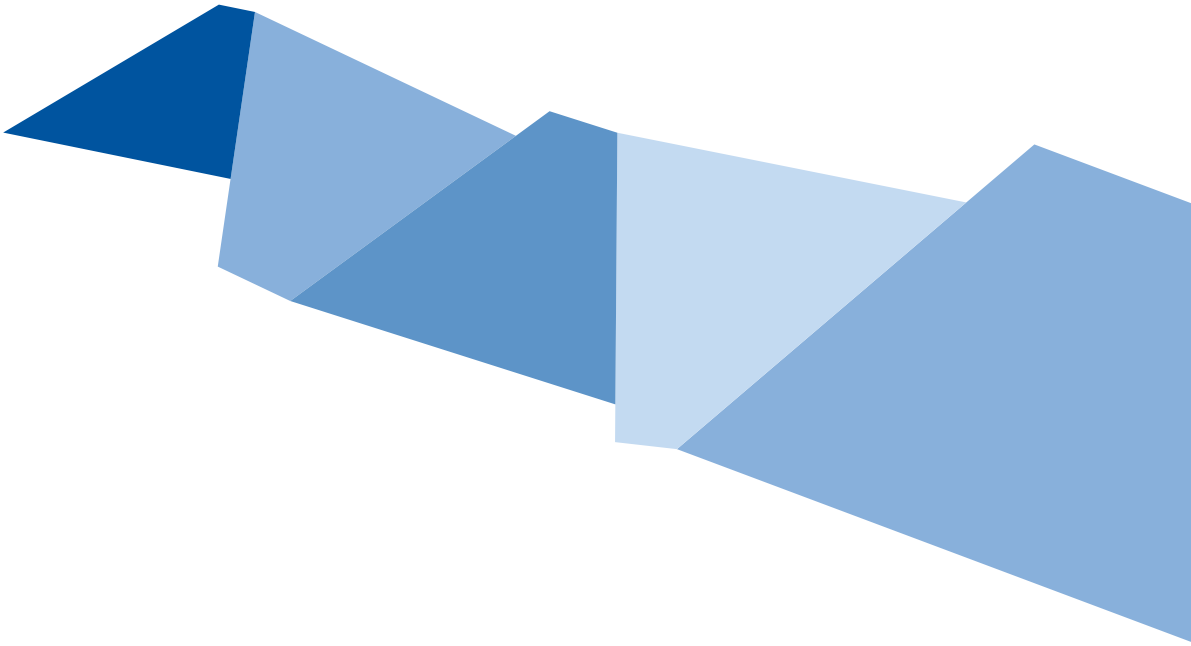


Adaptierung des Zeitverhaltens nicht-echtzeitfähiger Software für den Einsatz in zeitheterogenen Netzwerken

John F. Schommer

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2020-03
RWTH Aachen University | Department of Computer Science | Mai 2020

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Adaptierung des Zeitverhaltens nicht-echtzeitfähiger Software für den Einsatz in zeitheterogenen Netzwerken

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
John F. Schommer
aus
Mönchengladbach

Berichter: Professor Dr.-Ing. Stefan Kowalewski
Professor Dr. rer. nat. Bernhard Rumpe

Tag der mündlichen Prüfung: 24. Juni 2019

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

John F. Schommer
Lehrstuhl Informatik 11
schommer@embedded.rwth-aachen.de

Aachener Informatik Berichte AIB-2020-03

Herausgeber: Fachgruppe Informatik
RWTH Aachen Universität
Ahornstraße 55
52074 Aachen
Germany

ISSN 0935-3232

Der besten Frau der Welt
und unseren wundervollen Kindern

Zusammenfassung

Eingebettete Systeme werden in immer höherem Maße vernetzt. Dabei kommunizieren echtzeitfähige mit nicht-echtzeitfähigen Systemen, ohne dass dabei die zeitkritischen Anforderungen aufgegeben werden. Diese Anforderungen werden dann über das Netzwerk auf andere Netzwerkknoten weitergegeben. Erfüllen jedoch diese Netzwerkknoten nicht die propagierten Anforderungen, können im Umkehrschluss die vernetzten eingebetteten Systeme nicht mehr garantieren, dass ihre spezifische Hauptfunktion korrekt ausgeführt wird. Es wird eine Vorgehensweise benötigt, die eine Adaptierung von nicht-echtzeitfähigen Netzwerknoten ermöglicht, um die Funktion der mit ihnen vernetzten zeitkritischen Systeme nicht zu gefährden.

Die vorgelegte Arbeit befasst sich mit der Software solcher Netzwerkknoten, auf die zeitliche Anforderungen einerseits propagiert werden, andererseits aber beim Entwurf nicht berücksichtigt wurden. Es werden Methoden zur Evaluierung und Adaptierung dieser Software vorgestellt, welche das Ziel haben, die Konformität der entsprechenden Netzwerkknoten gegenüber den propagierten zeitkritischen Anforderungen zu erhöhen.

Die dazu vorgestellte Vorgehensweise beinhaltet eine Konformität basierende Überprüfung physikalischer Signale sowie eine Pfadüberdeckung basierend auf Metriken über Lebenszyklen der Software. Eine anschließende Adaptierung unter Berücksichtigung der Evaluierungsergebnisse wird durch fünf Entwurfsmuster zur Refaktorierung der Software unterstützt. Drei dieser Entwurfsmuster wurden hauptsächlich für Knoten-zu-Knoten Kommunikation realisiert, die drei anderen für den Einsatz innerhalb von Netzwerkknoten entwickelt. Sowohl Evaluierung als auch Adaptierung der Software können iterativ durchgeführt werden. Eine Adaptierung ist dann erfolgreich, wenn die Software zeitkritische Anforderungen nun erfüllt; überprüft zum Beispiel über Konformität eines Ausgabesignals mit einem Referenzsignal.

Im Rahmen dieser Arbeit werden existierende Evaluierungsmethoden passend strukturiert und neu entwickelte Entwurfsmuster vorgestellt. Zusätzlich werden auch Anwendungsfälle und zeitkritische Anforderungen in den Kategorien Scheduling, Speicherverwaltung und Zeitgebern der Softwareplattformen katalogisiert, um diese für eine Evaluierung strukturiert einsetzen zu können. Die Arbeit zeigt, wie Testfälle als Grundlage dieser Evaluierung aus diesen Katalogen generiert werden können, und stellt außerdem Metriken vor, um die Laufzeitergebnisse der Software zu beurteilen.

In Fallstudien und Kooperationsprojekten mit der Industrie wurden die entwickelten Methoden geprüft und Softwareprototypen entwickelt, um die Anwendbarkeit der entwickelten Methoden innerhalb etablierter Entwicklungsprozesse zu zeigen.

Abstract

Embedded systems are becoming increasingly connected in time-heterogeneous networks. Communicating with non-real-time systems without changing or deleting the often time-critical requirements. New is here the mix up of real-time and non-real-time requirements within one network. The time-critical requirements are then propagated across the network to other nodes. That is, passed while communication is done at runtime between real-time and non-real-time systems. However, if non real-time systems do not meet the propagated requirements, the conncected real-time systems can no longer guarantee that their specific main function is correctly executed. A methodology is needed that enables a structured evaluation and adaptation of non-real-time systems, in the case that the main function of the real-time systems is not jeopardized.

The presented work deals with the software of such network nodes, which time-critical requirements are propagated on the one hand; where on the other hand, the requirements were not taken into account in the original design of the systems.

Methods for the evaluation and adaptation of the software of these systems are presented, which aim to increase the fulfilment of the corresponding network nodes with the propagated time-critical requirements. These methods can also be used iteratively in agile development processes in an automated way, which are also being established today in the development of embedded systems.

The evaluation is carried out on existing software or software freshly implemented. The multistep approach to this involves conformance-based validation of signals and path coverage with metrics based on software lifecycles. The subsequent adaptation should take into account the evaluation results and is supported via five design patterns on software in this thesis. Three of these design patterns have been developed mainly for node-to-node communication, the other two design patterns for use within network nodes. The evaluation and adaptation of the software can be carried out repeatedly. If the software under adaptation achieves a certain quality, for example a signal conformity to a certain reference signal, corresponding network nodes can be installed with real-time systems or on real-time platforms after completion of the adaptation in such a way that the specific main function of the connected real-time systems is executed correctly in time-critical terms. Also, use cases and time-sensitive requirements are catalogued in aspects of scheduling, memory management, and software platform timers. The thesis shows how test cases can be generated as the basis of an evaluation from these catalogues, and also provides metrics to assess the runtime results of the software.

Case studies and collaborative projects with industry have examined the developed methods and software prototypes to demonstrate the applicability of them within established development processes.

Danksagung

An erster Stelle bedanke ich mich bei Herrn Prof. Dr.-Ing. Stefan Kowalewski für die Möglichkeit, an seinem Lehrstuhl promovieren und diese Dissertation schreiben zu können, insbesondere aber für seine Geduld bis heute. Ich danke Herrn Prof. Dr. rer. nat. Bernhard Rumpe für seinen Aufwand als zweiter Berichter, Herrn Prof. Dr. rer. nat. Peter Rossmanith als Prüfer und Herrn Prof. Dr. rer. nat. Martin Grohe als Vorsitzenden der Promotionskommission. Die Promotion wurde durch das UMIC Excellence Cluster gefördert und ich danke hier Dr. rer. nat. Carsten Weise als meinem persönlichen Betreuer.

Vielen Dank gilt allen Kollegen des Lehrstuhls Informatik 11 der RWTH Aachen Universität. Um einige zu nennen, danke ich Dr. rer. nat. Dominik Franke und Dr.-Ing. Ashraf Armoush für unsere gute Zusammenarbeit, sowie Dr.-Ing. Andre Stollenwerk und Andreas Derks für ihre Unterstützung. Ich muss mich bei allen studentischen Mitarbeitern und Abschlussarbeitern bedanken, insbesondere bei Mareike Bültmann, Dominik Schmithausen, Tim Lange, Jan Garcia und Dr. rer. nat. Thomas Gerlitz für ihre Mitwirkung an den Ergebnissen der vorliegenden Arbeit. Meinen besonderen Dank schulde ich Eva Beckschulze, Marina Herkenrath und Karin Vonderstein für eine konstant herzliche Arbeitsatmosphäre.

Ich bedanke mich bei meinen Freunden David Thönnessen, Herwig Linß und Dr. rer. nat. Dirk Wilking, die mich über die Jahre getrieben haben, diese Dissertation fertig zu stellen. Zu guter Letzt danke ich meiner Familie, namentlich Linus, Maya, Charlotte und allen voran Tabea Schommer, welche mich jahrelang begleitet haben.

John F. Schommer
Heinsberg, Mai 2020

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Echtzeitfähigkeit ist oftmals eine essentielle Anforderung an eingebettete Systeme. In determinierter Zeit soll eine Ausgabe periodisch oder als Reaktion auf ein Eingabeereignis berechnet werden. Dieser zeitliche Determinismus ist die Garantie dafür, dass die geforderten eingebetteten Systeme ihre Hauptfunktion, das Steuern und Überwachen eines zeitkritischen technischen Kontextes, erfüllen können. Solche eingebetteten Systeme nennt man Echtzeitsysteme [3, 11, 82]. Echtzeitsysteme werden einerseits in sicherheitskritischen Bereichen zum Beispiel in Flugzeugen, Automobilen oder der Medizintechnik eingesetzt, andererseits dort, wo großer betriebswirtschaftlicher Schaden durch zeitliches Fehlverhalten zum Beispiel beim Einsatz in Fertigungsanlagen zu erwarten ist.

In den traditionellen Einsatzgebieten von Echtzeitsystemen wird der Bedarf an neuer, erweiterter Funktionalität immer größer [41, 58]. Deswegen werden mehrere Echtzeitsysteme für verschiedene Aufgaben in einen einzelnen technischen Kontext verbaut. Viele dieser Systeme müssen miteinander vernetzt werden, um übergeordnete Funktionen zu realisieren. Typische Beispiele aus der Automobiltechnik sind Fahrassistenzsysteme oder Motorensteuerung. Dieser Trend zu mehr Systemen und mehr Vernetzung wird durch den steigenden Ressourcenaufwand beschränkt. Ferner muss zum Beispiel in der Automobiltechnik auch der Platzbedarf und in der Avionik das Gewicht der verwendeten Teile berücksichtigt werden.

Unabhängig von der Art der Vernetzung werden aber keine lokalen, oftmals zeitkritischen Anforderungen an die einzelnen Echtzeitsysteme abgeschwächt oder aufgegeben [64, 82, 83]. Kommunizieren also vernetzte Echtzeitsysteme zur Laufzeit mit anderen Systemen innerhalb desselben Netzwerks, werden die zeitkritischen Anforderungen an diese Echtzeitsysteme durch Datenfluss und Kontrollfluss auf die anderen Netzwerkteilnehmer propagiert, in dem Sinne, dass diese Netzwerkteilnehmer die Erfüllung der immer noch lokalen zeitkritischen Anforderungen negativ beeinflussen. Datenpakete müssen deterministisch versendet und verarbeitet werden, um verspätete Datenverarbeitung oder sogar Datenverlust zu vermeiden. Kritische Ereignisse müssen fristgerecht kommuniziert werden. Allerdings entsprechen die anderen Netzwerkteilnehmer nicht a priori den propagierten Anforderungen. Je nach Aufgabe innerhalb des technischen Kontextes werden beim Entwurf anderer Netzwerkteilnehmer keine oder nur schwache zeitliche Anforderungen berücksichtigt. Ist ein Echtzeitsystem mit einem solchen Netzwerkteilnehmer

vernetzt und es findet zur Laufzeit Kommunikation statt, kann es passieren, dass dieses Echtzeitsystem seinen zeitkritischen Anforderungen nicht mehr entsprechen und damit seine Hauptfunktion nicht mehr erfüllen kann. Zum Beispiel könnte ein entsprechender Netzwerkteilnehmer ausfallen oder bereits andere Informationen verarbeiten. Dann werden Daten als Eingabe an das Echtzeitsystem nicht fristgerecht oder regelmäßig genug gesendet. Oder es werden Ausgaben des Echtzeitsystems nicht schnell genug verarbeitet, so dass es zu Pufferüberläufen und damit einhergehenden Datenverlust kommt [11, 82].

Sowohl durch vernetzte Echtzeitsysteme als auch im Kontext von Echtzeitplattformen können zeitkritische Anforderungen nicht ohne weiteres erfüllt werden, wenn entweder nicht-echtzeitfähige Kommunikation mit diesen stattfindet oder die Leistungsstärke der Plattformen der wachsenden Anforderungen nicht gewachsen sind. Diese Problemstellung wurde während der Durchführung von drei Kooperationsprojekten des Lehrstuhls Informatik 11 mit verschiedenen industriellen Partnern identifiziert und untersucht. In der Automobilindustrie wurden Werkzeuge für die Realisierung komplexer Motorensteuerungen gesucht, die einerseits zeitkritische Systeme ausmessen und andererseits komplexe Softwaresysteme zur Simulation und Benutzerinteraktion bereitstellen müssen. In der Licht- und Signaltechnik sollten Hardware-in-the-Loop Tests durch Softwaresimulation substituiert werden. Die Softwaresimulation muss dabei mit den Echtzeitsteuerungen der Signaltechnik zeitkritisch kommunizieren können. In der Avionikindustrie sollten im Vorfeld einer Hardwareherstellung spezifische Anforderungen an diese Hardware ermittelt werden. Dazu werden die entwickelten Algorithmen mit einem adaptierten Software-in-the-Loop Ansatz, das auch zeitliche Anforderungen austesten kann, analysiert. Basierend auf den Erfahrungen aus den beiden ersten Industriekooperationen wurde nach Abschluss des Projekts *Adaptive Protocol Stacks* [16, 56, 57] aus dem UMIC[1] Excellence Cluster ein Folgeprojekt aufgesetzt. Der inhaltliche Schwerpunkt wurde dabei von den Kommunikationsprotokollen auf die Software der Netzwerkknoten verallgemeinert. Das Kooperationsprojekt mit der Avionikindustrie fand im Wesentlichen dem zweiten Forschungsprojekt nachgelagert statt.

Die in dieser Arbeit untersuchte Problemstellung eines zeitheterogenen, Definition siehe Seite 24 Netzwerkes lässt sich grundsätzlich vermeiden, wenn zeitkritische Anforderungen und deren Propagation bereits beim Entwurf einer entsprechenden Infrastruktur wie folgt berücksichtigt werden.

- *Gewährleistung einer netzwerkweiten Echtzeitfähigkeit.* Beim Entwurf des Netzwerks werden entsprechend hohe zeitkritische Anforderungen an die Infrastruktur und alle Netzwerkknoten gestellt. Dadurch entstehen allerdings hohe Kosten und lange Vorlaufzeiten, da das Programmieren von Echtzeitplattformen auch heutzu-

[1] *Ultra High Speed Mobile Information and Communication*

tage noch immer schwierig ist. Es ist anzunehmen, dass der Schweregrad sich dabei stetig im Vergleich zur Netzwerkgröße erhöht. Außerdem gibt es technische Probleme, die eine echtzeitfähige Umsetzung behindern; wie zum Beispiel Schnittstellen zur Mensch-Maschine-Interaktion oder paketbasierte Netzwerkdienste.

- *Unidirektionaler Datenverkehr.* Daten können aus Richtung Echtzeitsystem in das restliche Netzwerk versendet werden. Sollte es dabei zu Datenüberläufen kommen, werden Daten im Netzwerk einfach mit einer passenden Heuristik überschrieben. Das bedeutet aber, dass keine Laufzeitdaten aus dem Netzwerk für den Einsatz des Echtzeitsystems essentiell sein dürfen, in solchen Netzwerken keine zeitkritischen Anforderungen propagiert werden können und Kontrollfluss zwischen den Systemen ausgeschlossen ist. Da Echtzeitdaten verloren gehen können, dürfen diese Daten ebenfalls nicht wesentlich für die Funktion des restlichen Netzwerks sein.

- *Disjunkte Infrastrukturen.* Echtzeitsysteme und nicht-echtzeitfähige Systeme werden in zwei separaten Netzwerken verbaut, was im Allgemeinen technisch aufwendig ist. Einerseits kann dadurch der Aufwand zur Realisierung einer echtzeitfähigen Infrastruktur begrenzt werden, andererseits werden viele Funktionen wie zum Beispiel Mensch-Maschine-Interaktion möglich. Dabei findet aber prinzipiell keine Kommunikation zwischen den beiden Infrastrukturen statt. Eine unidirektionale Kommunikation von Echtzeitdaten in Richtung der nicht-echtzeitfähigen Systeme ist dabei optional möglich.

Die genannten Ansätze erfordern einen hohen Aufwand in Zeit und Ressourcen. Der Aufwand lässt sich nur durch entsprechende topologische als auch funktionale Einschränkungen reduzieren. Deswegen ist die vorgestellte Problemstellung heutzutage allgegenwärtig in der Entwicklung eingebetteter Systeme und relevanter Gegenstand der Forschung.

1.1. Zielsetzung

Ziel der vorgelegten Arbeit ist es, eine Verbesserung von a priori nicht-echtzeitfähigen Netzwerkknoten gegenüber den auf diesen Knoten propagierten zeitkritischen Anforderungen zu ermöglichen. Dazu befasst sich die Arbeit mit der Software solcher Netzwerkknoten, auf die zeitkritischen Anforderungen einerseits propagiert werden, andererseits aber im ursprünglichen Entwurf der Software nicht oder nicht strikt genug berücksichtigt wurden. Es werden Methoden zur Evaluierung und Adaptierung dieser Software vorgestellt. Wird durch den Einsatz dieser Methoden eine entsprechende Verbesserung erreicht, lassen sich diese Netzwerkknoten mit Echtzeitsystemen oder auf Echtzeitplattformen so vernetzen, dass propagierte zeitkritischen Anforderungen innerhalb des Netz-

werks erfüllt werden und die spezifische Hauptfunktion der vernetzten Echtzeitsysteme zeitlich korrekt ausgeführt wird.

Software für eingebettete Systeme wird anders entwickelt als Software, die nicht in einem technischen Kontext verbaut wird. Nicht-funktionale Anforderungen, die sich aus diesem technischen Kontext ergeben, müssen in Entwicklungsprozessen für Software im Allgemeinen als auch speziell für eingebettete Software analysiert, spezifiziert und die Software so umgesetzt werden, dass diesen Anforderungen entsprochen wird. Die Vorgehensweise während einer Anforderungsanalyse ist gerade für traditionelle Echtzeitsysteme im Detail dokumentiert. Eine auf die Problemstellung zugeschnittene Vorgehensweise zur Evaluierung betroffener nicht-echtzeitfähiger Netzwerkknoten oder Anteilen einer Echtzeitplattform ist dagegen noch nicht methodisch verfügbar. Ein weiteres Ziel der vorgelegten Arbeit ist es deshalb, Methoden zur Evaluierung des Datenflusses und Kontrollflusses zwischen echtzeitfähiger und nicht-echtzeitfähiger Software innerhalb eines Netzwerks oder auf einer Plattform zur Verfügung zu stellen.

Software für eingebettete Systeme ist so vielfältig wie der technische Kontext, in dem diese verbaut werden. Dabei werden unterschiedlichste Softwarearchitekturen und Kommunikationsprotokolle eingesetzt. Ziel der vorgelegten Arbeit ist es deshalb, flexibel einsetzbare Entwurfsmuster [19] zur Adaptierung nicht-echtzeitfähiger Netzwerkknoten vorzugeben. Nach einer Auswahl passender Entwurfsmuster, zum Beispiel mit Hilfe der in dieser Arbeit vorgestellten Methoden zur Evaluierung, können diese dazu verwendet werden, den Konformitätsgrad von Netzwerkknoten bezüglich propagierter zeitkritischer Anforderungen zu erhöhen.

Mobile Endgeräte sind heutzutage allgegenwärtig. Grundsätzlich sind solche mobilen Endgeräte drahtlos-vernetzte eingebettete Systeme mit dem Schwerpunkt Mensch-Maschine-Interaktion. Ziel der vorgelegten Arbeit ist es deshalb auch, aktuelle mobile Technologien in die Problemstellung mit einzubeziehen. Im Rahmen des UMIC Excellence Clusters werden in dieser Arbeit einzelne Betriebssysteme und Plattformen von mobilen Geräten untersucht.

1.2. Beitrag

Die Arbeit liefert die folgenden wissenschaftlichen Beiträge:

Evaluierung von Software und Softwareplattformen

Die vorgelegte Arbeit stellt eine auf die beschriebene Problemstellung zugeschnittene Vorgehensweise zur Evaluierung von Software und Softwareplattformen vor, nicht notwendig auf zeitkritische Systeme beschränkt, aber zugeschnitten. Mit dieser Vorgehens-

weise können zeitliche Eigenschaften insbesondere von nicht-echtzeitfähigen Netzwerkknoten in zeitkritischen Netzwerken oder Software als Anteil einer Echtzeitplattform methodisch evaluiert werden. Die Vorgehensweise beinhaltet die folgenden Konzepte: Konformitätsüberprüfung physikalischer Signale aus dem und in das zu testende System, eine neue Methode zur lebenszyklusabdeckenden Operationalisierung, um die Konformität messbar zu machen, der sowie ein Rahmenwerk zur Testfallgenerierung anhand zeitkritischer Anforderungen.

Die Arbeit liefert außerdem eine Übersicht von Metriken zur Beurteilung von Laufzeitverhalten der zur evaluierenden Software oder Softwareplattformen.

Systematisierte Anforderungen und Anwendungsfälle

Die vorgelegte Arbeit liefert sowohl einen Katalog aus exemplarischen Anwendungsfällen, die entsprechend der beschriebenen Problemstellung verwendet werden können, als auch einen Katalog aus zeitkritischen Anforderungen. Diese Anforderungen sind systematisch erfasst und nach Scheduling, Speicherverwaltung und Zeitgebern der Softwareplattformen kategorisiert. Die Anforderungen sind ebenfalls den Anwendungsfällen zugeordnet, mit deren Hilfe sich Testfälle beziehungsweise Testfallklassen ableiten lassen.

Entwurfsmuster zur Softwareadaptierung

Die vorgelegte Arbeit liefert fünf auf die beschriebene Problemstellung zugeschnittene Entwurfsmuster [19] zur Adaptierung von Software beziehungsweise Softwareplattformen.

- Das Entwurfsmuster *Decoupling Datastructure* entkoppelt Schichten einer Softwarearchitektur voneinander und erlaubt unidirektional echtzeitfähigen Datenzugriff.
- Das Entwurfsmuster *Active Cross-Layer Balancing* verlagert den Datenzugriff in die Schichten einer Softwarearchitektur und balanciert aktiv den Datendurchsatz zwischen diesen Schichten. Der Datenzugriff wird dabei asynchron verzögert.
- Das Entwurfsmuster *Adaptive Controlflow Transitions* ermöglicht einen echtzeitfähigen Wechsel des Netzwerkbetriebs auf allen Netzwerkknoten. Insbesondere erfolgt die Freigabe von Ressourcen unmittelbar.
- Das Entwurfsmuster *Resume on Exception* behandelt Ausnahmen in Netzwerkknoten unter Berücksichtigung des netzwerkweiten Betriebszustands. Die Ausnahmeroutine muss dabei die lokale Ausführung nicht mehr grundsätzlich abbrechen.

- Das Entwurfsmuster *Virtual Real-time* virtualisiert zeitliches Verhalten. Innerhalb der virtuellen Plattform können Programme echtzeitfähig ausgeführt werden.

Die ersten drei Entwurfsmuster wurden hauptsächlich für Softwarearchitekturen entwickelt, die Knoten-zu-Knoten-Kommunikation realisieren. Die letzten beiden Entwurfsmuster wurden für den Einsatz in Softwarearchitekturen auf nur einem Knoten entwickelt.

Fallstudien zu verschiedenen Plattformen

Drei Fallstudien wurden durchgeführt und bewerten die vorgestellten Methoden zur Evaluierung von Software und Softwareplattformen. Dabei wurden ein Linux Betriebssystem, eine Unix-basierte Mittelschicht und eine Erweiterung für Microsoft Windows™ mit Hilfe dieser Methoden auf Echtzeitfähigkeit hin untersucht.

Werkzeuge und Prototypen

Während der Durchführung des Kooperationsprojektes mit einem Unternehmen der Automobilindustrie wurde ein Werkzeug zur Konformitätsüberprüfung physikalischer Signale entwickelt. Die Entwurfsmuster Decoupling Datestructure und Active Cross-Layer Balancing entstanden für eine Softwaresimulation für die Licht- und Signaltechnik. Das Entwurfsmuster Virtual Real-time wurden in einen Software-in-the-Loop Prototypen verbaut. Weiterhin wurde das Betriebssystem *EvalOS* für Mikrocontroller entwickelt, das ein Testrahmenwerk beinhaltet mit dem Benutzerprozesse ausgewertet werden können.

