erweiterbaren Roboters dar. Diese Kriterien sind beide insofern relevant, dass sie jeweils zu einem niedrigerem Entwicklungsaufwand und damit zu einem günstigeren Gesamtsystem führen.
Komplexe Roboter sind derzeit noch ein wesentlicher Forschungsbestandteil und dementsprechend kostenintensiv. Der Entwicklungskostenanteil für Roboter mit einer derart komplexen Software übersteigt bisher bei Weitem den jeweiligen Materialkostenanteil, obwohl auch dieser, je nach Anwendungsszenario, problemlos im Bereich mehrerer Tausend Euro liegen kann. Materialkosten sind ganz wesentlich von der verwendeten Hardware abhängig. Diese sind jedoch nach unten begrenzt, weshalb der Stand der Technik in diesem Bereich zumindest den gleichen Stellenwert wie die verwendete Hardware hat. Er bestimmt die Höhe der Materialkosten entscheidend mit.
Während die Materialkosten pro Roboter konstant sind, teilen sich die Entwicklungskosten jedoch auf die Gesamtabsatzmenge auf. Die Kosten der Entwicklung der verwendeten Software lassen sich in erster Linie also durch eine Effizienzsteigerung reduzieren. Da die hier vorgestellte Ablaufsteuerung ebendieses Ziel verfolgt, könnten sich durch ihren Einsatz sowie ihre konsequente Weiterentwicklung folglich neue Marktperspektiven eröffnen, beispielsweise indem auch weniger liquide Zielgruppen erreicht werden, oder aber, indem die erzielten Kosteneinsparungen in neue Funktionalitäten reinvestiert werden. Nichtsdestotrotz ist die hier vorgestellte Ablaufsteuerung noch lange nicht

marktreif. Wesentliche Komponenten zur Vereinheitlichung der Nutzerinteraktion mit den Applets fehlen ebenso wie eine ausgereifte und auf Dauer ausreichende Plattform. Anreize zur Verbesserung dieser Punkte werden in Kapitel 6.2 diskutiert.

6.1 Erreichung der Zielstellung

Die bereits in Kapitel 1.2 vorgestellten Ziele bestanden in der Erschaffung einer Ablaufsteuerung zur weitestgehend automatisierten Steuerung von geteilten und exklusiven Zugriffen auf *robotische Basisleistungen*, also Fähigkeiten, die eine alltagsfähige Mensch-Maschine-Interaktion zulassen. Nicht-funktionale Anforderung für diese Ablaufsteuerung waren die Realisierung von:

- Nebenläufigkeit
- Modularität
- Wiederverwendbarkeit
- Nachvollziehbarkeit

Das hier vorgestellte Konzept gestattet durch die Nutzung voneinander unabhängiger Applets eine Nebenläufigkeit verschiedener Abläufe. Es ist beispielsweise durchaus möglich, dass Verhaltensweisen wie die Begrüßung, Verabschiedung oder die Interaktion mit Menschen (jede dieser Funktionen als einzelnes Applet realisiert) völlig unabhängig voneinander und ohne Kenntnis der jeweils anderen umgesetzt werden. Die Zusammensetzung zu einem robotischen Gesamtverhalten gestaltet sich, mit Ausnahme einiger Konfigurationsparameter, vollkommen modular und unabhängig von den jeweils anderen Komponenten. Einzelne Module (sprich: Applets) können durchaus auch in einem anderen Kontext verwendet und ohne Weiteres auf andere Roboter übertragen werden. Auch hier eröffnen sich durch Produktvaration und -differenzierung (Veränderung der jeweiligen Verkaufskonfigurationen) neue Marktperspektiven.
Wie bereits zuvor genannt, können Konflikte zwischen verschiedenen Modulen im Einzelfall zwar nicht jedes Mal behoben werden, jedoch ist auch hier die Modellierung