1.3. Struktur

Das folgende Kapitel 2 zeigt diejenigen technischen Grundlagen auf, welche im weiteren Verlauf der vorliegenden Arbeit als bekannt vorausgesetzt werden müssen. Im Anschluss werden die Beiträge der vorliegenden Arbeit diskutiert. Im Kapitel 3 wird die Motivation vertieft und die Beiträge in einen technischen Kontext gesetzt. Im Kapitel 4 werden die Evaluierungsmethoden vorgestellt und dann, im Kapitel 5, die Entwurfsmuster zur Adaptierung von Software diskutiert. Das Kapitel 6 rundet die Diskussion ab.

Im Anhang der vorliegenden Arbeit finden sich katalogisierte Anwendungsfälle und eine erweiterte Liste mit zeitkritischen Anforderungen, die in den vorherigen Kapiteln verwendet wurden. Verwandte Arbeiten anderer Autoren werden in den entsprechenden Kapiteln und Abschnitten diskutiert und jeweils in Bezug zu den Beiträgen gesetzt.

1.4. Eigene Vorarbeiten

Die vorgelegte Arbeit basiert auf Vorarbeiten, die im Rahmen der Kooperationsprojekte mit der Industrie stattfanden oder in früheren Publikationen veröffentlicht wurden.

Mit der Automobilindustrie wurde die Konformitätsüberprüfung physikalischer Signale entwickelt. Für diese Arbeit wurde dazu ein Algorithmus und eine dazu passende Visualisierung implementiert, vorgestellt in Abschnitt 3.3. Möglichkeiten zur Evaluierung zeitkritischer beziehungsweise mit zeitkritischen Systemen vernetzter Software wurde in einer Tauglichkeitsstudie [4] erarbeitet, und dann sowohl in einer Veröffentlichung als Hauptautor [63] als auch Nebenautor [15] weiter vertieft. Eine Fallstudie dazu wurde im Rahmen einer weiteren Abschlussarbeit [59] erarbeitet. Der Einsatz nicht-echtzeitfähiger Softwareplattformen für zeitkritische Anwendungen wurde in zwei Serien kleiner Softwareprojekte [25] untersucht, innerhalb des Praktikumsbetriebs am Lehrstuhl Informatik 11. Die Entwurfsmuster Decoupling Datastructure [59] und Active Cross-Layer Balancing [39, 64] wurden erst im Rahmen von Abschlussarbeiten entwickelt, dann in einer Softwaresimulation für die Licht- und Signaltechnik Industrie verbaut und publiziert [64]. Die Adaptierung mobiler, nicht-echtzeitfähiger Softwareplattformen wurde erst in [31, 32] veröffentlicht und durch eine Fallstudie [76] vertieft. Ein Ansatz zur Evaluierung mobiler Applikationen wurde ebenfalls als Nebenautor vorgestellt [17]. Das Entwurfsmuster Adaptive Controlflow Transitions [62] wurde prototypisch für die vorliegende Arbeit entwickelt. Die Entwurfsmuster Resume on Exception und Virtual Real-time wurden im Rahmen der Kooperation mit der Avionikindustrie selbstständig am Lehrstuhl entwickelt.

2. Voraussetzungen

In diesem Kapitel werden einige technische Grundlagen vorgestellt, die im weiteren Verlauf der vorgelegten Arbeit als bekannt vorausgesetzt werden. Im ersten Abschnitt werden nicht-funktionale Anforderungen eingeführt. Dabei liegen die Schwerpunkte auf zeitkritischen Anforderungen 2.1.1 und die Umsetzung in Echtzeitsoftware 2.1.2. Danach stellt Abschnitt 2.2 Schedulingmechanismen vor und Abschnitt 2.3 die Umsetzung dieser Mechanismen im Kernel von Derivaten des Betriebssystems Linux. In Abschnitt 2.4 werden schließlich Kommunikationsmechanismen unter der Berücksichtigung zeitkritischer Anforderungen eingeführt. Der letzte Abschnitt befasst sich mit dem Testen von Software für eingebettete Systeme.

2.1. Nicht-funktionale Anforderungen

Echtzeit gehört zu den so genannten nicht-funktionalen Anforderungen [55, 82]. Diese Anforderungen müssen von eingebetteten Systemen erfüllt werden, um ihre Hauptfunktion zu erfüllen. Dabei können Eigenschaften der Software für eingebettete Systeme, die nicht-funktionale Anforderungen realisieren, nicht ohne Referenzwerte bewertet werden. Aus diesem Grund werden diese Eigenschaften operationalisiert, um mit Hilfe einer Metrik und einer Instrumentalisierung verfügbarer Messdaten eine Beurteilung der Software durchführen zu können.

2.1.1. Genauigkeit, Präzision und Rechtzeitigkeit

Die Präzision eines technischen Systems bezeichnet den Grad zu welchem wiederholte Berechnungen unter denselben Eingaben und Bedingungen zum selben Ergebnis führen. Das wird auch Reproduzierbarkeit oder Wiederholbarkeit genannt. Die Genauigkeit eines technischen Systems ist dagegen die Abweichung der berechneten Werte vom Erwartungswert. Eine einfache Messung zum Beispiel misst ein stetiges Signal und führt zu einem systematischen Fehler. Die Eliminierung des systematischen Fehlers erhöht die Genauigkeit der Messung ohne die Präzision zu beeinflussen. Computersysteme können genau und präzise sein, nur eines davon oder keine dieser Eigenschaften haben.

Rechtzeitigkeit liegt dann vor, wenn ein Signal spätestens zu einem vorher bestimmbaren Zeitpunkt eintrifft. Rechtzeitigkeit ist eine weitere Anforderung neben Präzision und Genauigkeit, wird aber durch diese Faktoren direkt beeinflusst. Eine unterliegende Hardware, zum Beispiel ein Zeitgeber, muss gemessen in Zyklen genügend genau arbeiten, um die Rechtzeitigkeit der Software gewährleisten zu können. Präzision wird in der vorgelegten Arbeit nicht näher untersucht. Es wird außerdem angenommen, dass die funktionale Korrektheit der untersuchten Computersysteme im Vorfeld validiert wurde.

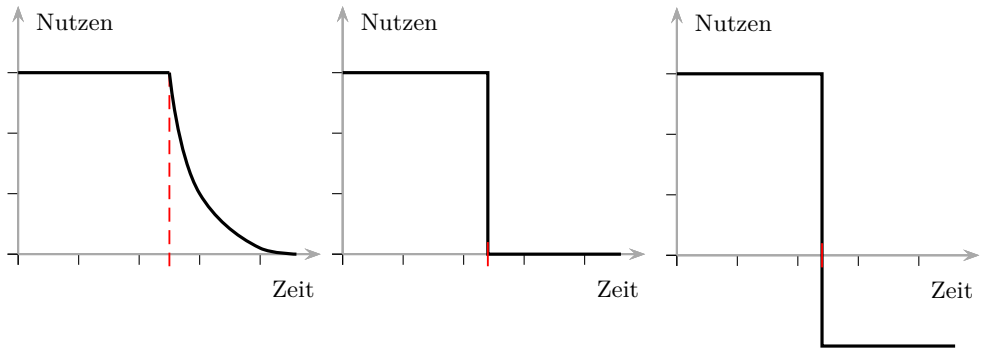
2.1.2. Echtzeitsoftware

Echtzeitsoftware muss die Rechtzeitigkeit von Berechnungen gewährleisten, welche durch die Software durchgeführt werden soll. In vorhersagbarer Zeit soll eine Ausgabe periodisch oder als Ergebnis eines Eingabeergebnisses berechnet werden. Häufig wird der Begriff Pünktlichkeit synonym zu Rechtzeitigkeit verwendet. Während Präzision und Genauigkeit 2.1.1 gemessen werden können, ist das für das Laufzeitverhalten von Echtzeitsoftware nicht ohne weiteres der Fall [82]. Das Laufzeitverhalten ist oftmals komplex und nicht vorhersagbar. Wird das Echtzeitsystem auf einer Softwareplattform ausgeführt, sind auch die Anforderungen Gleichzeitigkeit und Verfügbarkeit essentiell. Anhang A beschreibt im Kontext der vorgelegten Arbeit diese und weitere konkrete zeitkritische Anforderungen. Da diese Anforderungen als Eigenschaften von Echtzeitsystemen getestet werden müssen, werden zur Beurteilung von Echtzeitsoftware Metriken eingesetzt (2.1); der Vorgang wird Operationalisierung genannt. Für diese Operationalisierung ist die Granularität der zeitkritischen Anforderungen entscheidend. Ist diese zu fein, besitzt das Echtzeitsystem nicht die benötigte Rechenleistung. Ist diese zu grob, wird die Nützlichkeit der Berechnung reduziert. Dabei ist es üblich, dass zeitkritische Anforderungen in Kategorien eingeteilt werden, zum Beispiel

- Harte Echtzeit. Zeitkritische Anforderungen müssen eingehalten werden, da Schaden an Ausrüstung oder Personen verursacht wird
- Feste Echtzeit. Die Berechnungen werden nach Überschreiten der Zeitfrist wertlos, es droht jedoch kein unmittelbarer Schaden
- Weiche Echtzeit. Die Zeitfristen sind Richtlinien, die in einem entsprechenden Toleranzrahmen überschritten werden können

Wörn und Brinkschulte setzen eine Nützlichkeitsfunktion [82] ein, um die Kategorie des Laufzeitverhaltens von Echtzeitdatenverarbeitung zu erfassen. Mit Hilfe dieser Funktion kann das Laufzeitverhalten wie in Abbildung 2.1 beurteilt werden. Muss weiches Echtzeitverhalten realisiert werden, nimmt die Nützlichkeit der Berechnung in Abhängigkeit

des Laufzeitszenarios langsam ab. Bei fester Echtzeit wird die Berechnung direkt wertlos, wenn die Zeitfrist überschritten wird. Liegt harte Echtzeit als Anforderung vor, wird der Wert der Nützlichkeitsfunktion negativ.



(a) Weiche Echtzeit (b) Feste Echtzeit (c) Harte Echtzeit

Abbildung 2.1.: Wertfunktion zur Bewertung zeitkritischer Anforderungen [82]

Heutzutage bestehen Echtzeitsysteme aus vielen Komponenten und Funktionen; auf Echtzeitplattformen kommen verschiedene Applikationen zur Ausführung. Für jede dieser Komponenten oder Funktionen müssen konkrete zeitkritischen Anforderungen umgesetzt werden, um die Echtzeitfähigkeit des Gesamtsystems oder der Software zu realisieren.

2.1.3. Sicherheitskritische Systeme

Eingebettete Systeme werden oft in einem sicherheitskritischen Kontext eingesetzt, in dem die Sicherheit von Ausrüstung und Personen von dem Laufzeitverhalten der eingebetteten Systeme abhängt. Die Echtzeitfähigkeit ist eine der essentiellen Eigenschaften dieser Systeme, in denen nur eine rechtzeitige Berechnung eine korrekte Berechnung ist. Die Entwicklung vom ersten Entwurf bis zum Testen dieser Systeme ist deswegen vollkommen anders als von Standardsoftware (zum Beispiel werden spezielle Entwurfsmuster wie der Watchdog entwickelt). Für den Einsatz sicherheitskritischer Software muss eine entsprechende Zertifizierung erreicht werden [5, 6, 29, 60], deren Umfang zum Beispiel durch Normen vorgeben ist.

2.2. Schedulingmechanismen

Werden auf einer Softwareplattform mehrere Prozesse zur Ausführung gebracht, muss die Plattform die Ablaufplanung der Prozesse übernehmen. Dafür gibt es eine Reihe von Strategien, die in Abhängigkeit des Laufzeitszenarios eingesetzt werden. Dieser Abschnitt gibt einen Überblick über diejenigen Strategien, die im Rahmen der vorgelegten Arbeit relevant sind.

FIFO-Scheduling Der Schedulingmechanismus arbeitet mit einer *First-In-First-Out* Warteschlange. Wurde einem Prozess gerade Prozessorzeit gegeben, wird dieser Prozess zur weiteren Ablaufplanung am Ende der Warteschlange wieder eingereiht. Das Betriebssystem Linux zum Beispiel verwendet nicht-präemptives FIFO-Scheduling. Der geplante Prozess wird also so lange ausgeführt, bis dieser durch Aufruf der Methoden `exit()`, `yield()` oder durch das Warten auf eine `I/O`-Operation die zugeteilten Ressourcen freigibt. Neue Prozesse werden einfach an das Ende der Warteschlange eingereiht, vollständig abgearbeitete Prozesse aus der Warteschlange entfernt. Muss der Schedulingmechanimus einen neue Prozess auswählen, wird einfach der erste Prozess aus der Warteschlange ausgewählt.

Fair-Scheduling Ein *Completely-Fair* Scheduler [81] ist ein präemptiv arbeitender Schedulingmechanismus mit variablen Zeitscheiben. Die Strategie basiert auf einem *Rot-Schwarz-Baum* [7], geordnet nach der bisherigen tatsächlichen Laufzeit der einzelnen Prozesse. Der Scheduler aktualisiert regelmäßig diese Laufzeit und plant anschließend den Ablauf neu. Erwachen schlafende Prozesse, werde diese bei der Ressourcenvergabe ebenfalls fair behandelt. Die Planung erfolgt in zwei Schritten: Zuerst wird der gerade abgelaufene Prozess in den Rot-Schwarz-Baum eingefügt. Dann wird derjenige Prozess ausgewählt, der am weitesten unten links im Rot-Schwarz-Baum liegt.

Earliest-Deadline-First Scheduling Der *Earliest-Deadline-First* Scheduler organisiert die zu planenden Prozesse in einer Prioritätswarteschlange aufsteigend entsprechend der Zeitfrist dieser Prozesse. Diese Warteschlange wird immer dann aktualisiert, wenn ein neuer Prozess ausgewählt werden muss, um ausgeführt zu werden. Der Scheduler wählt dann den ersten in der Warteschlange wartenden Prozess aus. In Echtzeitsystemen wird oft auch die präemptive Variante des Schedulers eingesetzt.

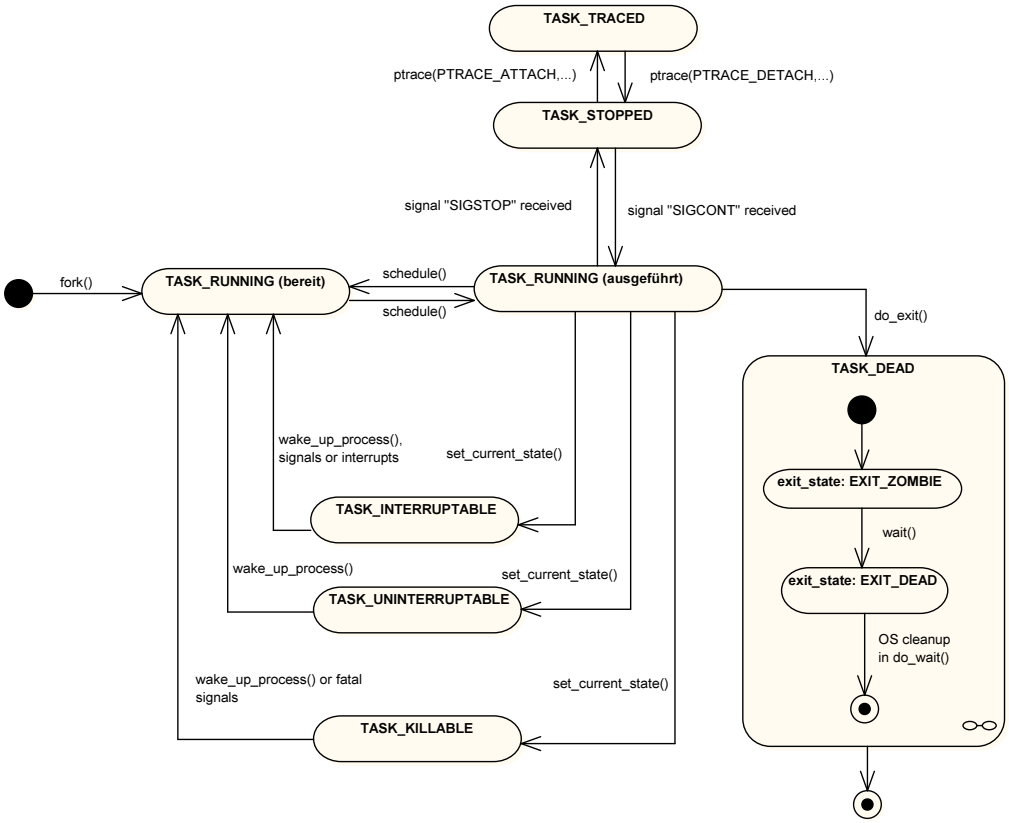


Abbildung 2.2.: Lebenszyklus eines Linuxprozesses

2.3. Linux Kernel und Lebenszyklen von Prozessen

Dieser Abschnitt beschreibt den Schedulingmechanimus im Kernel des Betriebssystems Linux, das im Rahmen der vorgelegten Arbeit in verschiedenen Fallstudien zur Evaluierung eingesetzt wurde. Abbildung 2.2 zeigt den Lebenszyklus von Prozessen, die durch den Schedulingmechanimus geplant werden. Dieser Lebenszyklus basiert auf der folgenden Vorgehensweise bei der Ablaufplanung. Im Kernel erfolgen Zugriffe auf die Zustandsvariablen der Prozesse nur direkt, gegebenenfalls über das `set_current_state` -Makro, was eine Überwachung der Zugriffe erschwert. Dagegen werden im Benutzermodus Prozesszustände gekapselt über Systemaufrufe veröffentlicht.

- Alle Linuxprozesse werden durch `fork()` beziehungsweise `do_fork()` erstellt und befinden sich dann im Zustand `TASK_RUNNING`. Diesen Zustand behalten die Prozesse während der Ausführung und wenn sie bereit zur Ausführung sind; der Linux Kernel unterscheidet diese Zustände in diesem Kontext nicht weiter. Prozesse in diesem Zustand können ihre Ressourcen auf drei verschiedene Arten freigeben.

- Im Zustand `TASK_UNINTERRUPTAIBLE` wird der Prozess schlafen gelegt, zum Beispiel durch Treibersoftware, die auf eine kurzfristige `I/O`-Operation warten muss. In diesem Zustand wird der Prozess nicht mehr durch den Scheduler geplant. Durch die Funktion `wake_up_process()` wird der Prozess wieder aufgeweckt und in den Zustand `TASK_RUNNING` versetzt.

- Der Zustand `TASK_INTERRUPTIBLE` legt den Prozess ebenfalls schlafen. Dieser Prozess kann aber durch entsprechende Signale wieder aufgeweckt werden. Die Signalverarbeitung erfolgt dann wieder im Zustand `TASK_RUNNING`.

- Ein Prozess im Zustand `TASK_KILLABLE` verhält sich wie ein nicht-unterbrechbarer Prozess, kann aber durch ein spezifisches Signal aufgeweckt werden, das zur Terminierung des Prozesses führt.

- Ein Prozess kann aus dem Zustand `TASK_RUNNING` über das Signal `SIGSTP` in den Zustand `TASK_STOPPED` versetzt werden. Dieser Prozess wird nicht mehr durch den Schedulingmechanimus eingeplant. Durch das Signal `SIGCONT` wird der Prozess wieder in den Zustand `TASK_RUNNING` versetzt.

- Befindet sich der Prozess im Zustand `TASK_STOPPED` wird über die Funktion `ptrace (PTRACE_ATTACH,...)` der Zustand auf `TASK_TRACED` gesetzt. Umgekehrt wird über die Funktion `ptrace(PTRACE_DETACH,...)` der Zustand wieder zurückgesetzt.

- Über die Funktion `do_exit()`, aufgerufen im Kernelmodus, wird der Prozess von `TASK_RUNNING` auf `TASK_DEAD` gesetzt. Dabei wird zuerst der Zwischenzustand `EXIT_ZOMBIE` gesetzt und der Prozess wird nicht mehr ausgeführt. Wird nun durch den Vaterprozess die Funktion `waitpid()` des sterbenden Prozesses aufgerufen, wird die Systemfunktion `wait()` ausgeführt und dadurch den Prozess in den finalen Zustand versetzt und dann freigegeben.

2.4. Kommunikation

In Netzwerken eingebetteter Systeme ist die zeitkritische Kommunikation zwischen den einzelnen Knoten essentiell wichtig. Dabei wird diese Kommunikation durch die verwendeten Algorithmen und deren Anzahl, sowie dem Verhalten der Netzwerkknoten bestimmt. Bei der Datenübertragung werden Daten durch einen Sender, zum Beispiel einen schreibenden Prozess, versendet und durch einen Empfänger, zum Beispiel einen lesenden Prozess, empfangen. Diese Datenübertragung kann einerseits unidirektional oder bidirektional erfolgen, andererseits kann mehr als ein Sender an einen Empfänger senden oder mehr als ein Empfänger von einem Sender empfangen. Die Umsetzung solch einer Datenübertragung wird durch die benötigte Infrastruktur und die konkreten nicht-funktionalen Anforderungen beeinflusst, gegebenenfalls erschwert. Im Fokus der vorgelegten Arbeit liegen unidirektionale Verbindungen.

2.4.1. Echtzeitfähige Kommunikation

Sind Echtzeitsysteme mit nicht-echtzeitfähigen Systemen vernetzt und werden keine zeitkritischen Anforderungen an die vernetzten Echtzeitsysteme abgeschwächt oder aufgegeben, gibt die Kommunikation beziehungsweise die Datenübertragung diese Anforderungen durch Datenfluss und Kontrollfluss auf die anderen Netzwerkteilnehmer weiter. Um die Hauptfunktion der Echtzeitsysteme nicht zu beeinträchtigen, müssen Datenfluss und Kontrollfluss entsprechend implementiert werden. Dabei stellt ein echtzeitfähiges System für einen nicht-echtzeitfähigen Kommunikationspartner kein Problem dar, da keine zeitkritischen Anforderungen an das Echtzeitsystem gestellt werden. Andersherum werden allerdings entsprechende Anforderungen auf das nicht-echtzeitfähige System weitergegeben und müssen beim Entwurf oder später bei der Adaptierung der Software beziehungsweise Softwareplattform berücksichtigt werden. Kommuniziert also ein Echtzeitsystem zum Beispiel über unidirektionale Datenübertragung mit einem nicht-echtzeitfähigen System, überträgt entweder ein echtzeitfähiger Sender an nicht-echtzeitfähige Empfänger Daten oder umgekehrt.

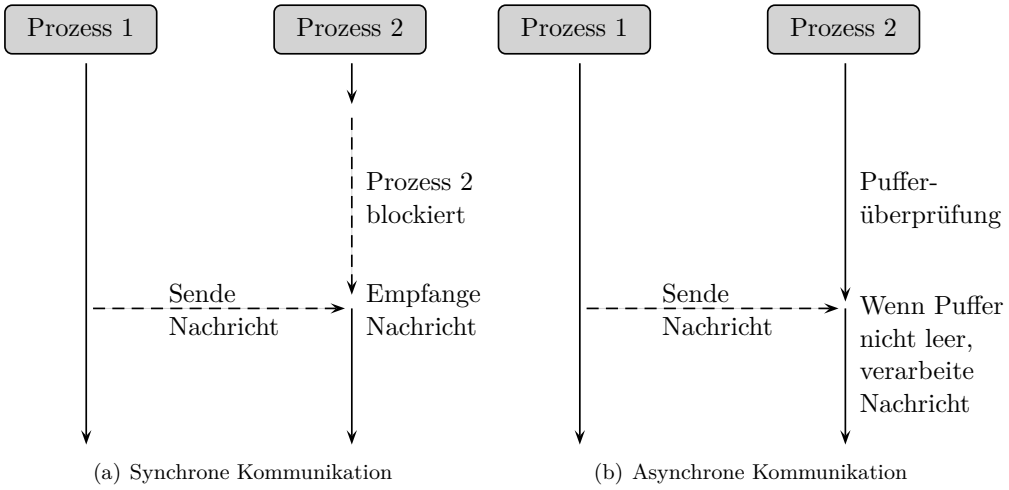
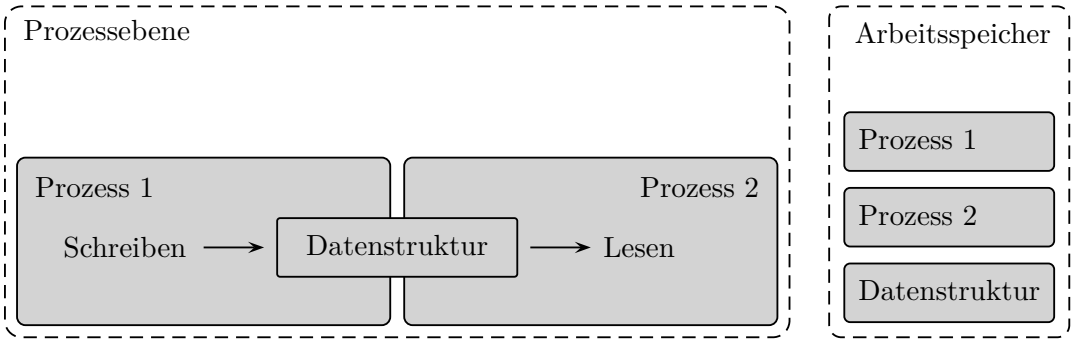


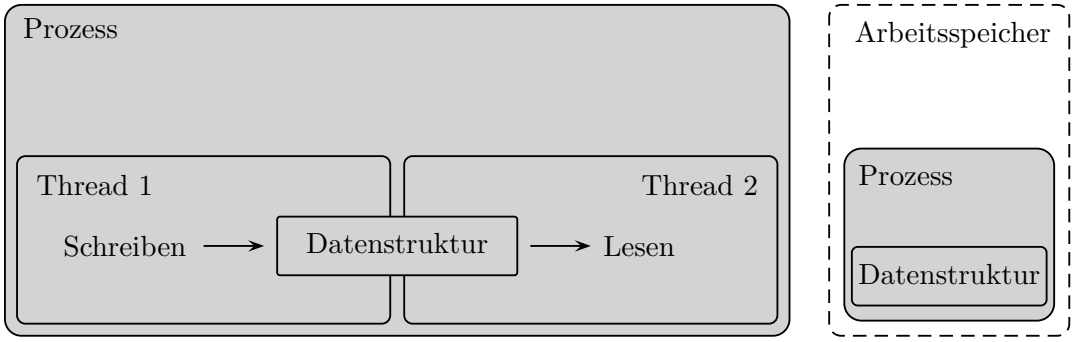
Abbildung 2.3.: Synchrone und asynchrone Kommunikation

2.4.2. Synchrone und asynchrone Datenübertragung

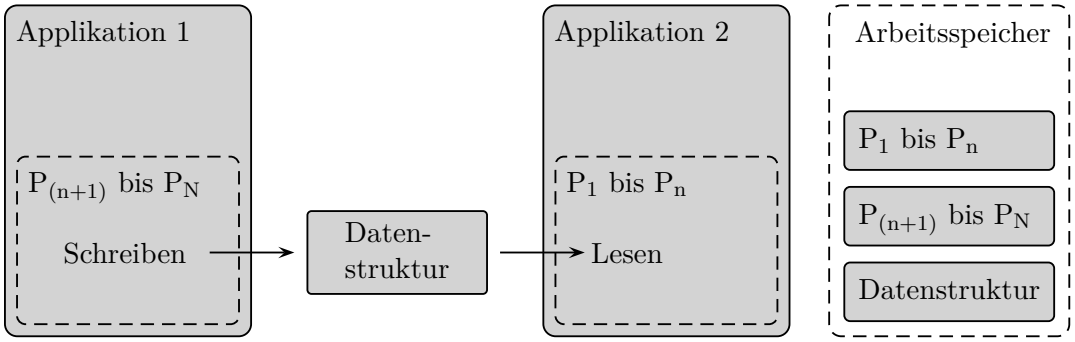
Kommunikation kann synchron oder asynchron erfolgen [82]. Abbildung 2.3 skizziert die beiden Kommunikationsarten. Bei einer synchronen Kommunikation muss der Empfänger die übertragenen Daten zum gleichen Zeitpunkt empfangen wie der Sender diese gesendet hat, das bedeutet im Normalfall das Warten des Empfängers auf den Sender. Der Empfänger blockiert dann. Während der Sender zeitkritische Anforderungen erfüllen kann, ist das im Rahmen dieser Kommunikation für den blockierenden Empfänger nicht ohne weiteres möglich. Diese Problemstellung wird durch eine asynchrone Kommunikation, in der die Daten nicht gleichzeitig gesendet und empfangen werden müssen, vermieden. Dies geschieht im Allgemeinen durch die Verwendung passender Datenpuffer, in denen die Daten zwischengespeichert werden. Dabei ist eine Datenübertragung über gemeinsam genutzten Speicher effizienter als über das Versenden zum Beispiel einer Warteschlange als Datenpuffer [82]. Asynchrone Kommunikation löst das Problem des blockierenden Lesers. Unabhängig von der gewählten Kommunikationsart müssen Sender und Empfänger unter Berücksichtigung zeitkritischer Anforderungen die übertragenen Daten verarbeiten.



(a) Zwischen Prozessen



(b) Zwischen Subprozessen (Threads)



(c) Zwischen Applikationen

Abbildung 2.4.: Kommunikationsszenarien

2.4.3. Kommunikationsszenarien

Die in diesem Abschnitt vorgestellten Kommunikationsszenarien wurden im Rahmen der Bachelorarbeit *Evaluierung des Ressourcen Scheduling in zeit-heterogenen Systemen* [59] von Dominik Schmithausen aufbereitet. Abbildung 2.4 zeigt die drei typischen auf Softwareplattformen implementierten Kommunikationsszenarien, in denen einzelne Prozesse, Subprozesse (Threads) oder auch ganze Applikation, die durch mehr als einen Prozess realisiert sind, miteinander kommunizieren. Abbildung 2.4(a) zeigt ein Kommunikationsszenario mit zwei Prozessen, die über eine zum Beispiel asynchrone unidirektionale Übertragung Daten austauschen. Dabei realisieren ein lesender und ein schreibender Prozess diese unidirektionalen Datenübertragung, die innerhalb derselben Anwendung ausgeführt werden. Da Prozesse keine Rechte auf dedizierte Speicherbereiche anderer Prozesse haben, muss die Datenübertragung über eine Datenstruktur erfolgen, deren Elemente nicht innerhalb eines dedizierten Speicherbereichs alloziert werden. Abhängig von der verwendeten Softwareplattform kann diese Einschränkung bei einer konkreten Implementierung aufgehoben werden. Abbildung 2.4(b) zeigt ein Kommunikationsszenario, in dem innerhalb eines Prozesses zwei Subprozesse (Threads) ausgeführt werden, die wieder über eine unidirektionale Datenübertragung kommunizieren. Die entsprechende Datenstruktur sollte im dedizierten Bereich des Hauptprozesses alloziert werden. Im dritten Kommunikationsszenario, gezeigt in Abbildung 2.4(c), werden zwei Applikationen auf einer Softwareplattform ausgeführt, indem jeweils mehrere Prozesse ausgeführt werden. In Applikation 1 sind das die Prozesse P_1 bis P_n, in Applikation 2 die Prozesse P_{n+1} bis P_N. In diesem Szenario wird eine Datenstruktur ebenfalls in gemeinsamen Speicher alloziert.

Prozesse und ihre Subprozesse können je nach Hardwareplattform auf mehreren Prozessorkernen verteilt ausgeführt werden, wobei eine Zuteilung der einzelnen Aufgaben im Allgemeinen durch die verwendete Softwareplattform geplant wird.

3. Entwicklung eingebetteter Systeme

Die in der Einleitung zusammengefasste Problemstellung und Zielsetzung wird in diesem Kapitel aufgegriffen und vertieft. Dazu beschreibt Abschnitt 3.1 den Trend einerseits zur Substitution eingebetteter Systeme durch Standard-IT [9] und andererseits zur Verwendung von Standard-IT zum Testen eingebetteter Systeme. Diese Problemstellung wird im Abschnitt 3.2 auf den Datenfluss und den Kontrollfluss innerhalb von Netzwerken abstrahiert, damit diese im Abschnitt 3.3 formalisiert werden kann. In Abschnitt 3.4 wird dann ein entsprechender Lösungsansatz bezüglich der Problemstellung gegeben, welcher in den Kapiteln 4 und 5 weiter herausgearbeitet wird.

3.1. Trend zur Standard-IT und Problemstellung

Software für eingebettete Systeme wird anders entwickelt als Software für Standard-IT [9]. Durch ihre spezifische Hauptfunktion, wie zum Beispiel in der Automobilindustrie die Motorensteuerung oder Fahrassistenz, müssen eingebettete Systeme und damit auch die Software für diese Systeme nicht-funktionale Anforderungen erfüllen, die durch die Hauptfunktion der Systeme bestimmt wird. Gegen diese nicht-funktionalen Anforderungen muss insbesondere im Rahmen der Entwicklung der Software für eingebettete Systeme getestet werden. Da eine nicht-funktionale Eigenschaft wie Echtzeitfähigkeit, Robustheit oder auch Verfügbarkeit nicht durch nur eine einzige physikalische Größe bestimmt ist, kann eine Bewertung dieser Eigenschaft nicht durch das Ausmessen einer einzelnen Messgröße quantifiziert werden. Vielmehr werden im Rahmen einer Metrik für eine konkret zu bewertende Eigenschaft alle relevanten Messgrößen, wie zum Beispiel Zeit oder (M)IPS[1], bestimmt. Diese werden dann innerhalb oder an den Schnittstellen des eingebetteten Systems gemessen. Dieselbe Metrik legt fest wie eine nicht-funktionale Eigenschaft anhand der resultierenden Messdaten bewertet werden muss. Die Messdaten werden mit Hilfe der Metrik *operationalisiert*, um eine Beurteilung des eingebetteten Systems und nicht zuletzt der Software dieses Systems durchzuführen. Das Testen von eingebetteten Systemen ist zum Beispiel bei der Durchführung von Langzeit- oder Regressionstests bis hin zu Zertifizierungsverfahren [5, 6, 29, 60] sehr aufwendig, insbe-

[1] (Millionen) Instruktionen pro Sekunde

sondere wenn *Software-in-the-Loop* (SiL) oder *Hardware-in-the-Loop* (HiL) Testmethoden angewendet werden oder das zu testende eingebettete System in den technischen Kontext, *Feldtest*, verbaut und unter realen Umweltbedingungen getestet wird. Der Testaufwand sowie die Relevanz der Testergebnisse ist dabei abhängig von der verwendeten Metrik, die zur Bewertung der nicht-funktionalen Eigenschaft verfügbar sind.

Eingebettete Systeme sind heutzutage allgegenwärtig. Dabei erzeugen entweder hohe Stückzahlen dieser Systeme oder die Komplexität von einzelnen Anlagensteuerungen hohe Kosten in der Entwicklung und Wartung. Diese Kosten bestimmen zusammen mit der *Time-to-Market* und Flexibilität die Konkurrenzfähigkeit vieler Hersteller eingebetteter Systeme. Bedingt durch diesen Kostendruck zusammen mit dem gleichzeitig wachsenden Bedarf an Funktionalität [41, 58], der sich in der Automobilindustrie und neuerdings auf dem Markt mobiler Applikationen am sichtbarsten manifestiert, werden Alternativen zu eingebetteten Systemen mit nur einer einzelnen Hauptfunktion gesucht. Deswegen werden eingebettete Systeme zunehmend vernetzt, um übergeordnete Funktionen zu realisieren. *Cyber-physical Systems* [37, 40, 67] entspricht diesem Paradigmenwechsel.

Diese Entwicklung zu mehr Systemen höherer Komplexität und mehr Kommunikation untereinander wird durch den steigenden Ressourcenaufwand beschränkt. Weiterhin werden *Hauptfunktionen*, die vorher durch dedizierte eingebettete Systeme realisiert wurden, auf einer gemeinsamen Plattform zusammengefasst. Das reduziert die Kosten aber auch den Platzbedarf, wichtig in der Automobilindustrie, und das Gewicht der verbauten Teile, zum Beispiel wichtig in der Avionik. Dabei werden auch Alternativen untersucht, die es ermöglichen, Funktionalität eingebetteter Systeme auf Rechnersystemen, konzipiert für die reine Informationsverarbeitung, zu realisieren. Verstärkt wird dieser Trend durch nicht-funktionale Anforderungen wie Benutzbarkeit oder Verwendbarkeit, die heutzutage aufgrund der Sichtbarkeit eingebetteter Systeme nicht zuletzt durch den Markt mobiler Applikationen in der Vordergrund treten. Dabei können solche Plattformen nicht mehr statisch geplant werden, sondern müssen zur Laufzeit auf dynamische Ereignisse reagieren können. Die Vernetzung beziehungsweise das Zusammenlegen von eingebetteten Systemen ermöglicht eine neue Komplexität der Hauptfunktionen, die auch eingesetzt wird, um den Bedarf nach neuer Funktionalität zu decken. Durch diesen Trend wird die Entwicklung von traditionell sicherheitskritischen Systemen und verlässlicher Software für diese Systeme erschwert.

Die Entwicklung von konventionellen eingebetteten Systemen hin zu Plattformen mit variablen Anforderungen an Sicherheit und Zuverlässigkeit der Systeme schlägt sich auch in den Testprozessen nieder. Einerseits müssen die Testmethoden optimiert werden, um Fehler und nicht-gewollte Eigenschaften eines Systems so früh wie möglich, zum Beispiel noch während der Modellierung der eingebetteten Systeme zu identifizieren und auszuschließen. Eine Möglichkeit ist die Testautomatisierung [14]. Andererseits werden

alternative Techniken entwickelt, um die eigentlichen Testprozeduren so kostengünstig wie möglich, aber so effektiv wie nötig aufzusetzen. In den durchgeführten Kooperationsprojekten mit der Industrie wurde aus diesem Grund versucht, für die reine Informationsverarbeitung ausgelegte Rechnersysteme in die Testprozeduren jeweils relevanter eingebetteter Systeme zu integrieren. Für die Automobilindustrie wurde im Rahmen des entsprechenden Kooperationsprojektes ein Werkzeug zum Testen von Motoren und Funktionen untersucht und entwickelt. Mit dem Kooperationspartner aus der Licht- und Signaltechnik konnten in einem weiteren Projekt HiL-Tests durch eine Softwaresimulation substituiert werden. Diese ersten Schritte schließen dabei nicht aus, dass solche Systeme zukünftig auch eingebettete Systeme ersetzen. Rechnersysteme ohne spezifische Hauptfunktion sind flexibler einsetzbar, für neue Aufgaben neu konfigurierbar und erweiterbar. Bei Wartung oder Ausfall sind generische Systeme leichter zu ersetzen.

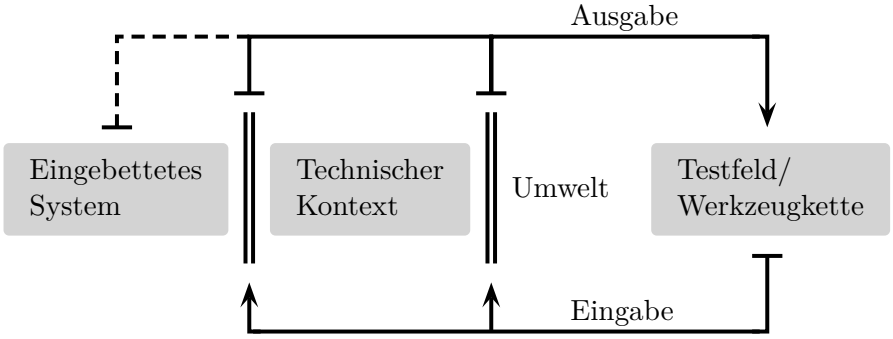


Abbildung 3.1.: Schnittstellen zum Testen eingebetteter Systeme

Eingebettete Systeme berechnen Ausgaben und setzen diese in den technischen Kontext ab. Dadurch werden physikalische Größen beeinflusst, das System als Ganzes kontrolliert. Dies geschieht periodisch oder auf Basis einer Eingabe durch den technischen Kontext. Dieser Kontext beeinflusst seine Umwelt und wird wiederum durch Eingaben von dieser beeinflusst. Für die Ausgabeberechnung wesentliche Umwelteingaben werden über den technischen Kontext an das eingebettete Systeme weitergegeben. Abbildung 3.1 setzt die Vorgehensweise beim Testen eingebetteter Systeme mit den Schnittstellen zu diesen Systemen schematisch in Verbindung. Technisch werden diese Bezugspunkte als bidirektionale Schnittstellen interpretiert, in der Abbildung als Doppelbalken dargestellt. An diesen Schnittstellen können entweder Dateninformation ausgegeben, dargestellt durch Querbalken, oder eingegeben werden, dargestellt durch Pfeilspitzen. Am Beispiel der Regelungstechnik werden diese Schnittstellen noch einmal in Abbildung 3.2 verdeutlicht. Um jetzt zum Beispiel die Hauptfunktion eines eingebetteten Systems zu

testen, werden über die oben genannten Schnittstellen spezifische Eingaben an das eingebettete System oder den technischen Kontext gegeben und Ausgaben gemessen, in der Abbildung als unidirektionale Pfeile dargestellt. Prinzipiell kann die Software für eingebettete Systeme auch direkt und ohne Verwendung von Schnittstellen getestet werden, in der Abbildung als gestrichelter Pfeil dargestellt. Generierung der Eingaben und Bewertung der Ausgaben, zum Beispiel durch eine Operationalisierung der Messdaten, findet innerhalb eines *Testfeldes* statt.

Definition 1: Testfeld Testfeld bezeichnet einen Verbund technischer Geräte beziehungsweise Systeme, in dem ein eingebettetes System (*System unter Test*, SuT) zur Durchführung von Tests verbaut werden kann. Das Testfeld substituiert dabei je nach Testmethode den technischen Kontext des eingebetteten Systems oder die Umwelt des technischen Kontextes. Das SuT wird oftmals als Teil des Testfeldes interpretiert. ◇

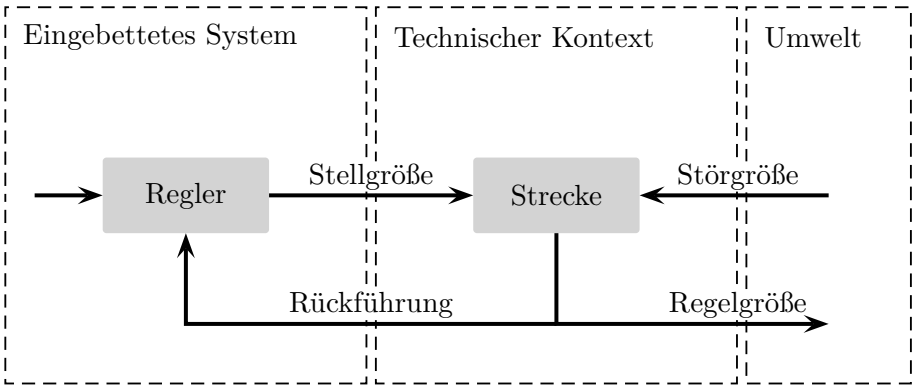


Abbildung 3.2.: Schnittstellen am Beispiel Regelungstechnik

Ein Regelkreis [45] ist ein technischer Prozess, in dem eine physikalische Größe beeinflusst wird. Dabei berechnet der Regler als eingebettetes System eine Stellgröße, mit der auf die Strecke als technischer Kontext eingewirkt wird. Die Ausgabe der Strecke ist die Regelgröße. Diese beeinflusst einerseits die Umwelt des technischen Kontextes und wird andererseits als Eingabe an den Regler zurückgeführt. Diese Rückführung und die Stellgröße entsprechen der bidirektionalen Schnittstelle zwischen dem eingebetteten System und dem technischen Kontext. Aus der Umwelt wirkt eine Störgröße auf die Strecke ein und damit über die Rückführung indirekt auf den Regler. Störgröße und Regelgröße entsprechen der bidirektionalen Schnittstelle zwischen Kontext und der Umwelt.

Getestet werden eingebettete Systeme mit unterschiedlichsten Werkzeugen, deren Aus-

wahl mit der Hauptfunktion der zu testenden Systeme korreliert. Dabei steht insbesondere bei eingebetteten Systemen das anforderungsbasierte Testen im Vordergrund [6, 14, 66]. Im Rahmen eines Testfeldes werden an den zur Verfügung stehenden Schnittstellen Messungen durchgeführt, die für eine Beurteilung unter Berücksichtigung der Testparameter eingesetzt werden. Mit Hilfe eines Oszilloskops zum Beispiel kann am technischen Kontext gemessen werden, Diagnosewerkzeuge lesen Daten direkt aus den zu testenden Systemen ab. Die gemessenen Daten werden dann mit Hilfe einer Metrik bewertet, wenn gegen nicht-funktionale Anforderungen getestet und beurteilt werden soll. Es ist üblich, dass die Software, auch für eingebettete Systeme, in frühen Phasen der Entwicklung mit Hilfe von *Whitebox*-Tests auf ihre Funktionen getestet wird. Das sind im Wesentlichen, aber nicht ausschließlich Modultests. Mit dieser Methode ist allerdings kein Testen gegen eine Spezifikation möglich, wodurch Whitebox-Tests in späteren Phasen der Entwicklung von eingebetteten Systemen nicht mehr praktikabel sind. Es gibt noch weitere Nachteile, wie die notwendige Einflussnahme auf den Quelltext der zu testenden Software, was eine Zertifizierung erschwert, oder den hohen Bedarf an Domänenwissen. Eine reine Messung ist für die Beurteilung von nicht-funktionalen Anforderungen an eingebettete Systeme nicht ausreichend. Aus diesem Grund werden das zu testende System oder der technische Kontext an den zur Verfügung stehenden Schnittstellen mit passenden Eingaben stimuliert. Jetzt kann auf der Basis einer Eingabe eine Ausgabe beurteilt werden, was dem klassischen *Blackbox*-Testen entspricht.

Eingebettete Systeme müssen oftmals unter realen Randbedingungen getestet werden, um fehlerhaftes Laufzeitverhalten im Feldeinsatz nahezu ausschließen zu können. Tests unter diesen Parametern sind kostenintensiv. Deswegen werden im Vorfeld dieser finalen Tests Testszenarien durchgeführt. Im Rahmen dieser Testszenarien wird das Verhalten des entsprechenden technischen Kontextes oder dessen Umwelt simuliert. Mit Hilfe einer passenden Modellierung interagiert des Testfeld mit dem zu testenden System. Das hat den Nachteil, dass von der Realität abstrahiert wird, erlaubt aber das auch automatisierbare Ausschließen vieler Fehler im Vorfeld kostenintensiver Feldtests. Das Testen von eingebetteten Systemen und deren Software mit Hilfe der Simulation lässt sich in drei Kategorien zusammenfassen. Mit *Model-in-the-Loop* wird ein erstes Model der Software gegen ein Modell des technischen Kontextes auf der Entwicklungsplattform getestet. Mit *Software-in-the-Loop* wird dann die tatsächliche, gegebenenfalls generierte Software getestet. Die zu testende Software wird weiterhin auf der Entwicklungsplattform ausgeführt, kann nun aber entweder gegen eine Softwaresimulation oder über entsprechende Hardware-Schnittstellen mit ihrem tatsächlichen technischen Kontext interagieren. Mit *Hardware-in-the-Loop* wird schließlich die Software auf dem eingebetteten System getestet, wobei als Testfeld ein Echtzeitrechner eingesetzt wird. In Abhängigkeit der gewählten Schnittstellen findet beim Testen eingebetteter Systeme entweder eine diskrete oder ste-

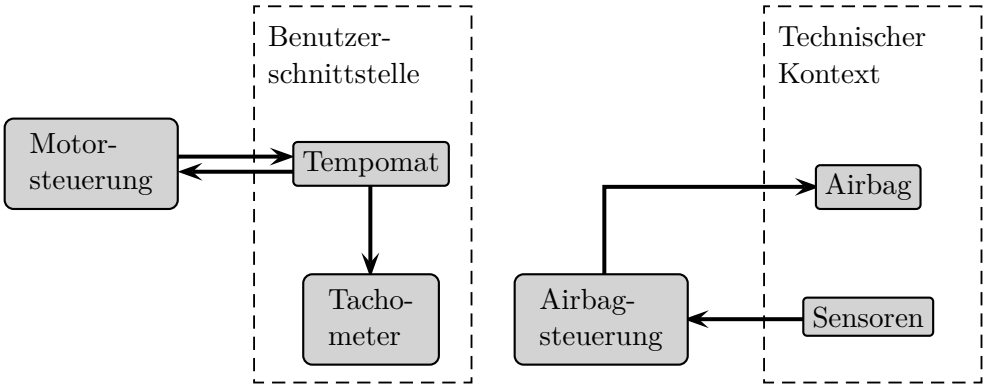
tige (kontinuierliche) Simulation statt. Die vorgestellten Methoden werden während der Entwicklung der eingebetteten Systeme iterativ eingesetzt. Oftmals werden bei diesen Iterationen neue Anforderungen identifiziert oder bereits spezifizierte Anforderungen als unerfüllbar, zu komplex oder auch als irrelevant eingestuft. Diese Ergebnisse werden dann bei der weiteren Entwicklung des eingebetteten Systems berücksichtigt.

Die beschriebenen Testmethoden machen einen kostenintensiven Teil in der Entwicklung eingebetteter Systeme aus. Aus diesem Grund unterliegen die Werkzeuge zum Testen eingebetteter Systeme den gleichen technischen Trends wie die Systeme selbst. Es wird versucht, in die Testprozesse primär Standard-IT zu integrieren. Dieser Bedarf wurde auch in unseren Kooperationsprojekten mit der Industrie deutlich und wir konnten diesen Trend untersuchen und neue Methoden und Konzepte unter kontrollierbaren Bedingungen in den Testfeldern einsetzen. Die Entwicklung war nicht a priori auf konventionelle Methoden und Konzepte für eingebettete Systeme beschränkt. Außerdem konnten wir weitere Anforderungen wie die Benutzerfreundlichkeit, Bedienbarkeit oder auch die Robustheit der Werkzeugkette berücksichtigen.

Unabhängig von der Motivation Standard-IT und eingebettete Systeme in demselben Netzwerk oder auf derselben Plattform zu verbauen, muss die Infrastruktur zusammen mit den Softwarearchitekturen der einzelnen Systeme derart kommunizieren, dass die essentiellen spezifischen Hauptfunktionen, die das Netzwerk implementieren soll, zur Laufzeit erfüllt werden. Die Entwürfe von Rechnersystemen der Standard-IT erfüllen im Allgemeinen keine nicht-funktionalen Anforderungen, die an eingebettete Systeme gestellt werden. Im folgenden Abschnitt 3.2 werden die dadurch entstehenden, technischen Herausforderungen solcher Netzwerke beziehungsweise Plattformen identifiziert. Abschnitt 3.3 setzt den Schwerpunkt der Untersuchung dann auf diejenigen Systeme der Standard-IT, die mit konventionell entworfenen eingebetteten Systemen kommunizieren müssen. Die in dieser Arbeit vorgestellte iterative Vorgehensweise, um Rechnersysteme so zu adaptieren, dass diese Systeme mit eingebetteten Systemen kommunizieren zu können, ohne die spezifische Hauptfunktion dieser Systeme zu beeinflussen, wird in Abschnitt 3.4 vorgestellt.

3.2. Zeitheterogene Netzwerke

In einem Netzwerk und auf Plattformen findet zwischen den einzelnen Systemen beziehungsweise Laufzeitprozessen Kommunikation statt. Abstrahiert man von den verwendeten Protokollen wie zum Beispiel den Bussystemen in Netzwerken oder dem gemeinsamen Speicher auf Plattformen, dann besteht diese Kommunikation aus Datenfluss und Kontrollfluss zwischen den Systemen oder Laufzeitprozessen. Abbildung 3.3 zeigt jeweils



(a) Datenfluss (b) Kontrollfluss

Abbildung 3.3.: Abstrakte Beispiele zur Netzwerkkommunikation

ein Beispiel zu diesen Kommunikationskonzepten.