eines Ausweichverhaltens möglich. Angemerkt sei an dieser Stelle, dass auch menschliches Verhalten und menschliche Verhaltensweisen keineswegs stets widerspruchsfrei sind. Die Begrüßung einer Person unmittelbar während der (evtl. sozial fordernden) Interaktion mit einer anderen stellt für uns logisch ebenso einen Konflikt dar, wie es für die entsprechenden Applets der Fall wäre. Durch Konditionierung und lebenslanges Lernen ist es uns jedoch möglich, solche Konflikte aufzulösen. Ist für den Roboter keine Modellierung eines expliziten Ausweichverhaltens möglich, so gestattet das hier vorgestellte Konzept weiterhin, ein lernfähiges Paradigma für die Koordinierung der Zugriffe zu implementieren, ermöglicht zumindest rein syntaktisch also eine ähnliche Konditionierbarkeit wie es bei einem Menschen der Fall wäre. Diese Lernfähigkeit gilt es jedoch noch umzusetzen (siehe Kap. 6.2).
Der Komplexität des verwendeten Modelles geschuldet, fällt die Nachvollziehbarkeit der vorgestellten Ablaufsteuerung im Vergleich zu einfachen Konzepten, wie dem der State-Machine (siehe Kap. 3.2.2), deutlich geringer aus. Dies gilt jedoch nur insoweit, wie die State-Machine selbst als Indikator für den Gesamtzustand des robotischen Systems gewertet wird. Da die Nutzung von verdeckter Logik für die Sinnhaftigkeit ihrer Benutzung jedoch essentiell ist, entspricht auch sie nur einem Draufblick auf die Oberfläche des robotischen Verhaltens. Trotzdem bietet sie einen starreren Rahmen für die Realisierung von Abläufen und ist aus Modellsicht besser nachvollziehbar. Um diesem Faktor entgegenzuwirken wurde bereits eine weitestgehend intuitive Visualisierung der Ressourcenzuteilungen entwickelt (siehe Kap. 4.3). In Zukunft geschaffene weitere Analysewerkzeuge sollten die Nachvollziehbarkeit noch einmal verbessern.

6.2 Weitere Ansätze und Optimierungsmöglichkeiten

Das aus dieser Arbeit hervorgegangene Framework stellt einen Schritt in Richtung einer modularen, generischen und einheitlichen Ablaufsteuerung dar. Trotz des aus dieser Arbeit hervorgegangenen Konzeptes fehlen jedoch noch einige Schritte, um eine vollständige, für einen Nutzer sinnvoll nutzbare, vor allem intuitive Systemumgebung zu erschaffen. Dies umfasst im Wesentlichen:

- ein einheitliches Look-and-Feel
- nutzerspezifische, adaptive Abläufe
- Netzwerkfähigkeit und Dezentralität[1]
- eine durchweg vereinheitlichte API
- eine entsprechende Entwickler-Dokumentation

Die Erschaffung eines gemeinsamen Look-and-Feels soll zukünftig durch die Entwicklung eines geeigneten GUI-Managers erfolgen. Dabei ist geplant, Bildschirmbereiche jeweils als eigenen Service zu modellieren. Möchte ein Applet also Zugriff auf einen Bildschirmbereich erhalten, so beantragt es diesen als entsprechende Ressource. Möchte ein Applet Zugriffe auf die restlichen Bildschirmbereiche durch andere Applets verhindern, so muss es diese ebenfalls beantragen (z. B. Vollbild-Apps).
Weiterhin ist bisher problematisch, dass die vorgestellte Ablaufsteuerung zwar konfliktfreie Vergaben von Ressourcen vornimmt, diese jedoch aus Nutzersicht nicht in jedem Falle sinnvoll sein müssen. Dem soll durch Integration des in [Mueller et al., 2014b] vorgestellten Dialog-Managers entgegengewirkt werden. Dieser bietet bereits umfassende Möglichkeiten, an den Nutzer angepasste Dialoge zu erstellen, was die Akzeptanz des Roboters als alltäglicher Wegbegleiter deutlich vergrößern sollte. Eine Integration in das Scheduler-Konzept würde hier durch einen „adaptiven Scheduler" erfolgen, welcher seine Ressourcenzuteilungen also auf Basis von vergangenen Nutzerinstruktionen vornimmt.
Auch die Integration einer Roboterpersönlichkeit ist hier nicht weiter betrachtet worden. Diese sollte ebenso in die Entscheidung des adaptiven Schedulers einfließen und die Basis der Interpretation von multimodalen Eingaben stellen.
Generell erfolgen Eingaben bisher vornehmlich auf Basis des Graphical User Interfaces. Auch wenn einzelne Applets bereits eine Auswertung weiterer Sensorik vornehmen und auf diese entsprechend reagieren können, muss diese Umsetzung bisher innerhalb jedes

[1] Derzeit müssen sowohl Scheduler als auch Applets jeweils im selben Prozess ablaufen, da beide lediglich in unabhängigen Threads ausgeführt werden. Aktuell schließt dies eine dezentrale Verwendung also konsequent aus.

benutzenden Applets erfolgen. Ein gemeinsames Eingabe-Interface wäre also durchaus nützlich und würde die Ansteuerung der Sensorik sowie Reaktionen auf von deren Daten entsprechend vereinfachen. Weiterhin würde die Redundanz in der sensorischen Auswertung deutlich reduziert, weshalb auch für diesen Zweck entsprechende Service-Applets geplant sind.