Ein Beispiel für Datenfluss ist die Datenübertragung zwischen Motorsteuerung, Tempomat und Tachometer 3.3(a). Dabei unterscheiden sich die nicht-funktionalen Anforderungen an die Kommunikation zwischen den einzelnen Systemen. Tempomat und Motorsteuerung müssen in Echtzeit Daten austauschen, um ihre jeweilige Hauptfunktion zu erfüllen. Diese Anforderung entsteht durch einen impliziten Kontrollfluss, wenn Ein- und Ausgaben eingebetteter Systeme für die Steuerung des technischen Kontextes berücksichtigt werden müssen. Die Datenübertragung zum Tachometer unterliegt dieser Anforderung nicht mit gleicher Strenge, der Tempomat muss nur ohne besondere zeitliche Anforderung die Daten Richtung Tachometer absetzen können. Ein Beispiel für reinen Kontrollfluss ist der Airbag-Mechanismus 3.3(b). Ausgehend von Kollisionssensoren steuert die Airbagsteuerung das eigentliche Airbagmodul. Das Auslösen eines Airbags ist ein zeitkritischer Vorgang. Sensoren und die Airbagsteuerung müssen robust und zuverlässig entworfen werden, damit der Airbag nur bei tatsächlichen Kollisionen, dann aber mit sehr hoher Wahrscheinlichkeit, auslöst.

Echtzeitfähigkeit ist eine nicht-funktionale Anforderung an eingebettete Systeme, damit diese Systeme ihrer spezifische Hauptfunktion entsprechen können. In Netzwerken solcher Systeme werden keine zeitkritischen Anforderungen widerrufen. Durch den Trend Standard-IT in Netzwerke eingebetteter System zu integrieren, entstehen zeitheterogene Netzwerke. Das Beispiel zum Datenfluss über einen Tempomat ist ein solches Netzwerk.

In zeitheterogenen Netzwerken werden durch den Datenfluss und den Kontrollfluss nicht-funktionale Anforderungen von einzelnen Netzwerkknoten auf diejenigen Netzwerknoten propagiert, mit denen Kommunikation stattfindet. Dies gilt insbesondere für die

zeitkritischen Anforderungen an das Laufzeitverhalten der vernetzten Systeme, wie in den folgenden Abbildungen aus einer Datenfluss- und einer Kontrollflussperspektive gezeigt wird.

Definition 2: Zeitheterogene Netzwerke Informationstechnologische Infrastrukturen werden als zeitheterogene Netzwerke bezeichnet, wenn in diesen Infrastrukturen Systeme beziehungsweise Prozesse interagieren, für die unterschiedliche strenge zeitliche Anforderungen gelten. ◇

Als Ausprägung solcher Infrastrukturen werden in dieser Arbeit die folgenden zwei Arten zeitheterogener Netzwerke bezeichnet:

- Netzwerke eingebetteter Systeme und Standard-IT. Die einzelnen Netzwerkknoten sind dabei unter Berücksichtigung zeitkritischer Anforderungen verschiedener Strenge entworfen worden.
- Softwareplattformen, auf denen Prozesse interagieren, die verschiedene spezifische Hauptfunktionen oder Informationsverarbeitung realisieren. Auch hier realisieren die einzelnen Programme Laufzeitverhalten unterschiedlicher Strenge.

Prinzipiell können beim Entwurf einzelner Netzwerkknoten oder Programme zeitkritische Anforderungen auch gänzlich unberücksichtigt geblieben worden sein, die für andere Systeme innerhalb desselben Netzwerkes essentiell sind.

Abbildung 3.4 skizziert potentiellen Datenfluss in einem beispielhaften zeitheterogenen Netzwerk. Drei eingebettete Systeme sind in diesem Beispiel vernetzt. Davon sind zwei Systeme in den gleichen technischen Kontext eingebettet und kommunizieren sowohl mit diesem als auch untereinander. Diese Kommunikation muss aufgrund der spezifischen Hauptfunktion der Systeme in Echtzeit erfolgen. Innerhalb der Kommunikation zwischen *System B* und dem Kommunikationsknoten werden Daten an das eingebettete System gesendet. Sind diese Daten zur Ausführung der spezifischen Hauptfunktion für System B relevant, muss auch diese Kommunikation in Echtzeit durchgeführt werden. Der Kommunikationsknoten fungiert als Schnittstelle zwischen dem zeitkritischen und unkritischen Teil des Netzwerks, er kommuniziert mit weiteren nicht-echtzeitfähigen Netzwerkknoten. Das sind hier als Beispiele ein mobiles Endgerät, eine Verwaltungsschnittstelle und ein Datenspeicher. Zwischen *System A* und einem externen Monitor findet ein Datenfluss in Richtung des Monitors statt. Während an den Monitor selbst keine zeitkritischen Anforderungen gestellt werden, muss die Kommunikation zwischen beiden Netzwerkknoten

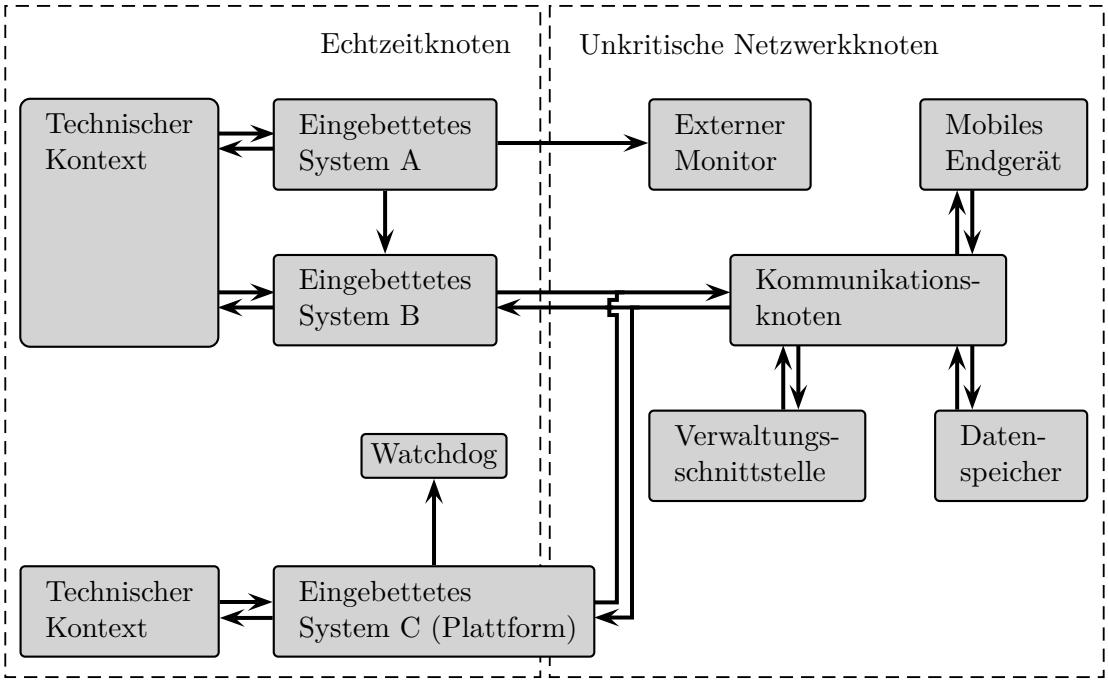


Abbildung 3.4.: Datenfluss in zeitheterogenen Netzwerken

derart implementiert sein, dass durch die Kommunikation die spezifische Hauptfunktion von System A nicht beeinflusst wird. Das kann zum Beispiel durch eine gepufferte Kommunikation geschehen, in der System A in eine Warteschlange schreibt. Einträge in diese Warteschlange werden dann überschrieben, wenn der Monitor diese Einträge nicht rechtzeitig ausgelesen hat. *System C* ist eine Echtzeitplattform und ist in einen weiteren technischen Kontext eingebettet. Auf dieser Plattform sind sowohl Echtzeitprozesse als auch nicht-echtzeitfähige Prozesse implementiert worden. Dabei muss die Plattform so entworfen sein, dass die Laufzeit der Echtzeitprozesse nicht durch die ebenfalls vorhandene Kommunikation mit dem Kommunikationsknoten oder die nicht-echtzeitfähigen Prozesse auf derselben Plattform beeinflusst wird. Eine *Watchdog* Implementierung, ein typisches Entwurfsmuster in sicherheitskritischen Systemen, überwacht System C.

Abbildung 3.5 skizziert potentiellen Kontrollfluss in einem anderen beispielhaften zeitheterogenen Netzwerk. Ein eingebettetes System ist mit einem technischen Kontext, einem Watchdog und einem externen Monitor vernetzt. Das eingebettete System, der technische Kontext und die Kommunikation zwischen diesen müssen prinzipiell echtzeit-

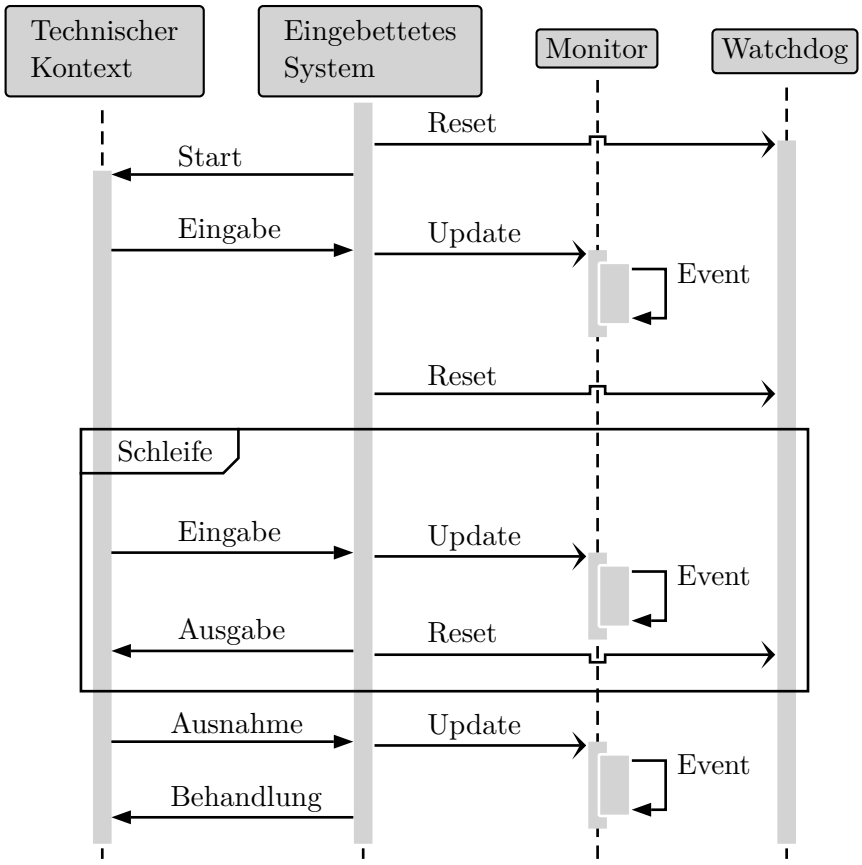


Abbildung 3.5.: Kontrollfluss in zeitheterogenen Netzwerken

fähig entworfen werden, um die spezifische Hauptfunktion nicht zu beeinflussen. Diese Kommunikation erfolgt im Allgemeinen synchron. Die Kommunikation mit Monitor und Watchdog kann prinzipiell asynchron erfolgen. Dabei müssen aber die spezifischen Anforderungen dieser Systeme berücksichtigt werden. Der Watchdog zum Beispiel wird eine Ausnahme auslösen, sollte der Zähler des Watchdog (Timer) durch das eingebettete System nicht rechtzeitig zurückgesetzt werden. Zu Beginn der Laufzeit setzt das eingebettete System den Zähler des Watchdog zurück und startet danach den technischen Kontext neu, um das Netzwerk in einen determinierten Zustand zu versetzen. Der Kontext quittiert nach erfolgreichem Neustart. Das eingebettete System sendet diese Information weiter an den externen Monitor, der in diesem Beispiel eine grafische Benutzerschnittstelle bereitstellt, welche diese Information weiter verarbeitet. Nach der Weitergabe der

Information wird erneut der Zähler des Watchdog zurückgesetzt. Das Netzwerk und insbesondere das eingebettete System führt nun die spezifische Hauptfunktion durch. Dazu werden in einer Schleife Eingaben aus dem technischen Kontext verarbeitet und an den Monitor weitergegeben. Nach der Verarbeitung erfolgt eine Ausgabe an den technischen Kontext und das eingebettete System setzt den Watchdog zurück. Diese Schleife wird solange durchgeführt, bis das Netzwerk deaktiviert wird oder eine Ausnahme, zum Beispiel im technischen Kontext, auftritt. In diesem Fall muss das eingebettete Systeme eine Ausnahmebehandlung durchführen und darüber ebenfalls den Monitor informieren. Während der Durchführung der Hauptfunktion muss die Echtzeitfähigkeit der Kommunikation gesichert werden. In speziellen Fällen, wie hier zum Beispiel der Ausnahmebehandlung, ist aber auch eine zeitkritische Weitergabe des kritischen Netzwerkzustandes erforderlich, um den potentiell entstehenden Schaden einzugrenzen.

Definition 3: Propagation zeitkritischer Anforderungen Zeitkritische Anforderungen an ein in einem zeitheterogenen Netzwerk mit anderen Systemen verbundenes eingebettetes System können nur dann erfüllt werden, wenn entsprechende zeitkritische Anforderungen von den mit dem eingebetteten System verbundenen Systemen und den beteiligten Kommunikationssystemen eingehalten werden. [64, 82, 83]. Das ist immer dann der Fall, wenn die anderen Systeme oder die Kommunikation mit diesen Knoten die spezifische Hauptfunktion des eingebetteten Systems zur Laufzeit beeinflussen.

Das eingebettete System propagiert seine zeitkritischen Anforderungen auf diejenigen Systeme, mit denen es kommuniziert. ◇

Tritt die Propagation auf, müssen die entsprechenden Netzwerkknoten Datenpakete deterministisch versenden und verarbeiten, um verspätete Datenverarbeitung oder sogar Datenverlust zu vermeiden. Kritische Ereignisse müssen fristgerecht kommuniziert werden. Sind auf einer Echtzeitplattform mehrere Aufgaben zusammengefasst, müssen diese entsprechend auf die vorhandenen Ressourcen verteilt werden, um rechtzeitig und gleichzeitig ausgeführt werden zu können.

Dazu kommt, dass mit steigender Komplexität von Echtzeitplattformen und deren Aufgaben deren Umsetzung immer schwieriger wird. Außerdem werden zunehmend Echtzeitsysteme entworfen, bei denen dynamisch zu verarbeitende Aufgaben im Laufzeitkontext auftreten. Bei diesen Entwürfen müssen die Ressourcen auch während der Laufzeit neu verteilt werden [3, 11, 41]. Wenn auf solchen Echtzeitsystemen Aufgaben mit zeitkritischen Anforderungen realisiert sind, werden diese Anforderungen auf die gesamte Plattform propagiert. Kann diese Plattform dann aber dynamische Berechnungen mit damit einhergehenden Ressourcenverteilungen nicht in deterministischer Zeit durchführen, zum Beispiel weil andere Aufgaben auf derselben Echtzeitplattform zu viele Res-

sourcen verbrauchen, können die propagierten Anforderungen nicht mehr durch diese Plattform erfüllt werden. Für Echtzeitsysteme mit dynamischer Ressourcenvergabe zur Laufzeit ist diese Problemstellung generell vorhanden.

Zeitkritisch sind in zeitheterogenen Netzwerken aber nicht nur die konventionellen Echtzeitsysteme. Zum Beispiel muss Standard-IT geeignete Methoden implementieren, um Echtzeitdaten in der richtigen Reihenfolge auszuwerten. Oder wenn Daten nach der eigentlichen Laufzeit *offline* ausgewertet werden, müssen alle Datensätze gesichert worden sein, um eine valides Ergebnis berechnen zu können. Sind solche Anforderungen vorhanden, können Datenpuffer nicht einfach überschrieben werden, wenn der stetige Datenstrom, der innerhalb des Netzwerkes durch die Echtzeitsysteme entsteht, in kurzer Zeit zu viele Informationen an die nicht-echtzeitfähigen Netzwerkknoten sendet. Das gilt insbesondere auch für das Testen eingebetteter Systeme mit Testfeldern, in denen nicht nur Echtzeitsysteme verbaut sind. Das Testfeld muss auch als zeitheterogenes Netzwerk solche Anforderungen berücksichtigen.

3.3. Propagationskonformität

In zeitheterogenen Netzwerken werden zeitkritische Anforderungen auf Netzwerkknoten propagiert, die nicht dafür entworfen wurden, diese Anforderungen zu erfüllen. In den traditionellen Entwicklungsprozessen eingebetteter Systeme gibt es oftmals noch keine passenden Konzepte, Methoden oder Werkzeuge, um die Propagation zeitkritischer Anforderungen zu berücksichtigen und den Trend zu zeitheterogenen Netzwerken zu unterstützen. Das gilt auch für die Werkzeugketten, mit denen eingebettete Systeme während der Entwicklung getestet werden. Die einzelnen Netzwerkknoten eines zeitheterogenen Netzwerkes müssen sich zur Laufzeit konform gegenüber den propagierten zeitkritischen Anforderungen verhalten und dem entsprechend entwickelt werden.

Definition 4: Propagationskonformität Die Propagationskonformität eines Netzwerkknotens in einem zeitheterogenen Netzwerk bezeichnet den Grad, zu welchem mit diesem Netzwerkknoten vernetzte Echtzeitsysteme ihre spezifische Hauptfunktion durchführen können, ohne über den Datenfluss und Kontrollfluss von und zu diesem Netzwerkknoten negativ beeinflusst zu werden.

Ein zeitheterogenes Netzwerk ist dann propagationskonform beziehungsweise vollständig propagationskonform, wenn alle enthaltenen Netzwerkknoten diese Eigenschaft besitzen. ◇

Diese Propagationskonformität kann o.B.d.A durch eine Konformitätsbewertung ent-

sprechend der Norm *ISO/IEC 17000:2004* [2, 65] bestimmt werden. Dabei sind die Objekte der Konformitätsbewertung nicht eingeschränkt. Diese kann zum Beispiel durch eine Testfallüberdeckung oder eine andere Operationalisierung bestimmt werden. Ein Netzwerkknoten in einem zeitheterogenen Netzwerk ist dann vollständig propagationskonform, wenn kein mit diesem Knoten vernetztes Echtzeitsystem beeinflusst wird.

Echtzeitfähigkeit ist eine stärkere nicht-funktionale Anforderung als Propagationskonformität zu zeitkritischen Anforderungen. Deswegen können weitere nicht-funktionale Anforderungen, die gegebenenfalls nicht orthogonal zur Echtzeitfähigkeit eines Systems sind, durch propagationskonforme Netzwerkknoten erfüllt werden. Ziel dieser Arbeit ist es deswegen, die Konformität zu propagierten Anforderungen auf Netzwerkknoten zu erhöhen, die solche Anforderungen nicht a priori erfüllen. Dazu befasst sich die Arbeit mit der Software solcher Netzwerkknoten und es werden Methoden zur Evaluierung und Adaptierung dieser Software vorgestellt. Wird dadurch eine signifikante Propagationskonformität erreicht, lassen sich diese Netzwerkknoten in einem vollständig propagationskonformen zeitheterogenen Netzwerk verbauen.

Um die Propagationskonformität herzustellen, muss die Kommunikation zwischen und innerhalb von Softwarearchitekturen untersucht und gegebenenfalls adaptiert werden. Dadurch können Datenpakete propagationskonform kommuniziert und verarbeitet werden, um verspätete Datenverarbeitung oder sogar Datenverlust zu vermeiden. Kritische Ereignisse können dann fristgerecht kommuniziert werden.

3.4. Iterative Softwareadaptierung

Um die Propagationskonformität eines Netzwerkknotens in einem zeitheterogenen Netzwerk herzustellen, muss die Software dieses Netzwerkknotens adaptiert werden. Die hier vorgestellte Vorgehensweise ist dabei iterativ, kann aber in konventionellen Entwicklungsprozessen eingebetteter Systeme verwendet werden.

Das Aktivitätsdiagramm in Abbildung 3.6 skizziert die Vorgehensweise als iterativen Prozess, um die Propagationskonformität einer zu entwickelnden Software schrittweise zu erhöhen. Dieser Subprozess kann zu einem beliebigen Zeitpunkt im Entwicklungsprozess eines eingebetteten Systems initiiert werden. Zu dieser Prozessskizze gehört deshalb als erster Schritt eine Anforderungsanalyse, um spezifische Anforderungen an zeitheterogene Softwarearchitekturen zu erfassen. Sind konkrete Anforderungen an die Software gestellt worden, findet im zweiten Schritt eine Evaluierung der zu diesem Zeitpunkt vorhandenen Software statt. Erfüllt die vorhandene Softwarearchitektur bereits die zeitkritischen Anforderungen an das Netzwerk, kann der Subprozess erfolgreich beendet und der Entwicklungsprozess fortgeführt werden. Ist das nicht der Fall, muss die Soft-

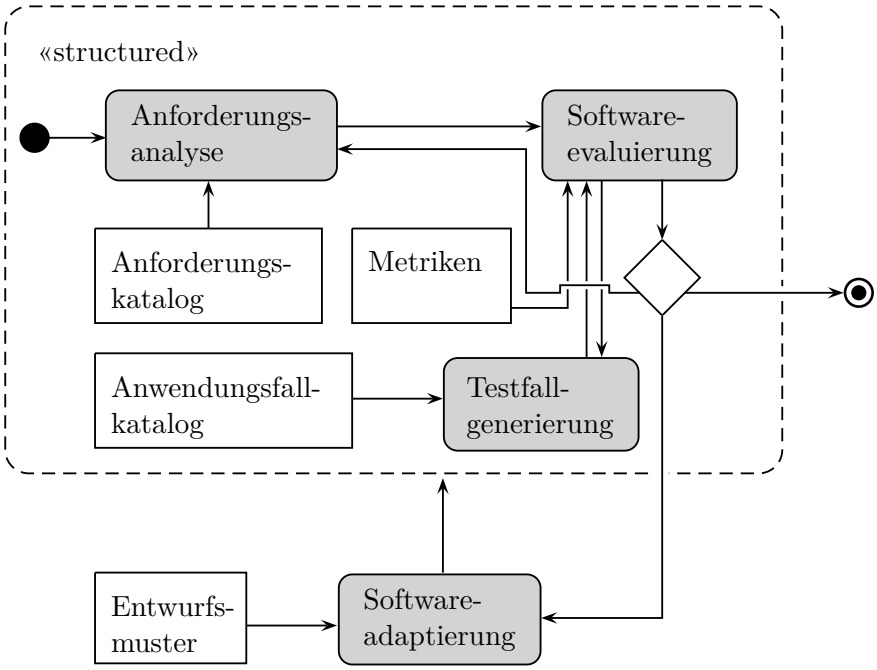


Abbildung 3.6.: Iterative Softwareadaptierung

ware beziehungsweise die Softwarearchitektur adaptiert werden, um nach diesem dritten Schritt einer oder mehreren zeitkritischen Anforderungen zu entsprechen. Danach muss die Softwarearchitektur erneut evaluiert werden. Während der Evaluierung und der Adaptierung der Softwarearchitektur kann auch klar werden, dass die Anforderungsanalyse nicht erschöpfend war. Anforderungen können obsolet werden, müssen modifiziert oder neu spezifiziert werden. Dann muss die Anforderungsanalyse gegebenenfalls, einer erneuten Evaluierung vorgelagert, neu durchgeführt werden. Diese drei Prozessschritte werden solange iterativ durchgeführt, bis der Subprozess erfolgreich beendet werden kann. Dieser Subprozess kann und muss gegebenenfalls zu einem späteren Zeitpunkt der Entwicklung wiederholt werden, wenn der Subprozess zur Adaptierung der Software zu früh im Entwicklungsprozess der Software zum ersten Mal durchgeführt wurde. Dies ist zum Beispiel dann der Fall, wenn Netzwerkknoten ausgetauscht, angepasst oder neu hinzugefügt werden. Die vorgestellte Arbeit berücksichtigt die drei wesentlichen Schritte während der Durchführung der beschriebenen Vorgehensweise durch die in der Abbildung ergänzten Artefakte Anforderungskatalog, Auswahl von Metriken, einen Anwendungsfallkatalog und Entwurfsmuster, die auf Software für den Einsatz in zeitheterogenen Netzwerken

angewendet werden können.

Um die Anforderungsanalyse von Systemen, die innerhalb zeitheterogener Netzwerke verbaut werden sollen, zu unterstützen, wird im Rahmen dieser Arbeit ein *Anforderungskatalog* formuliert, um eine methodische Anforderungsanalyse zu ermöglichen. Dieser Katalog fasst Anforderungen aus verwandter Literatur und für Echtzeitsysteme im Allgemeinen systematisch zusammen; dabei nach Scheduling, Speicherverwaltung und Zeitgebern der Plattform kategorisiert. Den einzelnen Anforderungen werden ebenfalls Anwendungsfälle zugeordnet.

Für die *Evaluierung* vorhandener Software und Softwarearchitekturen beschreibt die vorgelegte Arbeit eine insgesamt dreistufige Vorgehensweise. Teil dieser Beschreibung ist ein Anwendungsfallkatalog, aus dem Testfälle und Testfallklassen generiert werden können. Eine konkrete Operationalisierung der Testergebnisse, die mit Hilfe der genierten Testfälle systematisch ermittelt werden können, wird durch die Auflistung und einen Vergleich relevanter, Echtzeit erfassender Metriken unterstützt. Dabei liegt der Schwerpunkt auf Datenfluss und Kontrollfluss zwischen echtzeitfähiger und nicht-echtzeitfähiger Software innerhalb eines Netzwerks oder auf einer Plattform.

Wurden die unterschiedlichen Softwarearchitekturen und Kommunikationsprotokolle evaluiert, findet bei Bedarf eine Adaptierung der Software oder Softwarearchitektur statt. Dazu werden passende *Entwurfsmuster* benötigt, die in dieser Arbeit vorgestellt beziehungsweise referenziert werden.

4. Evaluierung von Propagationskonformität

Durch die Konkretisierung der Problemstellung in Kapitel 3 wurde der Bedarf an einer auf die Problemstellung zugeschnittenen Evaluierungsmethode gezeigt, die auf Software oder Softwareplattformen angewendet werden kann. Aus diesem Grund wird nun in Abschnitt 4.1 eine dreistufige Vorgehensweise vorgestellt, welche die Durchführung einer solchen Softwareevaluierung unterstützt. Diese Vorgehensweise beinhaltet die Konzepte Konformitätsüberprüfung physikalischer Signale aus dem und in das zu testende System, eine neue Methode zur lebenszyklusabdeckenden Operationalisierung sowie weitere Metriken, und ein Rahmenwerk zur Testfallgenerierung auf der Basis von Anwendungsfällen. Diese Konzepte werden in den Abschnitten 4.5, 4.6 und 4.7 vertieft, während Abschnitt 4.2 als Vorbereitung auf diese Abschnitte einen Katalog aus für die Problemstellung relevanten, zeitkritischen Anforderungen liefert.