Den Schlusspunkt der Arbeit soll ein generisches Robotik-Framework darstellen, welches sowohl die konfliktfreie Ansteuerung von Hardware-Komponenten vornimmt, als auch die Assozziativität menschlicher Kontakte und damit einhergehend die Komplexität sozialen Verhaltens berücksichtigt und - zumindest aus Nutzersicht - selbst realisiert.

Derzeit gibt es nur wenige Arbeiten, die sich überhaupt systematisch mit der automatisierten Steuerung von robotischen Abläufen auf Basis eines fortwährend modular erweiterbaren Roboters befassen. Eine Veröffentlichung in der Open-Source-Community könnte dem Konzept als solchem - wenn auch nicht in der hier vogestellten Implementierung - also durchaus zum Durchbruch verhelfen und die Forschungsbestrebungen in diesem Bereich vergrößern.

Anhang A

Ergänzende Unterlagen

Übersichtsgrafik Eine grobe Übersicht über alle im Konzept vorkommenden Komponenten sowie deren Interaktionen findet sich in Abb. A.1

Beispiel-Test-Case Nachfolgend ist ein Beispiel eines Test-Cases gegeben, welches den grundsätzlichen Aufbau sowie die Funktionsweise der Continious-Integration-Tests verdeutlichen soll. Beschrieben sind hier:

- Untersuchte Funktion
- Services und Applets
- Ablauf des Test-Cases
- Erwartetes Verhalten
- Verhalten bei Fehlern

Test-Case zum Überprüfen der Synchronisationsmechanismen

Funktion

Gegenseitiges Synchronisieren exklusiver und geteilter Ressourcenzugriffe auf Services

Services

Service 1:

Methoden

- increment - inkrementiert internen Counter
- getCounter - gibt aktuellen Wert des Counters zurück

Beide Methoden des Services sind über einen Mutex synchronisiert, es kann jeweils nur auf eine davon gleichzeitig zugegriffen werden.

Applets

Applet 1:

Initialisierung

- geteilte Beantragung des Zugriffes auf den Service

Hauptschleife

- erhöhen des Zählers, wenn Zugriff erlaubt
- erhöhe internen Counter um 1

Methoden

- gib aktuellen Counter-Wert zurück
- schalte um zwischen exklusivem und geteilten Zugriff

Applet 2:

- Aufbau analog zu Applet 1

Ablauf

1. Starte Scheduler und Applets
2. Pausiere Applet 1, speichere Counter
3. Warte 5 Sekunden (entspr. 500 Main-Loops)
4. Prüfe: Counter Applet 1 = gespeicherter Wert; setze Applet 1 fort

5. Prüfe Counter Applet 1 + Counter Applet 2 = Counter Service
6. Setze Zugriffsmodus Applet 1 auf exklusiv
7. Speichere Applet-Counter (1&2)
8. Setze erst Applet 1, dann Applet 2 fort
9. Warte erneut 5 Sekunden
10. Prüfe Counter Applet 1 erhöht
11. Prüfe Counter Applet 2 unverändert
12. Prüfe Counter Service = Counter Applet 1 + Counter Applet 2
13. Ändere Zugriffsmodus Applet 1 auf geteilt
14. Warte 5 Sekunden
15. Prüfe Counter Applet 2 erhöht
16. Prüfe Counter Service = Counter Applet 1 + Counter Applet 2
17. Beende

Erwartetes Verhalten

- keine Fehlermeldung
- alle Tests erfolgreich

Mögliche Fehler

- Service-Counter ungleich Summe der Applet-Zähler (5, 12, 16)
- Aktivität bei Pausierung/Blockierung (4, 11)
- Keine Aktivität nach Fortsetzung (10, 15)