Die in diesem Kapitel vorgestellten Konzepte sowie der Anforderungskatalog wurden in einem Kooperationsprojekt mit der Automobilindustrie und in einigen Veröffentlichungen [4, 15, 20, 32, 63] erarbeitet beziehungsweise zusammengefasst. Die Metrik zur lebenszyklusabdeckenden Operationalisierung wurde im Rahmen eines Kooperationsprojektes mit der Avionikindustrie entwickelt.

4.1. Überblick

Das Aktivitätsdiagramm in Abbildung 4.1 zeigt einen auf die in Abschnitt 3.3 vorgestellte Problemstellung zugeschnittenen Evaluierungsprozess, der eine stufenweise Bewertung beziehungsweise Beurteilung von Software oder Softwareplattformen erlaubt; die Stufen sind in der Abbildung als Partitionen dargestellt. Die einzelnen Pfade innerhalb der Prozessdefinition sind voneinander unabhängig durchführbar, Iterationen über die Pfade können beliebig oft wiederholt werden. Dieser Evaluierungsprozess kann als Subprozess innerhalb des in Abbildung 3.6 gezeigten Prozesses zur Softwareadaptierung eingesetzt werden.

Die erste Stufe des Evaluierungsprozesses ist die Konformitätsmessung physikalischer Signale. Dazu wird eine Schnittstelle wie in Kapitel 3 und Abbildung 3.1 beschrieben

ausgewählt. An dieser Schnittstelle zwischen entweder dem zu testenden System und dem technischen Kontext oder dem Kontext und dessen Umwelt werden dann physikalische Signale ausgemessen. Die Messergebnisse wiederum werden direkt quantifiziert oder gegen Referenzsignale auf Konformität überprüft. Eine exemplarische konkrete Konformitätsprüfung, die während einem Kooperationsprojekt mit der Automobilindustrie umgesetzt wurde, wird in Abschnitt 4.5 vorgestellt.

Die zweite Stufe des Evaluierungsprozesses ist die Operationalisierung von Messergebnissen mit Hilfe von Metriken. Auf diesem Prozesspfad muss zuerst eine passende Metrik ausgewählt werden. In Abhängigkeit zu dieser Metrik wird dann eine Messung, gegebenenfalls ein umfangreicher Messaufbau und -ablauf, an den potentiellen Schnittstellen des zu testenden Systems durchgeführt. Die aus der Messung oder dem Test resultierenden Ergebnisse können nun quantifiziert oder über die ausgewählte Metrik beurteilt werden. Eine Auswahl von Metriken zur Beurteilung gegen zeitkritische nicht-funktionale Anforderungen und eine neue lebenszyklusabdeckende Metrik werden in Abschnitt 4.6 vorgestellt. Da passende Messergebnisse auch durch eine Konformitätsmessung ermittelt werden können, baut diese Stufe prinzipiell auf der ersten Stufe auf.

Die dritte Stufe des Evaluierungsprozesses ist das Testen von Software und Softwareplattformen. Zuerst wird ein Testfall ausgewählt, danach erfolgt der eigentliche Testablauf. Dabei können sowohl die erste als auch die zweite Stufe des Evaluierungsprozesses im Rahmen eines Testablaufs der Durchführung entsprechen; in der Abbildung sind die ersten beiden Stufen deswegen strukturell gekapselt. Auf der Basis eines Testablaufs erfolgt im Allgemeinen eine systematische Beurteilung der zu testenden Software, wobei dieser Prozesspfad für beliebig viele Testfälle wiederholt werden kann, um zum Beispiel eine signifikante Testfallabdeckung zu erreichen. Anhang B fasst auf die Problemstellung zugeschnittene Anwendungsfälle zusammen und setzt diese in Bezug zu den Anforderungen aus Abschnitt 4.2. In Abschnitt 4.7 wird gezeigt, wie aus diesem Katalog passende Testfälle und Testfallklassen generiert werden können.

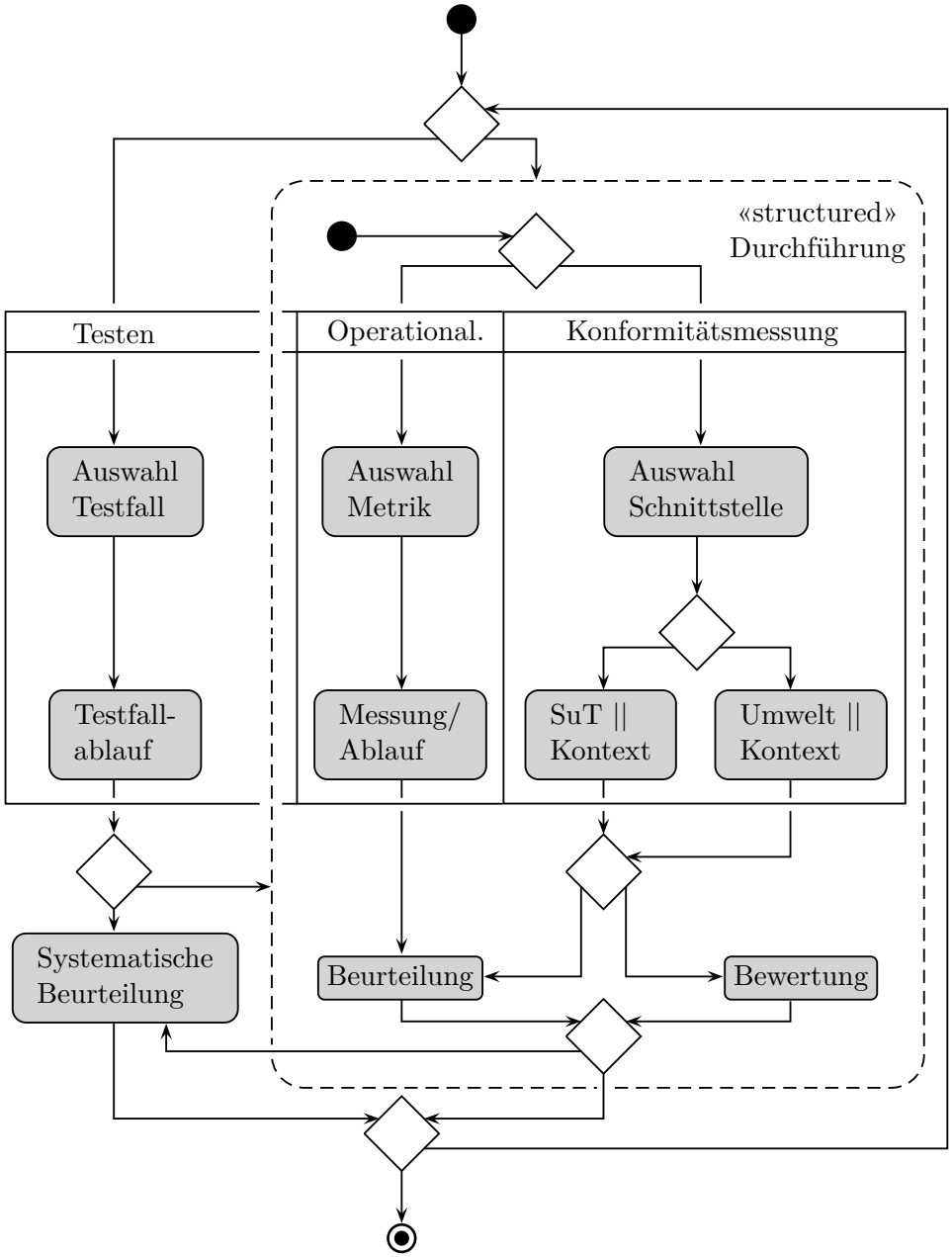


Abbildung 4.1.: Iterativer Evaluierungsprozess

Betriebssystem
- Zuverlässigkeit (Zuv.)
- Interaktion
- Verfügbarkeit
- Gleichzeitigkeit
- Effizienz
- Bedienbarkeit
- Rechtzeitigkeit
- Reaktivität
- Determinismus (Det.)
- Zeitliche Vorhersagbarkeit (Vor.)
- Wiederverwendbarkeit

Kommunikation
- Zustandsüberwachung
- Zeitige Fehlererkennung
- Datenintegrität
- Unabhängigkeit

Datenverarbeitung (DV)
- Persistenz-freie DV
- Datenintegrität
- Robust gegen Eingabefehler
- Persistente Datenspeicherung
- Determinismus bezüglich der Ausgabe (D.b.A.)

Scheduling
- Priorisiertes Blocken
- Deadlock(freiheit)
- Koordination und Ablauf
- Interrupt (Scheduler)
- Synchronisation
- Verwaltung der Ressourcen

Zeitgeber
- Watchdog
- Interrupt (Zeitgeber)
- Zeitgeber
- Anpassung

Speicherverwaltung
- Speicherbereinigung
- Speicherallokation
- Gemeinsamer Speicher
- Echtzeitwarteschlange
- Speicherschutz
- Verschiebbarkeit

Alle Kategorien
- Echtzeit

Tabelle 4.1.: Auflistung der zeitkritischen Anforderungen

4.2. Anforderungskatalog

Bei der Durchführung der Kooperationsprojekte mit der Licht- und Signaltechnik sowie der Automobilindustrie wurde sichtbar, dass während der Entwicklung von Echtzeitsystemen die zeitkritischen Anforderung an einzelne System- beziehungsweise Softwarekomponenten oft nicht konkret genug spezifiziert sind. Umgekehrt war dann auch nicht sichtbar, welchen Bezug diese konkreten Anforderungen auf die Entwicklung dieser Komponenten auch in Hinsicht auf weitere zeitkritischen Anforderungen haben. Deswegen werden in diesem Abschnitt die für die Problemstellung relevanten zeitkritischen

beziehungsweise zeitkritischen Anforderungen zusammengefasst und einerseits bezüglich der System-beziehungsweise Softwarekomponenten kategorisiert sowie andererseits in Abhängigkeit zueinander gesetzt. Dazu listet Tabelle 4.1 die Anforderungen nach den folgenden sechs Kategorien sortiert auf. Kommunikation zwischen einzelnen System- oder Softwarekomponenten, Datenverarbeitung über die Eingabe bis zur Ausgabe der Daten und die Kategorie Betriebssystem, welche wiederum aus den Kategorien Scheduling, Speicherverwaltung und Zeitgeber besteht. Dabei muss ein Echtzeitsystem nicht zwingend ein vollständiges Betriebssystem einsetzen oder realisieren. Die Angabe der Kategorie Betriebssystem bedeutet in diesem Kontext dann, dass die katalogisierte Anforderung in allen drei Unterkategorien Scheduling, Speicherverwaltung und Zeitgeber berücksichtigt werden muss. Die Zusammenfassung der zeitkritischen Anforderungen in diesem Abschnitt wird in Anhang A durch eine detaillierte Beschreibung ergänzt. Diese erfolgt auch deswegen im Anhang, um die Wiederverwendbarkeit zu erhöhen.

Die Bezeichnung einer Anforderung ist bezüglich der Zeitkritizität nicht immer selbsterklärend. Stattdessen ergibt sich die Zeitkritizität aus dem Kontext der jeweiligen Funktionalität. Ein Beispiel dafür ist die Anforderung Zustandsüberwachung aus der Kategorie Kommunikation: Wie in Abschnitt A.34 beschrieben überwachen sich entweder Komponenten eines einzelnen Echtzeitsystems oder mehrere Echtzeitsysteme innerhalb eines Netzwerkes gegenseitig. Dazu muss die Kommunikation zwischen den Komponenten oder innerhalb des Netzwerkes zeitkritische Anforderungen erfüllen, damit diese Überwachung unter Echtzeitbedingungen durchgeführt werden kann. Als ein weiteres Beispiel ist die Anforderung Zeitgeber, die im Kontext hier dann die Präzision, Genauigkeit und Rechtzeitigkeit spezifiziert.

Im Rahmen dieser detaillierten Beschreibung werden auch die Abhängigkeiten der Anforderungen untereinander aufgeführt, die in Abbildung 4.2 in Form eines Abhängigkeitsgraphen dargestellt sind. Die Pfeile in diesem Abhängigkeitsgraphen zeigen von Anforderungen, die als Eigenschaft eines Systems oder einer Software vorausgesetzt werden, auf diejenigen Anforderungen, welche durch diese bereits vorhandenen Eigenschaften erst erfüllbar werden. Dabei ist das essentielle Ergebnis bei der Entwicklung von Echtzeitsystemen die Realisierung der Echtzeitfähigkeit des Systems oder der Software. Diese Echtzeitfähigkeit ist wie in Kapitel 2.1.2 auf Seite 10 beschrieben im Grundsatz eine generische nicht-funktionale Anforderung, wenn sie für ein ganzes System beziehungsweise Software spezifiziert wird. Aus diesem Grund ist die Echtzeitfähigkeit beziehungsweise der Knoten *Echtzeit* im Abhängigkeitsgraphen stets der Zielknoten, auf dem alle möglichen Pfade des Graphen enden. Auf diesen Pfaden befinden sich weitere Knoten, die zeitkritische Anforderungen repräsentieren, und damit die Abhängigkeit der einzelnen Anforderungen untereinander aber auch transitiv interpretiert den Bezug zur Echtzeitfähigkeit des Systems beziehungsweise der Software skizzieren.

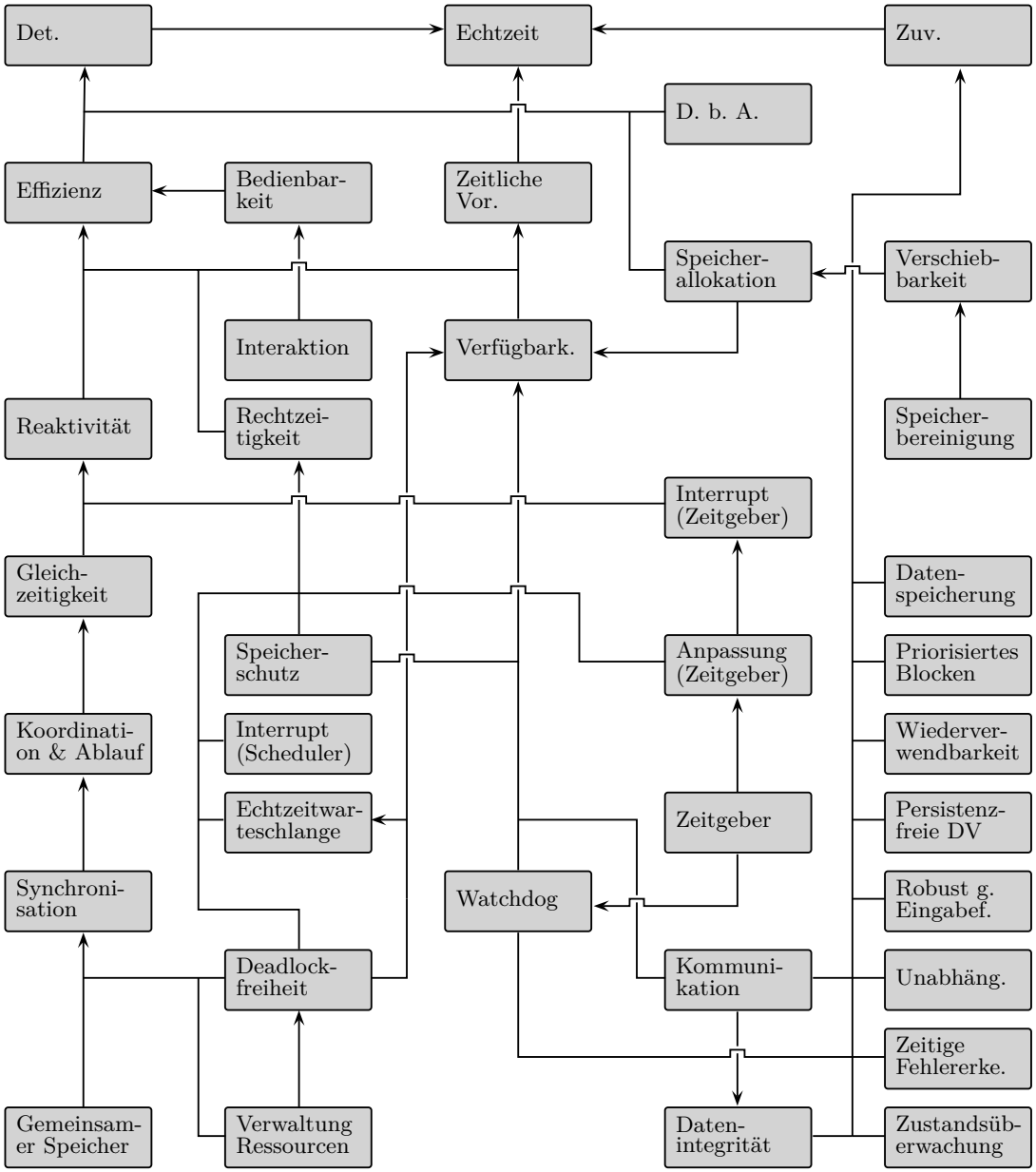


Abbildung 4.2.: Abhängigkeitsgraph zeitkritischer Anforderungen

Aus diesem Grund kann der Abhängigkeitsgraph in Abbildung 4.2 eingesetzt werden, zeitkritische Anforderungen zu bestimmen, die zur Umsetzung einer spezifischen Anforderung erfüllt sein müssen. Zum Beispiel müssen die Komponenten eines Systems, das mit einer integeren Datenverarbeitung (siehe Details in Abschnitt A.17) realisiert werden soll, zur Laufzeit des Systems zuverlässig ihre Funktion erfüllen. Ein weiteres Beispiel ist die Umsetzung gemeinsamen Speichers innerhalb eines Systems oder gegebenenfalls systemübergreifend. Alle Systemkomponenten oder Systeme, die über diesen gemeinsamen Speicher kommunizieren, müssen einer zeitkritischen Synchronisierung unterliegen, die zum Beispiel die korrekte Reihenfolge der Zugriffe sicherstellt. Auch diese Anwendung des Abhängigkeitsgraphen ist transitiv. Damit ein Echtzeitsystem eine spezifische Anforderung erfüllen kann, müssen alle Anforderungen im Abhängigkeitsgraphen, die auf dem Pfad von dieser Anforderung bis hin zum Echtzeitknoten liegen, als Eigenschaften innerhalb des System realisiert werden. Auf dem Abhängigkeitsgraphen ist also eine Rückwärtssuche möglich, um immer dann zeitkritischen Anforderungen an ein zu entwickelndes System oder Software beziehungsweise ihre Komponenten zu identifizieren, wenn eine spezifische, technisch erforderliche Anforderung an eben dieses System, Software oder Komponente gestellt wird.

Die in dieser Zusammenfassung vorgestellten zeitkritischen Anforderungen wurden teilweise bereits in verwandten Arbeiten formuliert und diskutiert. Eine Abgrenzung zwischen Leistung und Echtzeit von Systemen beziehungsweise Software lieferte John Stankovic [72] bereits 1988. Zeitkritische Anforderungen an die Verarbeitungen großer Datenmengen in kurzer Zeit wurden von Michael Stonebraker [75] diskutiert. Heinz Wörn und Uwe Brinkschulte [82] sowie Bruce Powell Douglass [11] skizzieren auch zeitkritische Anforderungen, Hermann Kopetz [36] liefert eine Zusammenfassung über eingebettete, zeitkritische Systeme. Während der laufenden Projekte wurden hier vorgestellte Anforderungen teilweise in [4, 15, 20, 32, 63] behandelt. Anhang A referenziert die verwandten Arbeiten noch einmal spezifisch für jede einzelne Anforderung.

4.3. Konformitätsüberprüfung physikalischer Signale

4.4. Konformitätsüberprüfung physikalischer Signale

4.5. Konformitätsüberprüfung physikalischer Signale

Während des Kooperationsprojektes mit der Automobilindustrie wurde eine Konformitätsüberprüfung physikalischer Signale in der Programmiersprache Java implementiert. Das Ergebnis war ein Werkzeug, mit dem Messdaten von physikalischen Signalen durch

den Vergleich mit realen oder simulierten Referenzsignalen untersucht werden können. Dieses Werkzeug wurde eingesetzt, um die Verarbeitung physikalischer Signale in einem komplexen Testwerkzeug des Kooperationspartners zu überprüfen, mit dem Steuerungen von Verbrennungsmotoren getestet werden. Die Messung der durch das Werkzeug später ausgewerteten Messdaten findet dabei an genau den Schnittstellen gemäß Kapitel 3 als Blackbox Test statt, mit denen das Testwerkzeug Messdaten von den Motorensteuerungen erhält. Das Werkzeug zur Konformitätsüberprüfung soll dabei einerseits Latenzen in den gemessenen Signalen sichtbar machen, die durch die Verarbeitung des Signals innerhalb des komplexen Testwerkzeugs entstehen, andererseits sollen Fehler im Signal bestimmt werden. Zum Beispiel dürfen keine Messdaten während der Tests der Motorensteuerung verloren gehen, da sonst die gesamte Messung unverwertbar wird. Durch den Einsatz des Werkzeugs zur Konformitätsüberprüfung muss also gezeigt werden, dass die Messdaten durch das Testwerkzeug in Echtzeit verarbeitet werden.

In diesem Abschnitt wird die Umsetzung des für den Kooperationspartner entwickelten Werkzeugs als konkrete Realisierung einer Konformitätsprüfung vorgestellt. Dazu wird in Abschnitt 4.5.1 der Funktionsumfang des Werkzeugs skizziert. Danach werden in Abschnitt 4.5.2 die implementierten Algorithmen zusammengefasst, welche die Konformitätsüberprüfung ermöglichen. Nach der Vorstellung exemplarischer Ergebnisse in Abschnitt 4.5.3 werden im letzten Abschnitt verwandte und weiterführende Arbeiten besprochen.

4.5.1. Überblick

Das Aktivitätsdiagramm in Abbildung 4.3 stellt die Durchführung einer Konformitätsüberprüfung schematisch dar. Innerhalb des Testaufbaus wird ein physikalisches Signal aus einer realen oder simulierten Quelle erzeugt und dann sowohl in das zu testende System eingegeben als auch der Konformitätsmessung als Referenzsignal zur Verfügung gestellt. Dieses Signal wird im zu testenden System verarbeitet und dann ebenfalls an die Konformitätsmessung gegeben. Dort wird das Signal entweder direkt gemessen (quantifiziert) oder, wenn eine direkte Messung nicht möglich ist, mit Hilfe einer Metrik messbar gemacht, also zum Beispiel gegen das Referenzsignal auf Konformität überprüft. Die Messdaten und Ergebnisse werden dann für spätere Auswertungen persistent gespeichert.

Der Testaufbau im Rahmen des Kooperationsprojektes sah dabei vor, dass das Werkzeug zur Konformitätsüberprüfung innerhalb einer Werkzeugkette eingesetzt wird. Diese Werkzeugkette ermöglicht eine nachgelagerte Konformitätsmessung, die damit selbst keinen zeitkritischen Anforderungen mehr unterliegt, indem die Messdaten durch ein proprietäres Gerät zur echtzeitfähigen Datenakquisition persistent gespeichert werden.

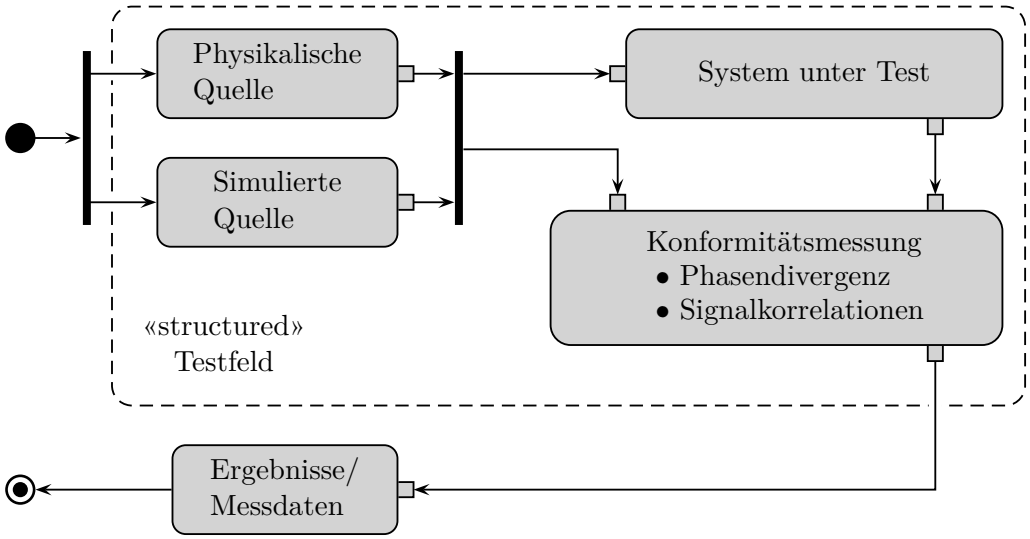


Abbildung 4.3.: Schema einer Konformitätsmessung

$$s_i - r_i >= 0$$
$$|s_i - r_i| < |s_i - r_{i-1}|$$

Abbildung 4.4.: Glättungsbedingung des Medianfilters

4.5.2. Verwendete Algorithmen

Dieser Abschnitt gibt einen Überblick über die im Werkzeug zur Konformitätsüberprüfung implementierten Algorithmen zur Auswertung von physikalischen Signalen.

Durch die Verarbeitung des Testsignals im zu testenden System verrauscht dieses Signal und es entstehen lokale Extremstellen, welche die Ergebnisse der Konformitätsüberprüfung des Signals beeinflussen. Aus diesem Grund werden im ersten Schritt der Konformitätsüberprüfung die rohen Messdaten mit Hilfe eines Medianfilters geglättet. Dieser einfache Algorithmus hat eine lineare Komplexität in der Länge der Signalaufzeichnung mit fixer Intervallbreite; Während der Entwicklung des Werkzeugs wurde empirisch eine Intervallbreite des Filters von sieben Funktionswerten als optimal bestimmt, wenn die Frequenz der Messung unter 1000Hz liegt.

Das geglättete Signal kann nun auf Konformität mit dem Referenzsignal geprüft werden. Dazu muss im zweiten Schritt des Algorithmus die Phasendivergenz der beiden

Signale bestimmt werden, die im Allgemeinen statistisch ermittelt werden muss. Diese Divergenz ist die Latenz, die bei der Verarbeitung des Signals im zu testenden System auftritt. Um diese zu bestimmen, werden die Funktionswerte des Signals mit einer passender Referenz verglichen. Typische während des Kooperationsprojekts dazu eingesetzte physikalische Signale waren Sägezahn- und Rechtecksignale. Die Funktionsverläufe dieser Signale sind nicht stetig und nicht an jeder Stelle ableitbar. Weiterhin kann a priori angenommen werden, dass das zu testende Signal dem Referenzsignal nachläuft. Der zur Bestimmung der Phasendivergenz implementierte Algorithmus wurde deswegen wie folgt implementiert:

1. Bestimme alle unstetigen Stellen im Referenzsignal
   Bestimme alle unstetigen Stellen im zu testenden Signal
2. Korreliere je eine unstetige Stelle aus Referenzsignal und Signal
3. Der Betrag der Differenz zwischen einer Stelle im Signal und der korrelierten Stelle im Referenzsignal ist die Phasendivergenz dieser Stelle. Innerhalb der mit der Implementierung einhergehenden Fallstudie sind keine falschen unstetigen Stellen sichtbar geworden und wurden deswegen auch nicht explizit behandelt.
4. Das statistische Mittel über allen lokalen Phasendivergenzen entspricht der Divergenz des zu testenden Signals

Die Bestimmung dieser unstetigen Stellen erfolgt mit einer einfacher Heuristik: Ist der Betrag der Differenz zweier benachbarter Funktionswerte größer als das 10-fache des Erwartungswertes, erfolgt ein unstetiger Sprung zwischen den Funktionswerten. Der Erwartungswert zum Beispiel für ein Sägezahnsignal ist dabei der Quotient aus Amplitude und der Anzahl Abtastungen während einer Periode des Signals. Diese Heuristik kann natürlich beliebig verfeinert werden. Die genannte Granularität reichte aber im Rahmen des Kooperationsprojektes aus. Eine Korrelation zwischen einer unstetigen Stelle des Signals s_i und der Referenz r_i liegt genau dann vor, wenn die Bedingungen in Abbildung 4.4 gelten. x_i bezeichnet Signal- oder Referenzwert zum Zeitpunkt i. Der Algorithmus arbeitet dabei unter der Annahme korrekt, dass die Referenz dem physikalischen Signal vorläuft, die Phase also immer positiv oder null ist. In dieser Version kann der Algorithmus keine Phasendivergenzen erkennen, die größer sind als die Periode des zu testenden Signals. Diese Funktion war für die Konformitätsüberprüfungen im Kooperationsprojekt nicht erforderlich. Der Algorithmus ist linear in der Anzahl Datensätze, die durch die Signalabtastung während der Messung gespeichert wurden.

Im dritten Schritt der implementierten Konformitätsüberprüfung wird die berechnete Phasendivergenz aus dem zu testenden Signal herausgerechnet. Dann wird im vierten

Schritt durch partielles Integrieren an jeder Funktionsstelle über die Differenz der beiden Funktionswerte von Referenz und zu testendem Signal die Fläche zwischen den beiden Signalverläufen bestimmt. Diese Fläche wird dann nicht-automatisch beurteilt.

$$v_i := \frac{n(\sum t_i f_i) - (\sum t_i)(\sum t_i)}{n(\sum t_i^2) - (\sum t_i)^2}$$

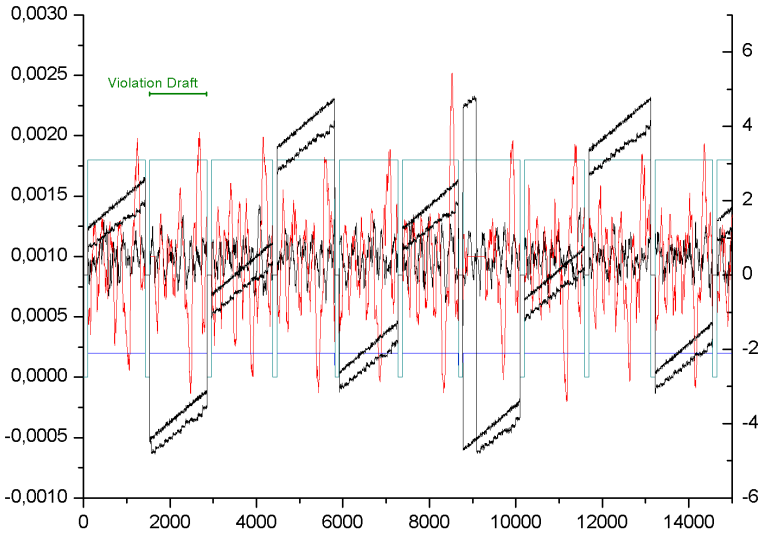
Formel 4.1: Lineare Regression [13]

Um diese nicht-automatische Beurteilung weiter zu unterstützen, wurde ein zusätzlicher Algorithmus im Werkzeug zur Konformitätsüberprüfung implementiert. Dazu wurde im ersten Schritt mit Hilfe der Formel aus Formal 4.1 die Ableitungsfunktion des zu testenden Signals berechnet, was einer linearen Regression entspricht. Der Algorithmus bestimmt an jeder Funktionsstelle des Signals die Steigung durch Bestimmung der Regressionskoeffizienten, dabei entspricht in der Abbildung v_i der Regressionsgeraden im Punkt $(t, f)_i$. Diese numerische Berechnung der Steigung über Intervallbreiten von mindestens 200 Funktionsstellen ist ebenfalls linear in der Anzahl der Datensätze, wird aber durch die Intervallbreite mitbestimmt und verbraucht messbar viel Rechenzeit pro Iteration. Das war für das Werkzeug zur Konformitätsüberprüfung aber zu vernachlässigen, da die Auswertungen mit Hilfe des Werkzeugs im Rahmen des Kooperationsprojektes dem Testlauf nachgelagert stattgefunden haben. Fehler innerhalb des zu testenden Signals wirken sich überproportional auf die Ableitungsfunktion aus und werden durch Auswertung dieser Funktion besser detektierbar. Dies geschieht mit Hilfe einer Einhüllenden, die um die Ableitungsfunktion gelegt wird. Im zweiten Schritt überprüft der zusätzliche Algorithmus dann, an welchen Stellen die Ableitungsfunktion die Hülle verlässt.

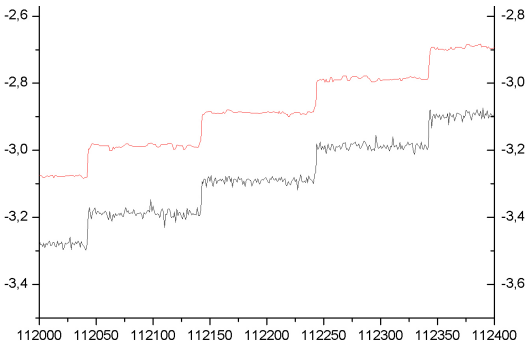
Das Werkzeug zur Konformitätsüberprüfung speichert alle Ergebnisse der beiden Algorithmen persistent in der Form von Komma-separierten Werten oder Textdateien. Zusammen mit den Datensätze der Messung wurden diese Ergebnisse skriptbasiert mit Hilfe eines weiteren Werkzeugs, das beim Kooperationspartner eingesetzt wird, grafisch aufbereitet. Diese Diagramme werden im folgenden Abschnitt zur Darstellung der Ergebnisse eingesetzt.

4.5.3. Ergebnisse

Die Abbildungen 4.5 und 4.6 veranschaulichen den Einsatz des Werkzeugs zur Konformitätsüberprüfung anhand einer konkreten während des Kooperationsprojekts durchgeführten Messung und Beurteilung dieser Messung. Die dargestellten Ergebnisse entstanden bei der Vermessung des zu testenden Systems an einer 100Hz CAN Bus Schnittstelle

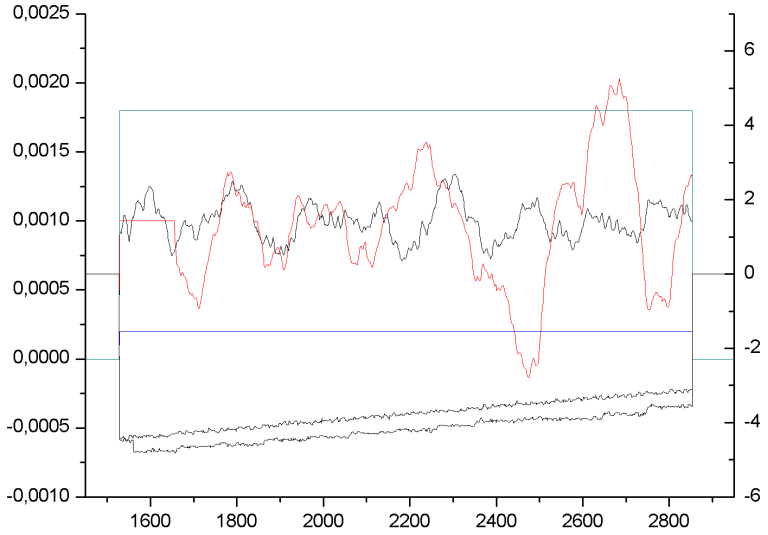


(a) Ausgaben der eingesetzten Algorithmen

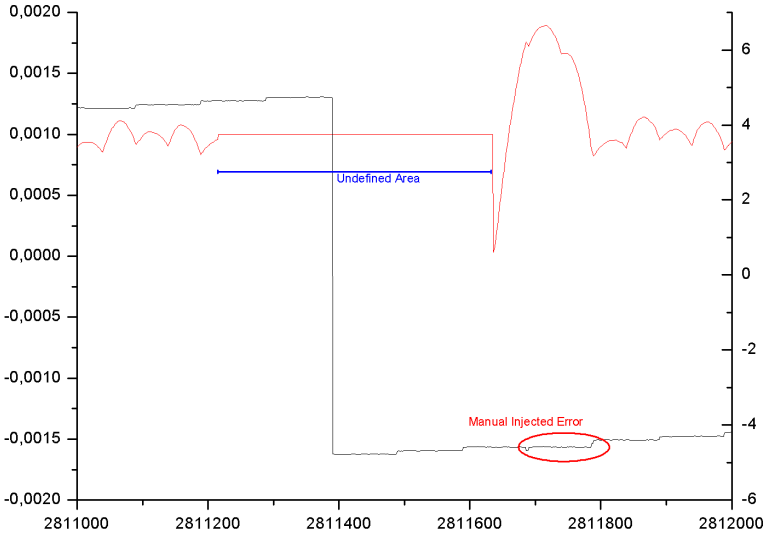


(b) Rohes und Gefiltertes Signal

Abbildung 4.5.: Ergebnisse der Konformitätsüberprüfung



(a) Anwendung



(b) Konzept

Abbildung 4.6.: Fehlerdetektion mit Einhüllenden

mit 10kHz Wandlungsrate. Das Testsignal ist eine Sägezahnkurve, die bedingt durch die Abtastrate in einhundert Stufen vom Minimum zum Maximum steigt. Die horizontale Achse der Diagramme entspricht dem Zähler der Abtastung, die Funktionswerte auf den vertikalen Achsen entsprechen proportional der angelegten Spannung zum Zeitpunkt der Abtastung beziehungsweise der einheitenlosen Steigung der ausgemessenen Funktion.

In Abbildung 4.5(b) wird das verrauschte Signal (schwarz) mit dem Ergebnis der Medianfilterung (rot) verglichen, wobei die im folgenden dargestellten Ergebnisse immer auf der Beurteilung des bereits gefilterten Signals basieren. Die Darstellung in Abbildung 4.5(a) wurde so gewählt, dass die wesentlichen Ergebnisse der Algorithmen deutlich werden. Die zeitliche Verschiebung der Signale, die Phasendivergenz, ist insbesondere um den Index 9000 herum gut erkennbar. Referenz und Signal (beide schwarz, treppenförmig) sind außer Phase und umschließen bedingt durch diese Phasendivergenz der Signalverläufe sichtbare Flächen. Während der Durchführung des Algorithmus werden diese Flächen reduziert, indem vor der Integration die Phasendivergenz aus dem zu testenden Signal herausgerechnet wird. Die Abbildung zeigt auch die Ableitungsfunktionen der beiden Signalverläufe (rot und schwarz, schwingend) sowie die Einhüllende (blau). Dabei ist zu erkennen, dass die Ableitungsfunktion des zu testenden Signals (rot) die Hülle verlässt, die Ableitungsfunktion des Referenzsignals aber nicht. Dadurch werden die Funktionsstellen des zu testenden Signals identifizierbar, die fehlerbehaftet sind. In Abbildung 4.6 werden die Ergebnisse des zusätzlichen Algorithmus veranschaulicht. Abbildung 4.6(b) zeigt den Verlauf des zu testenden Signals und seiner Ableitungsfunktion um eine unstetige Stelle herum. Deutlich ist zu erkennen, dass auch die Ableitung an dieser Stelle nicht definiert ist und nicht berechnet werden kann (blau markiert). In das gezeigte Signal wurde ein Fehlerfall synthetisiert (rot markiert). In der Ableitungsfunktion wird dieser Fehler durch eine schnelle Ab- und dann Aufschwingung sichtbar. Diese Eigenschaft wird im Algorithmus wie in Abbildung 4.6(a) zu sehen genutzt, um mit Hilfe von Einhüllenden (blau) den Fehler zu detektieren. An den unstetigen Stellen des zu testenden Signals können je nach Breite des zur linearen Regression genutzten Intervalls Fehler nicht detektiert werden. Verringert man die Intervallbreite erhöht man im Umkehrschluss die Fluktuation der Ableitungsfunktion. Im praktischen Einsatz muss dieses Verhalten berücksichtigt werden, indem Intervallbreite und der Abstand der Einhüllenden entsprechend parametrisiert sind. Das Werkzeug zur Konformitätsüberprüfung wurde bei Projektabschluss erfolgreich eingesetzt, um das Testwerkzeug für Motorensteuerungen auszumessen.

4.5.4. Für die Konformitätsmessung relevante Arbeiten

Die in diesem Abschnitt vorgestellte Implementierung beruht auf analytischen Verfahren wie der linearen Regression [13] und Ansätzen aus der digitalen Signalverarbeitung [70] sowie einschlägiger Literatur zur Verarbeitung von digitalisierten Daten [34, 38]. Dabei lag der Forschungsschwerpunkt des Kooperationsprojektes nicht auf der Entwicklung neuer Methoden zur Konformitätsüberprüfung physikalischer Signale. Aus diesem Grund wurden bekannte Algorithmen ausgewählt, um das Testwerkzeug des Kooperationspartners zügig implementieren zu können. Fortschritte bei der Konformitätsüberprüfung physikalischer Signale sind aber ebenfalls Teil aktueller Forschung, zum Beispiel die Anwendbarkeit von Kreuzkorrelationen [51] oder der Integration in modellbasierten Entwicklungsverfahren für die Software eingebetteter Systeme [52, 53].

4.6. Lebenszyklusabdeckende Operationalisierung

Während der Durchführung der Kooperationsprojekte wurden die zu entwickelnden Softwarekomponenten immer wieder getestet. Die zu testenden zeitkritischen Eigenschaften mussten dabei mit Hilfe einer geeigneten Operationalisierung messbar gemacht werden. Dazu wurden passende Metriken gesucht, die in Abschnitt 4.6.6 in Form eines kleinen Kataloges vorgestellt werden. Für die Beurteilung der Schedulingmechanismen der zu testenden Software musste allerdings eine eigene Metrik entwickelt werden. Da in immer mehr Echtzeitsystemen solche Schedulingmechanismen eingesetzt werden, entsteht hier ein wissenschaftliches Interesse, das über den Erfolg des einzelnen Kooperationsprojekts hinaus geht. Die Operationalisierung ist dann in Abhängigkeit zur jeweiligen ausgewählten Metrik eine Messung an den potentiellen Schnittstellen des zu testenden Systems. Die aus der Messung oder dem Test resultierenden Ergebnisse können nun quantifiziert oder über die ausgewählte Metrik beurteilt werden. Da passende Messergebnisse auch durch eine Konformitätsmessung ermittelt werden können, kann diese Stufe prinzipiell auf der ersten Stufe aufbauen.

In diesem Abschnitt wird eine neue Metrik vorgestellt, die im Rahmen einer lebenszyklenabdeckenden Operationalisierung verwendet werden kann. Zuerst wird in Abschnitt 4.6.1 ein Überblick gegeben und dann in Abschnitt 4.6.2 das Konzept der Metrik beschrieben. Da die neue Metrik auf dem jeweiligen Lebenszyklen der Prozesse der zu testenden Plattform aufbaut, wird mit Bezug auf das Betriebssystem Linux in Abschnitt 4.6.3 eine exemplarische Liste von Testfällen beschrieben, die auf dem in Abbildung 2.2 auf Seite 13 dargestellten Lebenszyklusmodell von Linuxprozessen basiert. Zur Evaluierung der neuen Metrik wird in Abschnitt 4.6.4 eine Fallstudie vorgestellt, die den Testfällen folgend auf einem gepatchtem Linux Derivat durchgeführt wurde, das Prozesse in Echtzeit

ausführen kann. Die Evaluierung der neuen Methode wird in Abschnitt 4.6.5 durch eine Diskussion abgeschlossen.

4.6.1. Überblick

Die hier vorgestellte Metrik ermöglicht eine Beurteilung von Prozessen, die durch einen Schedulingmechanismen geplant werden. Zusätzlich soll mit einer möglichst geringen Anzahl an Testfällen der gesamte Lebenszyklus dieser Prozesse abgedeckt sein. Um die Metrik anwenden zu können, müssen die Brutto- und die Nettolaufzeiten der zu testenden Prozesse gemessen werden, die wie folgt definiert sind. Weiterhin wird gemessen, zu welchen Zeitpunkten der Prozess aktiv war.

Definition 5: Brutto- und Nettolaufzeit, Aktivität von planbaren Prozessen Für einen Prozess, der durch einen Schedulingmechanimus geplant wird, ist die

- Bruttolaufzeit die Zeit, die der Prozess im Zustand *running* verbracht hat, also ausgeführt wurde.
- Nettolaufzeit die Zeit, wie lange der durch den Prozess zur Ausführung kommende Algorithmus tatsächlich kalkuliert hat.
- Aktivitätszeit ein Zeitpunkt im System, bei deren Messung ein spezifischer Prozess aktiv war.

◇

Um die beiden Laufzeiten und die Aktivität messen zu können, muss ein entsprechend exakter und trotzdem möglichst leistungsstarker Zeitgeber zur Verfügung stehen, da die Genauigkeit des Zeitgebers einen direkten Einfluss auf die Genauigkeit der Messdaten hat. Die Verfügbarkeit eines solchen Zeitgebers kann nicht a priori vorausgesetzt werden. Deswegen müssen die Zeitgeber im Rahmen der Operationalisierung ebenfalls berücksichtigt und während einem Testlauf ausgemessen werden. Weiterhin wird die Nettolaufzeit der Prozesse durch die Vorgänge im Schedulingmechanismus bestimmt. Wird diese nicht durch eine Funktion oder Eigenschaft der Softwareplattform bereitgestellt, muss die zu testende Software beziehungsweise die entsprechende Softwareplattform erweitert werden. Aus technischen Gründen werden während einer Messung viele Aktivitätszeiten eines Prozesses gemessen. Der Begriff *Aktivität* abstrahiert von dieser Messreihe und bezeichnet ein Intervall, in dem ein Prozess nachweisbar durch eben diese Messreihe aktiv war, beziehungsweise steht für alle Aktivitäten innerhalb einer Messreihe.

Definition 6: Lebenszyklusabdeckende Operationalisierung Eine Operationalisierung zur Beurteilung von Prozessen, die durch einen Schedulingmechanimus geplant werden, ist dann eine lebenszyklusabdeckende Operationalisierung, wenn jede Transition des Lebenszyklus von einem Zustand in den nächsten durch mindestens eine Testfall validiert wird. ◇

Die in diesem Abschnitt beschriebene Metrik liefert dazu die Messgrößen und die Erhebungsmethode, Abschnitt 4.6.3 eine exemplarische Messinstrumentalisierung. Mit Hilfe dieser Operationalisierung werden zeitkritische Eigenschaften eines zu testenden Systems messbar gemacht, die durch den Schedulingmechanismus beeinflusst werden. Das sind bezüglich der Problemstellung die in Tabelle 4.2 aufgelisteten Anforderungen, die in Abschnitt 4.2 und in Anhang A weiter beschrieben werden.

A.2	Determinismus
A.3	Zuverlässigkeit
A.4	Zeitliche Vorhersagbarkeit
A.5	Gleichzeitigkeit
A.6	Rechtzeitigkeit
A.7	Reaktionsfreudigkeit
A.8	Koordination und Ablauf
A.9	Verwaltung der Ressourcen
A.10	Deadlockfreiheit
A.11	Interrupt (Scheduler)
A.12	Verfügbarkeit
A.18	Effizienz
A.24	Wiederverwendbarkeit
A.25	Interaktion
A.27	Bedienbarkeit
A.28	Synchronisation

Tabelle 4.2.: Messbare zeitkritische Anforderungen

4.6.2. Konzept

Während einer Messung werden die Brutto- und Nettolaufzeiten des zu testenden Prozesses sowie die Systemzeit der Aktivität gemessen und im Arbeitsspeicher vorgehalten. Aus diesen Messdaten werden dann Intervallbreiten der Aktivität und Inaktivität des zu testenden Prozesses, die exklusive Prozessorzeit und der Anteil der Kontextwechsel

berechnet. Zusätzlich wird die Regelmäßigkeit von zufälligen und zyklischen Ereignissen untersucht. Die Vorgehensweise während der Messung beruht auf den Testfällen, die auf dem Testfeld ausgeführt werden. Dabei werden die folgenden Schritte für jeden Testfall so oft wie möglich iteriert. Der Zustand des zu testenden Prozesses wird auf *ausführbar* gewechselt. Im Lebenszyklus von Linuxprozessen entspricht das dem Zustand `TASK_RUNNING`, siehe Abbildung 2.2. Dann wird die aktuelle Systemzeit ermittelt sowie vorgehalten und danach der eigentliche Testfall abgearbeitet. Nachdem die zu testenden Prozesse in Abhängigkeit zum konkreten Testfall pausiert oder ihre Laufzeit aufgebraucht haben, wird die Bruttolaufzeit des zu testenden Prozesses auf der bisherigen Bruttolaufzeit und der Differenz aus der aktuellen und der vor der Abarbeitung gespeicherten Systemzeit ermittelt. Die Nettolaufzeit kann entweder über eine bereits durch die entsprechende Softwareplattform bereitgestellte Funktion oder Eigenschaft ermittelt werden oder muss durch eine Erweiterung der Plattform zur Verfügung gestellt werden. Für diese Vorgehensweise wertet ein messender Prozess die Messdaten aus und verarbeitet diese. Dann wird der Messprozess beendet und eine neue Iteration initiiert. Für eine verlässliche Statistik sollte jeder einzelne Testfall so oft wie möglich oder bis eindeutige Ergebnisse vorliegen durchgeführt werden.

Intervallbreiten der Aktivität und Inaktivität des zu testenden Prozesses. Nachdem die Messung durchgeführt wurde, können aus den Systemzeiten und der Aktivität des Prozesses die einzelnen Intervalle berechnet werden, in welchen dieser Prozess aktiv war. Die Inaktivität des Prozesses ergibt sich dadurch implizit ebenfalls. Das macht messbar, wie oft innerhalb einer Sekunde ein Prozess aufgerufen wurde und die zur Verfügung gestellte Prozessorzeit nicht ausreichend war. Zusätzlich liefert der Quotient aus Brutto- und Nettolaufzeit, gemittelt über alle Testläufe, ein Maß für die Effizienz des Schedulingmechanismus.

Exklusive Prozessorzeit Der Kehrwert der Effizienz, also der Quotient aus Netto- und Bruttolaufzeit, ist die exklusive Prozessorzeit des zu testenden Prozesses als prozentualer Wert. Der Quotient beschreibt also den Anteil der Ausführung im Verhältnis zur gesamten Laufzeit des Prozesses. Dieser Wert ist insbesondere in Testfällen aussagekräftig, in denen zusätzlich zum zu testenden Prozess eine Last erzeugt wird. Es kann nicht nur gezeigt werden, dass der Schedulingmechanismus effizient ist, sondern diese Effizienz auch bei hoher Systemlast erhalten bleibt. Wird eine entsprechende Frist spezifiziert, kann auch ein prozentuales Qualitätsmaß berechnet werden, wie oft diese Frist während der Durchführung der Messung nicht eingehalten wurde.

Kontextwechsel Anhand der Systemzeiten, die zur Bestimmung der Aktivität des zu testenden Prozesses ausgemessen wurden, werden die Kontextwechsel berechnet, die durch den Schedulingmechanimus ausgelöst werden. Dazu wird eine Zeitfrist als Heuristik auf der Basis der vorhandene Messdaten festgelegt. Dann werden die Differenzen der einzelnen Aktivitäten nebeneinander berechnet. Ist eine Differenz größer als die Heuristik, wurde während der Messung der beiden Systemzeiten ein Kontextwechsel durchgeführt.

4.6.3. Exemplarische Testfälle für den Lebenszyklus von Linuxprozessen

In diesem Abschnitt werden Testfälle definiert, die im Rahmen einer Operationalisierung den gesamten Lebenszyklus von Linuxprozessen abdecken, der in Abbildung 2.2 auf Seite 13 dargestellt ist. Tabelle 4.3 fasst Testfälle zusammen, die eine Testfallabdeckung über die typischen Pfade diesen Lebenszyklus ermöglichen. Eine vollständige Pfadabdeckung zu erreichen ist nicht Teil der vorgelegten Arbeit. Sollte die neue Metrik auf anderen Plattformen eingesetzt werden, müssen neue Testfälle generiert und dafür gegebenenfalls der Lebenszyklus im Vorfeld analysiert werden. Für die Evaluierung der neuen Metrik in Abschnitt 4.6.4 nimmt die dort vorgestellte Fallstudie dieses Beispiel wieder auf und führt Messungen auf einem gepatchtem Linux Derivat durch, das Prozesse in Echtzeit ausführen kann.