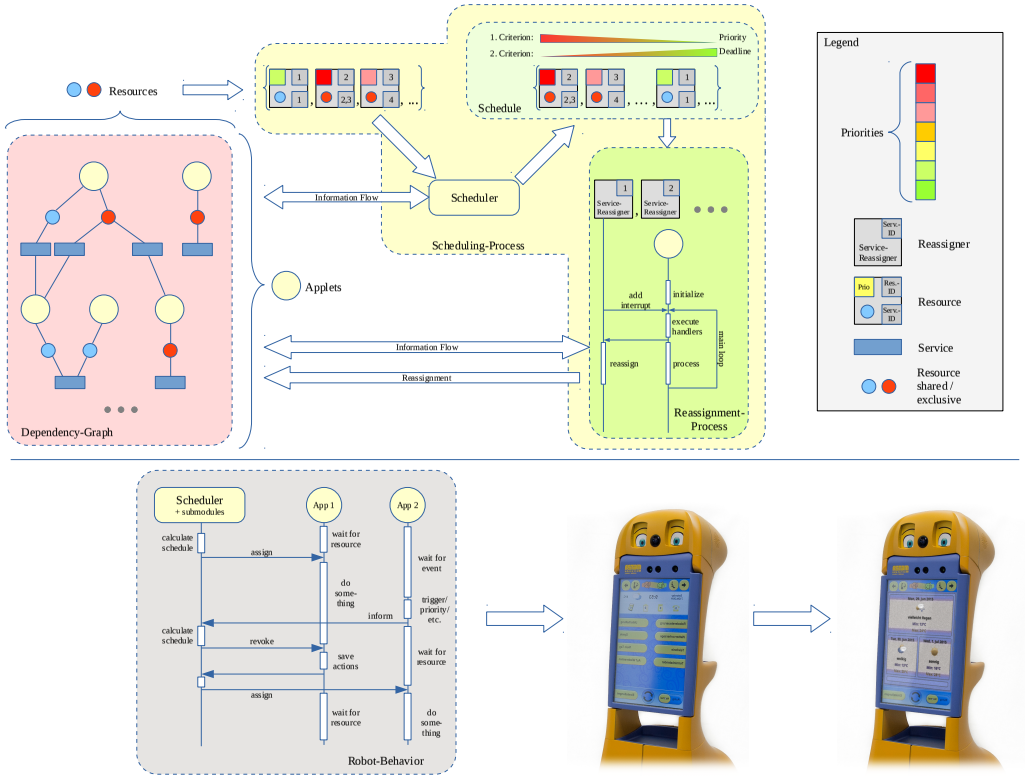


Abbildung A.1: Übersicht Scheduler-Konzept

*In dieser Grafik finden sich alle relevanten Komponenten, Interaktionen und Abläufe der dynamischen Ablaufsteuerung. Das Bild des Roboters stammt ursprünglich aus [*Stricker*, 2011].*

Abbildungsverzeichnis

3.1 Anwachsen der Zustandsmenge durch Gedächtnis 13
3.2 Finite-State-Machine zur Ansteuerung eines Roboters 15
3.3 Darstellung der Probleme bei der Erweiterung einer Ablaufsteuerung auf Basis von State-Machines. 17
3.4 Funktionale Struktur der Anwendungsschicht (ROREAS) 18
3.5 Darstellung der verschiedenen State-Chart-Konnektoren 21
3.6 Ausschnitt Zuse-Guide-SCXML 23
3.7 Behavior-State-Machine des Stack-basierten Systems 26
3.8 Grundsätzlicher Aufbau sowie Komponenteninteraktionen einer App auf Basis des Konzeptes der adaptiven, lernenden Ablaufsteuerung . . . 28
3.9 Robotics in Concert - Das Prinzip des Appable Robot 31

4.1 Komponentendiagramm zum Grundaufbau des Schedulers 37
4.2 Ablauf der Neuzuteilung von Services 38
4.3 Funktionale Architektur der dynamischen Ablaufsteuerung 42
4.4 Übersicht Komponenteninteraktionen 43
4.5 Aufbau von Services und Ressourcen 46
4.6 Gegenüberstellung Ressource und Resource-Bundle 47
4.7 Einordnung hierarchischer Service-Definitionen in das Gesamtkonzept . 47
4.8 Verdeutlichung eines Konfliktes ersten bzw. zweiten Grades 50
4.9 Darstellung einer zirkulären Abhängigkeit 51
4.10 Beispiel-XML-Datei zum Laden einer Applet-Grundkonfiguration . . . 53
4.11 Graphviz-Visualisierung bekannter Applets, Services, Ressourcen 55