#Test	Bezeichnung
1	Prozess berechnet eine einfache Funktion
1.1	Prozess gibt Rechenzeit freiwillig ab
1.2	Prozess beendet sich frühzeitig
1.3	Testfall 1.2, nach beliebiger Zeit
2	Testfall 1, bei hoher Prozessorlast
2.0.1	Testfall 2, mit niedriger Priorität
2.1	Testfall 1.1, bei hoher Prozessorlast
2.2	Testfall 1.2, bei hoher Prozessorlast
2.3	Testfall 1.3, bei hoher Prozessorlast

Tabelle 4.3.: Auflistung der Testfälle

In Testfall 1 berechnet der zu testende Prozess einfache arithmetische Funktionen. Diese Funktionen werden auch in den Testfällen 1.1, 1.2 und 1.3 berechnet, allerdings

wird hierzu zusätzliches Laufzeitverhalten vorausgesetzt. Der zu testende Prozess muss entweder freiwillig seine Prozessorzeit abgeben, sich frühzeitig beenden oder sich frühzeitig zu einem zufälligen Zeitpunkt während der Laufzeit beenden. Parallel dazu müssen die Testfälle 2 beziehungsweise 2.1, 2.2 und 2.3 durchgeführt werden. Zusätzlich wird ein weiterer Prozess ausgeführt, der während der Messung eine hohe Rechenlast erzeugt. In diesem Testaufbau hat der zu testende Prozess eine hohe Priorität. Um jetzt auch den Einfluss der Rechenlast auf einen niedrig-priorisierten zu testenden Prozess zu messen, wird mit Testfall 2.0.1 ein entsprechender Testfall in die Liste eingefügt. Die beschriebenen Testfälle decken den relevanten Teil des Lebenszyklen von Linuxprozessen, wie in Abbildung 2.2 dargestellt, ab. Der zu testende Prozess in Testfall 1 führt keine weiteren Systemaufrufe aus, sobald sich dieser Prozess im Zustand `TASK_RUNNING` befindet. Der Prozess kann deswegen nur zwischen den Zuständen `TASK_RUNNING` (bereit), `TASK_RUNNING` (ausgeführt) und `TASK_DEAD` wechseln. Außerdem wird der zu testende Prozess gesteuert durch den Testfall selbst in regelmäßigen Abständen in die Zustände `TASK_STOPPED` und `TASK_TRACED` versetzt und danach wieder zurückgesetzt. Testfall 1.1 erweitert die Testfallabdeckung auf den Zustand `TASK_INTERRUPTIBLE`. Dieser Zustand wird von Linuxprozessen sehr oft betreten, zum Beispiel durch den Systemaufruf `sleep`. Die Testfälle 1.2 und 1.3 erweitern dagegen nicht mehr die Testfallabdeckung, ermöglichen aber das Testen der Robustheit des Schedulingmechanismus bei der Anwesenheit von Fehlern. Die Testfälle 2.0 bis 2.3 entsprechen der ersten Testreihe mit zusätzlicher Rechenlast. Die Testfallabdeckung wird also nicht weiter erhöht, da mit diesen Testfällen das korrekte Laufzeitverhalten in Abhängigkeit des verwendeten Schedulingmechanismus getestet werden soll.

4.6.4. Fallstudie

In diesem Abschnitt wird die Fallstudie beschrieben, welche zur Evaluierung der neuen Metrik beziehungsweise der Operationalisierung durchgeführt wurde. Die Fallstudie basiert auf einem Standardcomputer auf dem das Linux Derivat Ubuntu 12.04 LTS als Softwareplattform installiert ist. Dessen Schedulingmechanismus wurde mit Hilfe des *Real-time Preemption Patch* [50][1] verbessert, in dem ein *Earliest-Deadline-First* Scheduler installiert ist, der echtzeitfähige Prozesse planen und ausführen kann. Die Evaluierung der neuen Methode wird in Abschnitt 4.6.5 durch eine Diskussion abgeschlossen. Sowohl der messende als auch der zu testende Prozess laufen im Benutzermodus ab, um das Laufzeitverhalten der Plattform möglichst wenig zu beeinträchtigen. Der Quelltext des zu testenden Prozesses wurde außerdem so wenig wie möglich verändert. Bei der

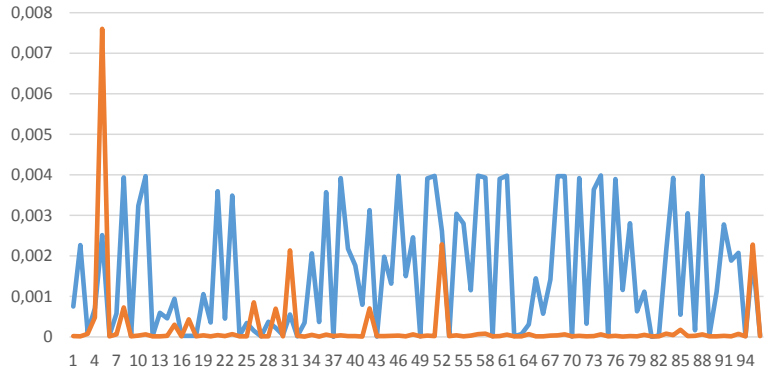
[1] OSADL: Open Source Automation Development Lab

Implementierung wurden ausschließlich Linux Standardbibliotheken und Header verwendet, die zum Beispiel im Debianpaket `build-essential` enthalten sind. Die Messdaten werden in einem kommagetrennten Format persistent gespeichert, die Kommandozeilenparameter werden mit Hilfe des Funktion `getopt` ausgelesen und ausgewertet. Die einzelnen Testfälle sind in der Programmiersprache *C*95 realisiert. Dazu wird über die Funktion `clock_gettime` die Systemzeit ausgelesen und in einen Datenpuffer geschrieben, der ausreichend groß gewählt ist. Im Anschluss an die Messung werden die Daten über die Funktion `ptrace` ausgelesen und persistent gespeichert. Muss im Rahmen eines Testfalls Rechenlast in einem System erzeugt werden, wurden dafür zwei weitere Funktionen implementiert. Die Nettolaufzeit von Linuxprozessen und damit auch des zu testenden Prozesses kann durch den Systemaufruf `getrusage` [42] ermittelt werden. Um statistisch aussagekräftige Messergebnisse sicherzustellen, werden die Messdurchläufe pro Testfall mindestens einhundert Mal wiederholt.

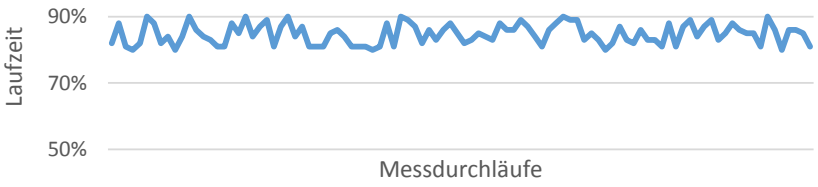
Abbildung 4.7 zeigt die Ergebnisse der Operationalisierung im Rahmen der Fallstudie. Die Intervallbreiten (rot) und Abstände (blau) der ersten 100 Durchführungen von Testfall 1 sind in Abbildung 4.7(a) dargestellt. Werden Anforderungen an die Aktivitätsintervalle oder die Frequenz eines durch den Schedulingmechanismus geplanten Prozesses spezifiziert, kann die Erfüllbarkeit über die Skala in Mikrosekunden abgelesen werden. Eine große Varianz der Aktivität entsteht dann, wenn der Schedulingmechanimus dem zu testenden Prozess nicht genügend Rechenzeit zur Verfügung stellt. In den Testfällen 1.1 und 2.1 werden die zu testenden Prozesse auch pausiert. Aus diesem Grund ist bei der Durchführung dieser Testfälle eine große Varianz der Inaktivität zu erwarten.

Die Nettolaufzeiten eines zu testenden Prozesses sollte auf derselben Plattform ausgeführt auf einem einzelnen Prozessor keine große Varianz zeigen. Im Gegensatz dazu schwankt die Bruttolaufzeit bei ineffizientem Scheduling. Während der Durchführung der Testfälle 1.1 und 2.1 wird zum Beispiel durch die Verwendung der Systemfunktion `sleep` die Bruttolaufzeit erhöht. Abbildung 4.7(b) zeigt eine Effizienz des Schedulingmechanimus während der Durchführung der Fallstudie zwischen 80 und 90 Prozent.

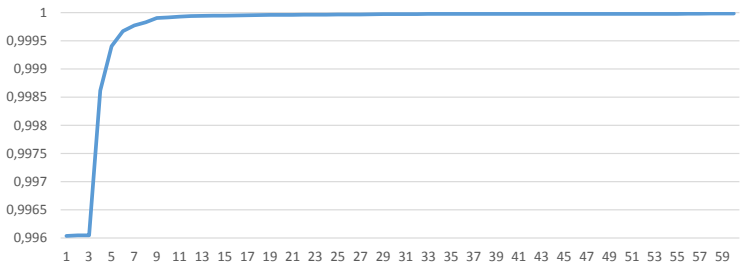
Abbildung 4.7(c) zeigt die Abstände der Aktivitätsmessung aus Testfall 1. Dabei wird die prozentuale Häufigkeit der Abstände gegenüber einer oberen Schranke der Abstandswerte in Mikrosekunden angegeben. Da bei der Durchführung von Testfall 1 nur eine kleine Menge der Abstandswerte größer als 10us war, entspricht die Häufigkeit ab diesem Wert 99.96%. Findet ein geplanter Kontextwechsel zwischen dem zu testenden und dem Rechenlast-erzeugenden Prozess statt, ist der Abstand zwischen der letzten und der nächsten Aktivität entsprechend größer als wenn die nächste Messiteration unmittelbar ausgeführt wird. Es ist auch anzunehmen, dass ein Kontextwechsel geplant durch den Schedulingmechanismus wesentlich länger dauert als eine Messiteration. Deswegen wird die heuristische Zeitfrist auf unter 3μs festgelegt, da die Abstandswerte in diesem Bereich



(a) Intervallbreiten und Abstände

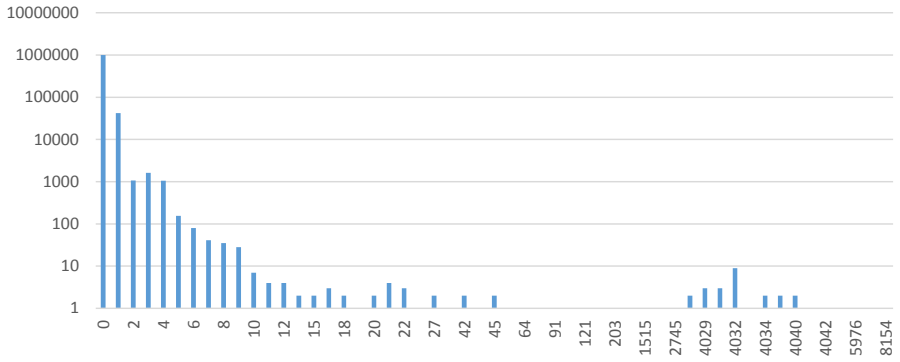


(b) Exklusive Prozessorzeit

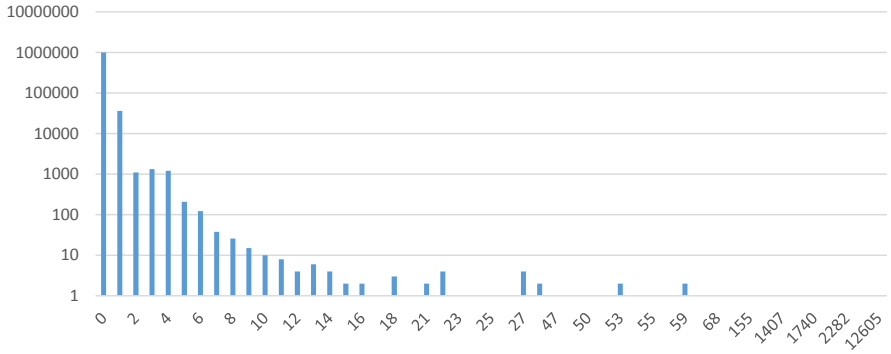


(c) Abstände der Aktivitätsmessung

Abbildung 4.7.: Ergebnisse der Operationalisierung



(a) Hohe Priorität (Testfall 2)



(b) Niderige Priorität (Testfall 2.0.1)

Abbildung 4.8.: Laufzeitverhalten der Testfälle 2 und 2.0.1

einen auffälligen Sprung machen. Dann findet in 99.6% der Fälle kein Kontextwechsel statt. Zusätzlich kann jetzt noch eine obere Schranke festgelegt werden, ab deren Wert sicher ein Kontextwechsel stattgefunden hat. In dieser Fallstudie wurde die Schranke auf 10μs gesetzt, das heißt in 0.04% der Fälle hat ein Kontextwechsel sicher stattgefunden.

4.6.5. Diskussion

Die Ergebnisse der Fallstudie zeigen die Anwendbarkeit der neuen Metrik innerhalb einer lebenszyklusabdeckenden Operationalisierung. Diese Operationalisierung wurde nach dieser Fallstudie beim Kooperationspartner aus der Avionikindustrie eingesetzt, um eine Softwareplattform, auf der Hardwaresimulationen aufgesetzt werden sollen, zu vermessen. Dabei wurden die zu testenden Prozesse während der Durchführung im Be-

nutzermodus ausgeführt und die Nettolaufzeit mit Hilfe einer Systemfunktion auf der Basis eines internen Zeitgebers ausgelesen, um so wenig invasiv wie möglich zu testen. Aus dem gleichen Grund wurden die Messdaten während den iterativen Messungen nicht persistent gespeichert, sondern vollständig im Datenpuffer vorgehalten. Das begrenzt die Dauer einer einzelnen Messung durch den zur Verfügung stehenden Arbeitsspeicher. Es kann aber auch erforderlich sein, dass die zu testenden Prozesse im Kernelmodus ausgeführt oder externe Zeitgeber verwendet werden müssen. In diesem Fall wird die Invasivität der Messung durch erweiterte Systemfunktionen erhöht, was die Messdaten und damit die Ergebnisse beeinflusst.

Abbildung 4.8 vergleicht zwei Ergebnisse der Fallstudie, in der die Priorität des zu testenden Prozesses bei sonst gleichem Testaufbau 2 manipuliert wurde. Abbildung 4.8(a) zeigt das Ergebnis, wenn die Priorität des zu testenden Prozesses höher, Abbildung 4.8(b) das Ergebnis, wenn die Priorität des zu testenden Prozesses niedriger ist als diejenige des Rechenlast-erzeugenden Prozesses. Es gibt nur einen signifikanten Unterschied in den Ergebnissen, nämlich das im Bereich von 4000μs eine Häufung von Zeitdifferenzen auftritt, wenn der zu testende Prozess eine höhere Priorität hat. Als Schedulingmechanimus wurde *Completely Fair Scheduler* eingesetzt.

4.6.6. Weitere Metriken

Während der Kooperationsprojekte wurden weitere Metriken zur Operationalisierung eingesetzt, um die zu entwickelnde Software oder die Softwareplattform auf zeitkritische Anforderungen hin zu untersuchen. In diesem Abschnitt werden diese Metriken zusammengefasst, um in Form eines Kataloges verwendet werden zu können.

Latenz

Kontext	Zeitkritische Kommunikation
Voraussetzung	Zeitgeber von Sender und Empfänger müssen synchron sein.
Bezug	A.36, A.6, A.7, A.4
Referenzen	[73]

Latenz bezeichnet die zeitliche Verzögerung zwischen Versenden und Empfang eines Datenpaketes. Dabei müssen die Systemzeiten jeweils beim Sender und Empfänger gemessen werden. Aus diesem Grund ist die Synchronizität der beiden Zeitgeber eine Voraussetzung, um die Metrik anwenden zu können. Die Differenz der beiden Werte ergibt die Latenz der Datenübertragung, mit dessen Hilfe das zeitliche Verhalten der Kommunikation beurteilt werden kann.

Bandbreite

Kontext	Zeitkritische Kommunikation
Voraussetzung	Sowohl die Datenübertragung als auch die Zeitgeber müssen ausreichend genau sein, die Zeitgeber außerdem synchron.
Bezug	A.36, A.6, A.7, A.4, Metrik Latenz
Referenzen	[73]

Bandbreite bezeichnet die maximal mögliche Datenmenge, die pro Zeiteinheit übertragen werden kann. Um eine Messung durchzuführen, werden Daten vom Sender zum Empfänger übertragen. Der Quotient aus der Datenmenge und der Latenz während dieser Datenübertragung entspricht der Bandbreite. Je größer die Datenmenge und die Dauer der Übertragung ist, desto repräsentativer wird die Bandbreite beurteilt.

Ausführungszeit

Kontext	Rechtzeitigkeit
Voraussetzung	Der Zeitgeber muss ausreichend genau sein.
Bezug	A.6, A.4, A.2
Referenzen	[74]

Ausführungszeit bezeichnet die Nettolaufzeit einzelner, zum Beispiel durch einen Schedulingmechanimus geplanter Prozesse. Eingabedaten sind dabei die Systemzeiten unmittelbar vor und nach der Ausführung des zu testenden Prozesses. In präemptiven Systemen werde alle Systemzeiten vor und nach einer iterativen Ausführung des Prozesses aufsummiert. Die Ausführungszeit entspricht der Differenz der beiden Systemzeiten. Für eine statistische Beurteilung werden möglichst viele Durchläufe und damit Messungen vorgenommen. Als Ausgaben werden berechnet:

- *Average-case*: Die durchschnittliche Zeit, in welcher der Prozess ausgeführt wurde.
- *Worst-case*: Längste Zeit, in welcher der Prozess ausgeführt wurde.
- *Best-case*: Kürzeste Zeit, in welcher der Prozess ausgeführt wurde.

Miss Percent

Kontext	Rechtzeitigkeit
Voraussetzung	Die Zeitfristen aller zu testenden Prozesse muss im Vorfeld konfiguriert sein.
Bezug	A.6, A.7, A.4, Metrik Mean Lateness
Referenzen	[26]

Miss Percent bezeichnet den prozentualen Anteil von Ausführungen eines zu testenden Prozesses, die eine feste Zeitfrist nicht eingehalten haben. Dabei werden für eine statistische Beurteilung möglichst viele Ausführungen eines zu testenden Prozesses gemessen. Die Metrik ist auch bei der Verwendung von präemptiven Schedulingmechanismen anwendbar. Nach der eigentlichen Messung werden die Differenzen aus geplanter Zeitfrist und der gemessenen Systemzeit bei Ausführungsende und danach der Quotient aus der Anzahl negativer und positiver Werte berechnet und ausgegeben.

Mean Lateness

Kontext	Rechtzeitigkeit
Voraussetzung	-
Bezug	A.6, A.7, A.4, Metrik Miss Percent
Referenzen	[26]

Wurde eine Ausführung des zu testenden Prozesses nicht fristgerecht beendet, wird die Differenz zwischen geplantem und tatsächlichem Ausführungsende berechnet und aufsummiert. Zusätzlich wird ein Zähler inkrementiert. Mean Lateness bezeichnet dann den Quotienten aus dieser Summe und dem Zähler und entspricht dem Durchschnitt der Zeitdifferenzen aller Ausführungen, die ihre Zeitfrist nicht eingehalten haben.

4.7. Testen von Software und Softwareplattformen

Die dritte und letzte Stufe des in der vorgelegten Arbeit beschriebenen Evaluierungsprozesses ist das Testen von Software und Softwareplattformen. Im Rahmen der Kooperationsprojekte wurden zum Testen Konformitätsmessungen und Operationalisierungen eingesetzt, wie für die erste und zweite Stufe des Evaluierungsprozesses beschrieben wurde. Natürlich wurden auch weitere Testmethoden eingesetzt wie zum Beispiel Modultests auf der Basis von Äquivalenzklassenbildung oder *Threshold*-Analyse. Wie in Kapitel 3 behauptet ist das Testen von Software für eingebettete Systeme wesentlicher aufwendiger als der allgemeine Softwaretest. Aus diesem Grund existieren bei den einzelnen Kooperationspartnern etablierte Testsysteme und Verfahren, die nicht ohne weiteres auf die Problemstellung anwendbar sind. In dem Kooperationsprojekt mit einem Unternehmen aus der Licht- und Signaltechnik-Industrie wurde deshalb eine generische Methode auf den vorhandenen Prozess aufgesetzt, die es ermöglicht auf die Problemstellung zugeschnittene Testfälle zu generieren, die in den vorhandenen Verfahren eingesetzt werden konnten. Ausgangspunkt sind dabei Anwendungsfälle und Szenarien, die als Spezifikation der zu entwickelnde Software im ersten Schritt der Softwareentwicklung spezifiziert

wurden. Dieser Ansatz beruht auf der Idee *lebendiger Artefakte*, die im Rahmen agiler Entwicklungsprozesse [54, 80] eingesetzt werden.

In Anhang B werden diejenigen Anwendungsfälle systematisch katalogisiert und in Bezug zu den Anforderungen aus Abschnitt 4.2 und Anhang A gesetzt, die für die Problemstellung der vorgelegten Arbeit relevant sind und im Rahmen der Kooperationsprojekte mit der Automobilindustrie und der Licht- und Signaltechnik spezifiziert wurden. Auf der Basis dieser Anwendungsfälle wurden in den Testphasen der Kooperationsprojekte Testfälle mit Hilfe der in Abschnitt 4.7.2 beschriebenen Methode generiert, indem aus abstrakten Anwendungsfällen auf den technischen Kontext der zu testenden Software oder Softwareplattform konkretisierte Testfälle hergeleitet werden. Diese Methode beruht auf den Ansätzen von Friske und Schlingloff [18] sowie Kamsties, Pohl, Reis und Reuys [33].

4.7.1. Testfallgenerierung

Die Vorgehensweise, um aus einem Anwendungsfall einen Testfall zu generieren, ist zweigeteilt. Im ersten Schritt erfolgt die Analyse des Anwendungsfalls:

- Bestimmung von Systemfunktionen
- Bestimmung von Systemreaktionen
- Ermitteln des Kontrollflusses
- Abbilden von Funktion und Reaktion auf den Kontrollfluss

Im zweiten Schritt erfolgt die Erstellung passender Testfälle:

- Erstellen einer Zwischenrepräsentation
- Erstellung der Testfälle

Ist ein Anwendungsfall in natürlicher Sprache formuliert, entspricht die Analyse einem manuellen Lesen und Abarbeiten des Artefakts. Erfolgt im Gegensatz dazu die Beschreibung der Anwendungsfälle formal, kann diese Vorgehensweise bis hin zu einer automatische Methode verbessert werden. Im ersten Schritt der Analyse werden nun die Systemfunktionen der zu testenden Software oder Softwareplattform identifiziert, die für eine Testdurchführung benötigt werden. Das sind diejenigen Schnittstellen, über die der Datenfluss oder Kontrollfluss durch den Testfall manipuliert werden soll, um eine spezifische Reaktion der Software oder Softwareplattform hervorzurufen. Im zweiten Schritt der

Analyse werden diejenigen Schnittstellen identifiziert, über die für den Testfall relevante Reaktionen der Software oder Softwareplattform getestet werden können. Im dritten Schritt der Analyse muss jetzt der durch den Anwendungsfall vorgegebenen Kontrollfluss nachvollzogen und formalisiert werden. Dabei ist eine systematische Beschreibung des Kontrollflusses oft bereits ein Teil des Anwendungsfalls. In diesem Schritt werden also einzelne Kontrollflusselemente wie Schleifen, Verzweigungen, sequenzielle Abfolgen oder bedingte Sprünge identifiziert und in eine Struktur formalisiert. Im letzten Schritt der Analyse werden nun die identifizierten Systemfunktionen und Reaktionen auf den Kontrollfluss abgebildet, indem jedem Schritt des Kontrollflusses die entsprechenden Systemfunktionen beziehungsweise Reaktionen zugeordnet werden.

Eine exemplarische Zuordnung ist in Abbildung 4.9 dargestellt, das gesamte Beispiel ist in *Von Use Cases zu Test Cases: Eine systematische Vorgehensweise* [18] beschrieben worden. Dabei wird jedem Schritt des Kontrollflusses die eigentliche Funktion und ein Typ zugeordnet, *F* für Systemfunktion, *R* für Reaktion und *I* für eine interne Funktion der Software oder Softwareplattform, die keine Systemfunktionen benötigten beziehungsweise Reaktionen zur Folge haben.

Schritt	Typ	Funktion
1	F	`Select_MessageSlot(slot)`
2	F	`Start_Recording()`
3	R	`Delete_MessageSlot(slot)`
4a	R	`Record_Message()`
4b	F	`Stop_Recording()`
4c	I	`Memory_Exhaust()`

Abbildung 4.9.: Beispiel einer konzeptuellen Darstellung eines Kontrollflusses

4.7.2. Anwendung

Die vorliegende Arbeit verwendet diesen Ansatz nun wie folgt: Im ersten Schritt der Testfallerstellung wird aus der konzeptuellen Darstellung des Kontrollflusses ein Aktivitätsdiagramm erstellt. Dabei wird jeder Schritt im Kontrollfluss durch eine Transition im Aktivitätsdiagramm dargestellt. Mit Hilfe dieses Aktivitätsdiagramms werden nun im letzten Schritt der Vorgehensweise die einzelnen Testfälle generiert. Dabei muss für jeden möglichen Pfad im Aktivitätsdiagramm ein Testfall erstellt werden, damit eine vollständige Pfadabdeckung erreicht wird. Die Parameter der Systemfunktionen müs-

sen so gewählt werden, dass *Boundary*-Tests berücksichtigt werden, also insbesondere Grenzen der Definitions- und Wertebereiche von Variablen. Das folgende Beispiel erzeugt aus dem Anwendungsfall *Interruptbehandlung* B.1.1 im Anhang B auf Seite 143 entsprechende Testfälle.

Zur Durchführung des Tests muss eine Interruptbehandlung angefordert werden, hier durch die Systemfunktion `RequestInterrupt()`. Als Reaktion auf diese Anforderung sollte das System eine Interruptbehandlung ausführen, danach wieder in der Ablaufplanung der Prozesse fortfahren und mindestens ein Prozess kann seine Zeitfrist aufgrund des Interrupts nicht mehr einhalten können. Aus dieser Analyse ergibt sich der konkrete Kontrollfluss für diesen Anwendungsfall:

1. `RequestInterrupt()`
2. `ExecuteInterrupt()`
3. `ContinueScheduling()`
4. `DeadlineError()`

Aus diesem Kontrollfluss ergibt sich direkt die konzeptionelle Darstellung aus Abbildung 4.10 sowie die Zwischenrepräsentation aus Abbildung 4.11.