4.12 Grundaufbau eines Applets 56
4.13 Einordnung des Service-Reassigners in das Gesamtkonzept 57
4.14 Einfügen eines Interrupt-Handlers in den Ablauf eines Applets durch Service-Reassigner 57
4.15 Prioritätsinversion durch hierarchische Service-Abhängigkeiten 60
4.16 Lösung der Prioritätsinversion durch Prioritätsvererbung 62
4.17 Einordnung des Schedulings auf Basis von Heuristiken 63
4.18 Reihenfolge der Zuweisung mit und ohne Triggering 70
4.19 Algorithmus calculateAssignments 73
4.20 Algorithmus allowed(r) 74
4.21 Algorithmus allowed(s) 75
4.22 Vorgeschlagene Trennung von sicherheitskritischen und nicht-sicherheitskritischen Anwendungen 78

5.1 Plot der Wartezeiten auf eine Ressourcenzuteilung 90
5.2 Histogramm der Wartezeiten auf eine Ressourcenzuteilung 90
5.3 Komplexes Realweltszenario zur Validierung der dynamischen Ablaufsteuerung 93
5.4 Geplantes Systemverhalten bei Ausführung des Testszenarios zur Validierung der Scheduler-Funktionalität 95

A.1 Übersicht Scheduler-Konzept 107

Literaturverzeichnis

[dud, 2015] (2015). *Duden Online-Ausgabe, Stichwort: präemptiv*. Online, URL: `http://www.duden.de/rechtschreibung/praeemptiv` (abgerufen am 01. Juli 2015).

[Bilgin et al., 2012] Bilgin, Arif, J. Ellson, E. Gansner, Y. Hu und S. A. L. R. North (2012). *Graphviz-Website*. Online, URL: `http://http://www.graphviz.org/` (abgerufen am 22. Juni 2015).

[Boehme et al., 1998] Boehme, H.-J., A. Brakensiek, U.-D. Braumann, M. Krabbes und H. Gross (1998). *Neural networks for gesture-based remote control of a mobile robot*. In: *Neural Networks Proceedings, 1998. IEEE World Congress on Computational Intelligence. The 1998 IEEE International Joint Conference on*, Bd. 1, S. 372–377 vol.1.

[Böhm und Jacopini, 1966] Böhm, Corrado und G. Jacopini (1966). *Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules*. Communications of the ACM, S. 366–371.

[Cootes et al., 1998] Cootes, Timothy F., G. J. Edwards und C. J. Taylor (1998). *Active Appearance Models*. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*, S. 484–498. Springer.

[DIN EN 60848:2014-12, 2014] DIN EN 60848:2014-12 (2014). *GRAFCET*.

[Eykholt et al., 1992] Eykholt, J. R., S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, D. Williams

und D. W. Sunsoft (1992). *Beyond Multiprocessing - Multithreading the SunOS Kernel*. In: *in Proceedings of the Summer USENIX Conference*, S. 11–18.

[Gross et al., 2014] Gross, H.-M., K. Debes, E. Einhorn, S. Mueller, A. Scheidig, C. Weinrich, A. Bley und C. Martin (2014). *Mobile Robotic Rehabilitation Assistant for walking and orientation training of Stroke Patients: A report on work in progress*. In: *Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference on*, S. 1880–1887.

[Gross et al., 2015] Gross, H.-M., S. Mueller, C. Schroeter, M. Volkhardt, A. Scheidig, K. Debes, K. Richter und N. Doering (2015). *Robot Companion for Domestic Health Assistance: Implementation, Test and Case Study under Everyday Conditions in Private Apartments*. In: *Intelligent Robots and Systems (IROS), Hamburg, Germany, 2015 IEEE/RSJ International Conference on*. to appear.

[Hall, 1966] Hall, E.T. (1966). *The hidden dimension*. Doubleday Anchor Books. Doubleday.

[Harel, 1984] Harel, David (1984). *Statecharts: A visual formalism for complex systems*. Sci. Comput. Programming 8, S. 231–274.

[Hinton et al., 2006] Hinton, G, S. Osindero und Y. Teh (2006). *A Fast Learning Algorithm for Deep Belief Nets*. Neural Computation, 18(7):1527–1554.

[Jensen et al., 2007] Jensen, Kurt, L. M. Kristensen und L. Wells (2007). *Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems*. Int. J. Softw. Tools Technol. Transf., 9(3):213–254.

[MetraLabs GmbH et al., 2012a] MetraLabs GmbH, GERMANY, Neuroinformatics und G. Cognitive Robotics Labs (NICR), TU Ilmenau (2012a). *MIRA-Entwicklerdokumentation: Rich Client Platform*. Online, URL: `http://mira-project.org/MIRA-doc/RCPPage.html` (abgerufen am 22. Juni 2015).

[MetraLabs GmbH et al., 2012b] MetraLabs GmbH, GERMANY, Neuroinformatics und G. Cognitive Robotics Labs (NICR), TU Ilmenau (2012b).

MIRA-Entwicklerdokumentation: Serialisierungsframework. Online, URL: `http://mira-project.org/MIRA-doc/SerializationPage.html` (abgerufen am 22. Juni 2015).

[MetraLabs GmbH et al., 2012c] MetraLabs GmbH, GERMANY, Neuroinformatics und G. Cognitive Robotics Labs (NICR), TU Ilmenau (2012c). *MIRA-Entwicklerdokumentation: Units*. Online, URL: `http://mira-project.org/MIRA-doc/UnitsPage.html` (abgerufen am 22. Juni 2015).

[Moore, 1965] Moore, Gordon E. (1965). *Cramming more components onto integrated circuits*. Electronics, 38(8).

[Mueller et al., 2014a] Mueller, Steffen, S. Sprenger und H.-M. Gross (2014a). *OfficeMate – a Study of an Online Learning Dialog System for Mobile Assistive Robots*. In: *Int. Conf. on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE)*, S. 104–110.

[Mueller et al., 2014b] Mueller, Steffen, S. Sprenger und H.-M. Gross (2014b). *Online Adaptation of Dialog Strategies based on Probabilistic Planning*. In: *IEEE Int. Symp. on Robot and Human Interactive Communication (RO-MAN)*, S. 692–697. IEEE.

[Oh et al., 1988] Oh, S.J., D. Ahn und E. L. Hall (1988). *A wide angle vision system for mobile robots*. In: *Intelligent Control, 1988. Proceedings., IEEE International Symposium on*, S. 490–498.

[Rumbaugh et al., 2004] Rumbaugh, James, I. Jacobson und G. Booch (2004). *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education.

[Sha et al., 1990] Sha, L., R. Rajkumar und J. P. Lehoczky (1990). *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*. IEEE Trans. Comput., 39(9):1175–1185.

[Stricker, 2011] Stricker, Ronny (2011). *SERROGA-Projektseite*. Online, URL: http://www.tu-ilmenau.de/fileadmin/media/neurob/research/robots/tweety/Tweety_2011_6_NIKR_Foto_Stricker.jpg (abgerufen am 26. Juni 2015).

[Stricker et al., 2012] Stricker, Ronny, S. Müller, E. Einhorn, C. Schröter, M. Volkhardt, K. Debes und H.-M. Gross (2012). *Konrad and Suse, Two Robots Guiding Visitors in a University Building*. In: *AMS'12*, S. 49–58.

[W3C, 2005] W3C, World Wide Web Consortium, Barnett Jim et. al. (2005). *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. Online, URL: http://www.w3.org/TR/2005/WD-scxml-20050705/, aktuelle Fassung jeweils unter: http://www.w3.org/TR/scxml/ (abgerufen am 7. Mai 2015).

[Weinrich, 2015] Weinrich, Christoph (2015). *Ausgewählte Aspekte einer sozialverträglichen Mensch-Roboter-Interaktion in öffentlichen Einsatzumgebungen*. Doktorarbeit, TU Ilmenau.

[Yujin-Robot et al., 2014] Yujin-Robot, O. S. R. Foundation und A. Learning Agents Research Group, University of Texas (2014). *Robotics-in-Concert-Wiki*. Online, URL: http://www.robotconcert.org/index.php/Main_Page (abgerufen am 23. Juni 2015).

Thesen

- Die Nutzung eines dynamischen Konzeptes zur Synchronisation und Verwaltung modularer Applets eignet sich zur Ablaufsteuerung in robotischen Applikationen.
- Die dynamische Ablaufsteuerung ist in der Lage, eine derartige Synchronisation und Verwaltung vorzunehmen.
- Die dynamische Ablaufsteuerung toleriert definierte Fehlerfälle und bietet Mechanismen zur gesonderten Fehlerbehandlung.
- Konfligierende Aktionen werden, sofern eine Möglichkeit dazu existiert, automatisiert aufgelöst, nicht auflösbare Konflikte als solche erkannt.
- Bei Modifikation und Erweiterung eines vorhandenes Systems führt eine Implementierung auf Basis der dynamischen Ablaufsteuerung zu Aufwandsreduktion und Zeitersparnis im Vergleich zu State-of-the-Art-Ansätzen.

Ilmenau, 02.07.2015 .....

Alexander Katzmann