Schritt	Typ	Funktion
1	F	`RequestInterrupt()`
2	R	`ExecuteInterrupt()`
3	I	`ContinueScheduling()`
3a	R	`DeadlineError()`

Abbildung 4.10.: Konzeptionelle Darstellung des Kontrollflusses

Auf Basis dieser Analyse kann nun ein neuer Testfall generiert werden, der die Reaktionszeit der Softwareplattform bezüglich externen Ereignisse überprüft. Bei der Durchführung des Testfalls müssen Zeitgeber, Schedulingmechanimus, Interruptbehandlung und der Prozessor des zu testenden System berücksichtigt werden. Der Testfall liegt in demselben Kontext wie die Anforderungen A.15 und A.20 sowie der Anwendungsfall. Während der Initialisierung des Testfalls wird eine *Interruptserviceroutine* bei der Softwareplattform angemeldet. Diese implementiert nur das aktive Warten von t_{wait} Zeiteinheiten, hier als Beispiel Sekunden. Dieses Zeitintervall ist das einzige Argument,

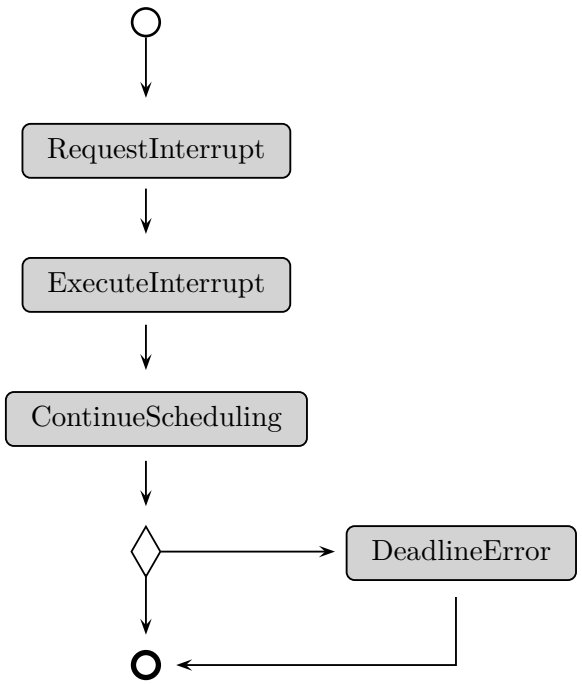


Abbildung 4.11.: Ablauf des Anwendungsfalls *InterruptBehandlung*

mit dem der Testfall arbeitet. Nach Ablauf dieser Zeit wird die Serviceroutine wieder verlassen. Während der eigentlichen Durchführung des Testfalls wird jetzt eine Interruptanforderung an das System gesendet. Als Reaktion auf diese Anforderung führt die Softwareplattform die Serviceroutine nun aus und das System muss t_{wait} Sekunden warten. Danach wird der zuvor unterbrochene Ablauf der Prozesse fortgeführt. Sollte nach diesem Interrupt einer der laufenden Prozesse seine Zeitfrist nicht mehr einhalten können, reagiert die Softwareplattform mit der Ausgabe einer entsprechenden Fehlermeldung. Um den Testfall durchführen zu können, muss das zu testende System ausgeführt werden und die Verarbeitung von Interrupts aktiviert sein. Das System befindet sich nach Ausführung des Testfalls entweder in einem Fehlerzustand oder wird ohne Fehlermeldung mit der Ablaufplanung der Prozesse fortfahren. Dieser Testfall kann nur auf Softwareplattformen mit Schedulingmechanimus angewendet werden. Externe Ereignisse müssen verarbeitet und das Verletzten einer Zeitfrist muss signalisiert werden können. Die Zeitgeber des Systems müssen mindestens so genau sein, wie die durch den Testablauf zu erwartenden Ergebnisse genau sein müssen.

5. Entwurfsmuster

Durch die Konkretisierung der Problemstellung in Kapitel 3 wurde der Bedarf an Entwurfsmustern [19] gezeigt, welche eine Refaktorierung des Verhaltens von Software oder Softwareplattformen zugeschnitten auf die beschriebene Problemstellung ermöglichen. Mit Hilfe dieser Entwurfsmuster können Software oder Softwareplattformen adaptiert oder in einer frühen Phase der Entwicklung in diese Richtung weiterentwickelt werden, wenn die in Kapitel 4 beschriebenen Evaluierungsmethoden gezeigt haben, dass zeitkritische Anforderungen an diese Software durch den aktuellen Stand der Entwicklung nicht erfüllt worden sind. Aus diesem Grund werden in den folgenden Abschnitten Entwurfsmuster vorgestellt, die im Rahmen der Kooperationsprojekte entwickelt und mindestens prototypisch eingesetzt werden konnten. Diese Entwurfsmuster können in dem in Abschnitt 3.4 vorgestellten iterativen Entwicklungsprozess der Evaluierung nachgelagert verwendet werden, um die zu verbessernde Software oder Softwareplattform entsprechend zu adaptieren und damit den Konformitätsgrad der Software gegenüber den propagierten zeitkritischen Anforderungen zu verbessern. Bei einer erneuten Evaluierung der Software ist der Konformitätsgrad so signifikant verbessert, dass diese Software innerhalb eines zeitkritischen Netzwerks eingesetzt werden kann, ohne die Hauptfunktion von Echtzeitknoten innerhalb dieses Netzwerkes zu erhöhen.

Zwei dieser Entwurfsmuster wurden hauptsächlich für Softwarearchitekturen, die Knoten-zu-Knoten Kommunikation realisieren, die anderen drei für den Einsatz innerhalb von Netzwerkknoten entwickelt. In Abschnitt 5.1 wird die Datenstruktur *Decoupling Datastructure* vorgestellt, die Schichten einer Softwarearchitektur voneinander entkoppelt und einen unidirektionalen echtzeitfähigen Datenzugriff erlaubt. Das Entwurfsmuster *Active Cross-Layer Balancing*, vorgestellt in Abschnitt 5.2, verlagert den Datenzugriff in die Schichten einer Softwarearchitektur und balanciert aktiv den Datendurchsatz zwischen diesen Schichten. Der Datenzugriff wird dabei asynchron verzögert. Abschnitt 5.3 beschreibt das Entwurfsmuster *Adaptive Controlflow Transitions*, das einen echtzeitfähigen Wechsel des Netzwerkbetriebs auf allen Netzwerkknoten ermöglicht. Insbesondere erfolgt die Freigabe von Ressourcen unmittelbar. Das Entwurfsmuster *Resume on Exception*, vorgestellt in Abschnitt 5.4, behandelt Ausnahmen in Netzwerkknoten unter Berücksichtigung des netzwerkweiten Betriebszustands. Die Ausnahme muss dabei die lokale Ausführung nicht mehr grundsätzlich abbrechen. Abschnitt 5.5 beschreibt das

Entwurfsmuster *Virtual Real-time*, das zeitliches Verhalten von Prozessen virtualisiert. Innerhalb der virtuellen Plattform können Programme echtzeitfähig ausgeführt werden. Die Beschreibung der einzelnen Entwurfsmuster besteht jeweils aus einer Motivation, der Vorstellung verwandter Arbeiten, der Darstellung einer Fallstudie beziehungsweise Diskussion sowie einem Fazit und Ausblick. Die Anforderungen in Abschnitt 4.2 und Anhang A sowie die Anwendungsfälle aus Abschnitt 4.7 und Anhang B werden in Tabelle 5.1 in Bezug auf die in diesem Kapitel vorgestellten Entwurfsmuster gesetzt.

Decoupling Datastructure	
Anforderungen	A.4, A.12, A.23, A.36
Anwendungsfälle	in B.1.1, B.1.2
Active Cross-Layer Balancing	
Anforderungen	A.29, A.36, A.37
Adaptive Controlflow Transitions	
Anforderungen	A.3, A.34, A.36
Anwendungsfälle	in B.1.2
Resume on Exception	
Anforderungen	A.3, A.18, A.30
Virtual Real-time	
Anforderungen	A.2, A.4, A.13, A.16, A.21, A.24
Anwendungsfälle	in B.1.1, B.1.2, B.1.3

Tabelle 5.1.: Übersicht Anwendungsfälle und Anforderungen

Die in diesem Kapitel vorgestellten Inhalte wurden in den Kooperationsprojekten mit Unternehmen aus der Licht- und Signaltechnik und der Avionikindustrie und in einigen Veröffentlichungen [20, 31, 39, 59, 62, 64, 76] erarbeitet beziehungsweise zusammengefasst. Das Entwurfsmuster *Resume on Exception* wurde von den Kooperationsprojekten unabhängig entwickelt und dann auf Basis des Kooperationsprojektes mit der Avionikindustrie evaluiert.

Die folgenden Abschnitte stellen die entwickelten Entwurfsmuster in einer einfachen Schablone 5.2 wie folgt vor. Eine noch feingranularere Struktur wurde nicht verwendet, da die Entwurfsmuster nicht verallgemeinert werden sollen.

Projektkontext	Einleitung des Abschnittes
Zielsetzung	Teil der Einleitung
Problemstellung	
Verwandte Arbeiten	
Konzept	Lösungsvorschlag, Musterbeschreibung
Evaluierung	Auswertung, ggf. Fallstudien, Fazit

Tabelle 5.2.: Schablone zur Beschreibung von Entwurfsmustern

5.1. Decoupling Datastructure

In diesem Abschnitt wird das Entwurfsmuster *Decoupling Datastructure* vorgestellt, das die Schichten einer Softwarearchitektur voneinander entkoppelt und einen unidirektionalen echtzeitfähigen Datenzugriff erlaubt. Dieses Entwurfsmuster wurde in gemeinschaftlicher Arbeit mit Dominik Schmithausen entwickelt und im Rahmen seiner Bachelorarbeit [59] veröffentlicht. Die Decoupling Datastructure wurde zuerst im Kooperationsprojekt mit der Licht- und Signaltechnikindustrie, insbesondere in dem Entwurfsmuster Active Cross-Layer Balancing verwendet, wurde aber auch in folgenden Projekten und Evaluierung häufig als Datenstruktur zum Einsatz in den Entwurfsmustern gewählt. Ausgangspunkt bei der Entwicklung dieses Entwurfsmusters ist der Bedarf an einer auf die Problemstellung zugeschnittenen Datenstruktur, die es Echtzeitsystemen erlaubt, ohne warten zu müssen, Daten von anderen nicht-echtzeitfähigen Systemen zu empfangen oder zu senden. Das ist der erste Schritt zu einer echtzeitfähigen Kommunikation, während der zweite Schritt eine faktisch verlustfreie und rechtzeitige Datenübertragung gewährleisten wird, wie zum Beispiel durch das Entwurfsmuster Active Cross-Layer Balancing. Ein mögliches Szenario ist der Tempomat in Abbildung 3.3(a), die Problemstellung wurde im Detail in Kapitel 3 beschrieben. Eine Implementierung der Decoupling Datastructure könnte zum Beispiel innerhalb der Kommunikation zwischen Tempomat und der Geschwindigkeitsanzeige eingesetzt werden.

5.1.1. Zielsetzung

Ziel bei der Entwicklung der hier vorgestellten Datenstruktur war also der Entwurf einer potentiell echtzeitfähigen Struktur, die robust gegen Datenverluste ist und auf einer existierenden Softwareplattform eingesetzt werden kann. Dazu wurden wie in Abschnitt 5.1.2 blockierende und nicht-blockierende Algorithmen besprochen und dann in Abschnitt 5.1.3 die verwandten Arbeiten bezüglich Verlust und block-freier Datenstrukturen betrachtet. Darauf aufbauend wird in Abschnitt 5.1.4 ein Konzept aus diesen

Ansätzen entwickelt. Danach wird in Abschnitt 5.1.5 eine Fallstudie ausgeführt. Es sollte dabei möglich sein, die Decoupling Datastructure in allen Kommunikationsszenarien aus Abschnitt 2.4.3 einzusetzen.

5.1.2. Problemstellung

Findet auf einer Softwareplattform Interprozesskommunikation statt, müssen die Zugriffe auf gemeinsame Ressourcen synchronisiert werden. Eine traditionelle Vorgehensweise ist die *Mutual Exclusion*. Durch das Prinzip des wechselseitigen Ausschlusses wird ausgeschlossen, dass verschiedene Prozesse zur gleichen Zeit auf eine gemeinsame Ressource zugreifen können. Aus der Sicht einzelner Prozesse kann immer nur ein Prozess den kritischen Bereich betreten und auch nur dann, wenn noch kein anderer Prozess diese Ressource beansprucht hat. Der Zugriffsschutz wird dabei durch Sperrmechanismen wie Mutexe oder Semaphoren realisiert [82].

Blockierende Algorithmen

Setzt eine Softwareplattform Mutexe oder Semaphoren ein, um kritische Bereiche durch die Prozesse gemeinsam verwendeten Ressourcen zu schützen, blockieren diejenigen Prozesse, die auf eine Ressource zugreifen müssen, dies aber gerade nicht können. Diese Algorithmen sind zur Realisierung der Kommunikation zwischen einzelnen Prozessen prinzipiell ungeeignet [48]. Liest zum Beispiel ein Prozess gerade eine gemeinsame Speicherstelle aus, dann muss ein Prozess blockieren, der gerade an dieser Speicherstelle schreiben möchte. In diesem Fall kann der schreibende Prozess keine Echtzeitfähigkeit garantieren.

Implementiert eine Softwareplattform den wechselseitigen Ausschluss, können *Deadlocks* aber auch *Livelocks* durch die Verwendung gemeinsamer Ressourcen auftreten. Als Deadlock bezeichnet man den Zustand, wenn mindestens zwei Prozesse jeweils eine Ressource sperren, die von dem anderen Prozess benötigt wird. Dadurch blockieren beide Prozesse permanent. Als Livelock bezeichnet man den Zustand, wenn sich zwei Prozesse permanent beim Zugriff auf eine gemeinsame Ressource abwechseln, ohne dabei ihre Funktion zu erfüllen [77].

Eine weitere Problemstellung, die durch den wechselseitigen Ausschluss vor allem in Echtzeitsystemen und Plattformen gelöst werden muss, ist eine potentielle *Priority-Inversion* in Abhängigkeit der verwendeten Schedulingmechanismen [77, 82]. Eine Inversion der Prioritäten tritt immer dann auf, wenn auf einer Softwareplattform drei Prozesse geplant werden und mindestens zwei dieser Prozesse auf dieselbe Ressource zugreifen müssen. Zur Laufzeit betritt ein Prozess einen kritischen Bereich und greift auf

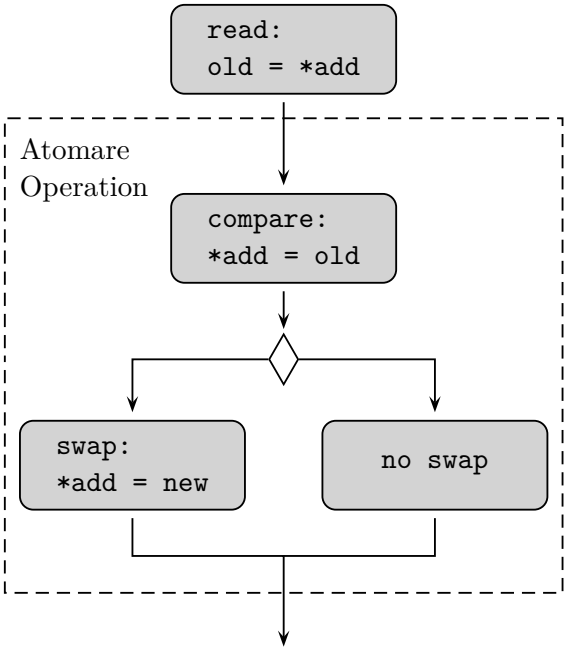
die gemeinsame Ressource zu. Ein höher priorisierter Prozess unterbricht diesen Prozess während dem Zugriff und blockiert dann, da die Ressource durch wechselseitigen Ausschluss gesperrt ist. Ein dritter Prozess, dessen Priorität zwischen den Prioritäten der konkurrierenden Prozess liegt, wird nun geplant und ausgeführt. Durch diese Inversion der Prioritäten zwischen dem höchst-priorisierten und dritten Prozess ist es möglich, dass der dritte Prozess seine Funktion eher abschließen kann als der höchst-priorisierte Prozess. Das Laufzeitverhalten der Softwareplattform ist dadurch nicht-deterministisch.

Nicht-blockierende Algorithmen

Im Gegensatz zu blockierenden Algorithmen kann Datenkommunikation auch nicht-blockierend stattfinden. Die eingesetzten Algorithmen basieren zumeist auf atomaren Operationen wie *Compare-And-Swap*, *Test-And-Add* oder *Fetch-And-Add*, welche die Softwareplattform beziehungsweise im Normalfall die Hardwareplattform zur Verfügung stellen muss. Eine atomare Operation bezeichnet dabei eine nicht durch die Softwareplattform unterbrechbare Abarbeitung von Einzeloperationen [43], die als logische Einheit betrachtet werden können.

Abbildung 5.1 veranschaulicht das Prinzip der Compare and Swap Operation, die keine Sperrmechanismen verwendet und deswegen keine Deadlocks, Livelocks oder Priority Inversion verursachen kann [48, 77]. Abbildung 5.1(a) zeigt den Kontrollfluss dieser Operation. Zuerst wird der Wert einer spezifischen Speicherstelle ausgelesen. Dann wird in einem atomaren Schritt der ausgelese Wert noch einmal mit dem aktuellen Wert der Speicherstelle verglichen. Sind die beiden Werte konsistent zueinander, wird die Speicherstelle mit einem neuen Wert beschrieben. Sind der ausgelesene und aktuelle Wert an der Speicherstelle nicht gleich, dann wird die Operation abgebrochen. Die Parameter der Operation in Abbildung 5.1(b) sind die Speicheradresse, auf die geschrieben werden soll, der alte Wert an der Speicherstelle vor der Operation, der bereits ausgelesen wurde, und der neue Wert nach der Operation. Die Operation quittiert den Vorgang entsprechend dem Ergebnis positiv oder negativ.

Bei der Compare-and-Swap-Methode kann potentiell das sogenannte *ABA*-Problem auftreten [10]. Während der Durchführung dieser Operation könnte der aktuelle Wert der betroffenen Speicherstelle mindestens zweimal verändert werden, wobei die letzte Veränderung wieder dem Wert entspricht, der während des ersten Schrittes des Algorithmus zum Vergleich ausgelesen wurde. Haben maximal zwei Prozesse Zugriff auf die entsprechende Speicherstelle, kann dieses dedizierte Problem nicht auftreten [10, 48]. Wird das Compare and Swap eines Prozesses auf diese Art durch den zweiten Prozess gestört, ist die Speicherstelle für den Algorithmus zu Beginn der atomaren Operation als defekt identifizierbar.



(a) Kontrollfluss

```
« atomic operation »
cas(int *add, int old, int new)
{
  if(*add == old)
  {
    *add := new;
    return true;
  }
  else
  {
    return false;
  }
}
```

(b) Pseudocode

Abbildung 5.1.: *Compare and Swap*

5.1.3. Verwandte Arbeiten

Eine mögliche Datenstruktur, die ohne Zugriffssperren arbeitet, wurde von A. Gottlieb, B. D. Lubachevsky und L. Rudolph [24] sowie J. M. Mellor- Crummey [46] vorgestellt. Allerdings blockiert der verwendete Algorithmus und einzelne Prozesse können beliebig lange unterbrochen werden, weswegen dieser Ansatz nicht für die in dieser Arbeit vorgestellte Problemstellung verwendet werden konnte. In einer Veröffentlichung von M. M. Michael und M. L. Scott [48] wird ein Algorithmus von R. K. Treiber evaluiert, der nicht-blockierend arbeitet. Die Leseoperation arbeitet allerdings sehr ineffizient, da die Komplexität überproportional zur Anzahl der verwalteten Elemente innerhalb der Datenstruktur ist.

Ein linearisierbarer und nicht-blockierender Algorithmus von S. Prakash, Y. H. Lee und T. Johnson setzt eine verlinkte Liste mit entsprechenden *Head* und *Tail* Zeigern ein [48]. Für eine Lese- oder Schreiboperation erstellt der beschriebene Algorithmus ein Abbild der gesamten Datenstruktur. Die Komplexität kann dann zwar deterministisch abgeschätzt werden, allerdings ist die Vorgehensweise oft nicht effizient genug, um in

Software zur Kommunikation mit Echtzeitplattformen eingesetzt zu werden. Weitergeführt entwickelte J. M. Stone einen Sperrmechanismus-freien Algorithmus, der einen verlinkten Ringpuffer einsetzt und nur einen Zeiger auf die Datenstruktur benötigt [47].

Ein anderer nicht-blockierenden Algorithmus [78, 79] , vorgestellt von J. D. Valois, der ebenfalls auf einer verlinkten Liste basiert, setzt *Dummy* Elemente ein, anstatt einzelne Abbilder der Datenstruktur zu erstellen. Dieses Element erhöht zwar die Konsistenz der Liste, allerdings ist es durch dieses Hilfselement möglich, dass der *Tail* vor dem *Head*-Zeiger auf der Liste steht. Dieses spezielle Problem erschwert eine konkrete Implementierung des Algorithmus. Aus diesem Grund wurde der Algorithmus noch einmal angepasst [47, 48], indem während der Leseoperation auch eine Korrektur des *Tail*-Zeigers vorgesehen ist.

5.1.4. Konzept

In diesem Abschnitt wird ein Ansatz vorgestellt, wie auf der Basis eines aus der Literatur entnommenen nicht-blockierenden Algorithmus [47, 48] von H. Massalin und C. Pu eine Datenstruktur aufgesetzt werden kann, die alle in Abschnitt 4.2 vorgestellten Anforderungen an eine echtzeitfähige Kommunikation erfüllt. Dieser Ansatz beruht auf der Verwendung eines Arrays und ist so konzipiert, dass mit einer einzelnen Instanz dieser Datenstruktur zwei Prozesse unidirektional Daten übertragen können. Dabei ist genau einer der Prozesse echtzeitfähig gemäß der Problemstellung, beschrieben in Kapitel 3. In den folgenden Paragraphen werden zuerst die Anforderungen an die neue Datenstruktur beziehungsweise an die Algorithmen zusammengefasst, die zur echtzeitfähigen Synchronisierung verwendet werden können. Da die Decoupling Datastructure in verschiedensten Szenarien eingesetzt wurde, berücksichtigt das vorgestellte Konzept die Kommunikation zwischen Threads, Prozesses und Applikationen. Unter der Einschränkung, dass eine Instanz der Decoupling Datastructure dediziert zwei einzelnen der genannten Kommunikationspartner zur Verfügung steht, werden keine *Multi-Core* Hardwareplattformen berücksichtigt.

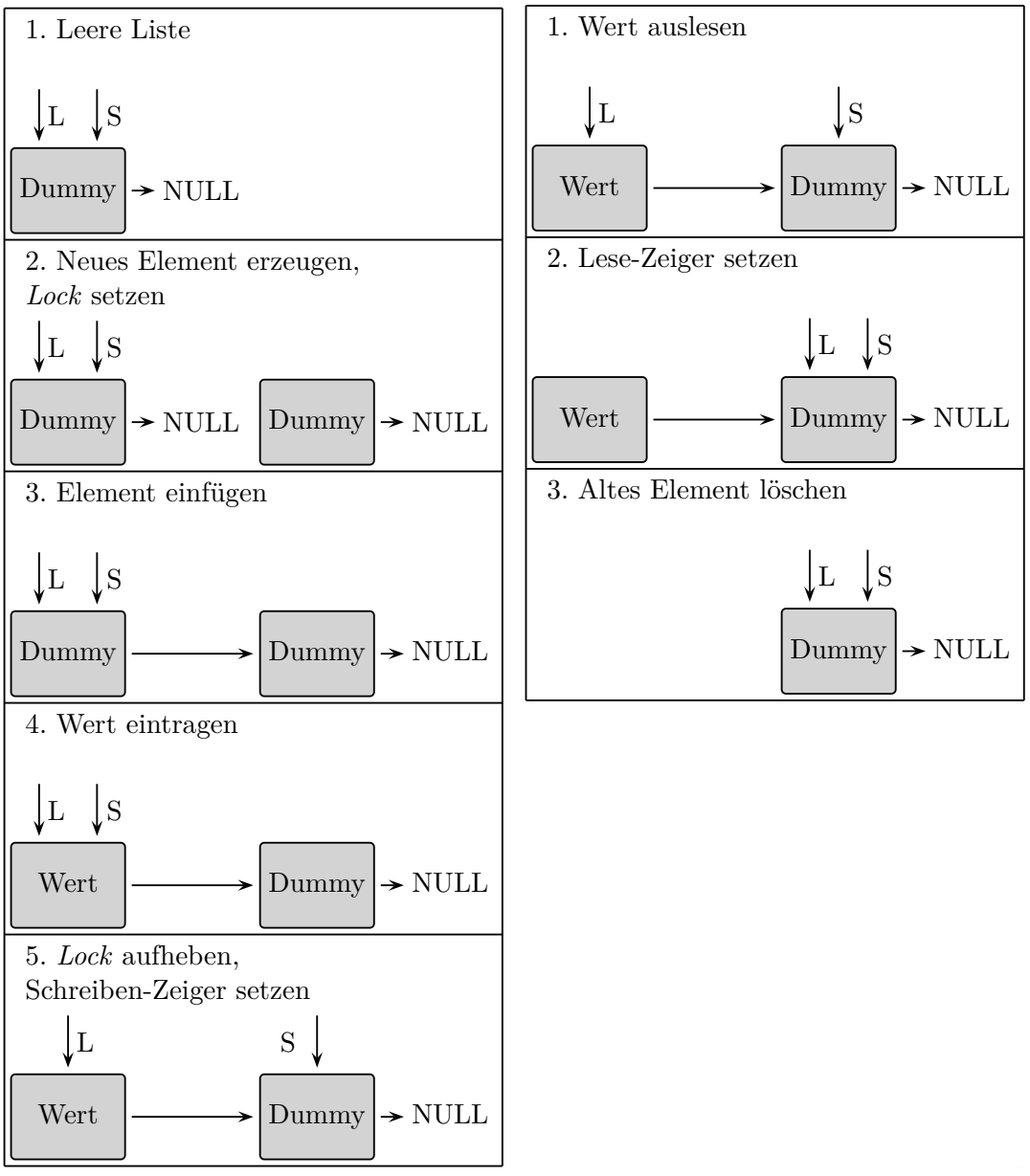
Im folgenden wird ein Algorithmus basierend auf *Compare and Swap* und ein Algorithmus mit traditionellen *Two-locks*-Mechanismus [47] vorgestellt, um die beschriebene Problemstellung zu lösen. Beide Algorithmen setzen je einen Zeiger als *Head* und *Tail* ein. Für die Beschreibung des neuen Entwurfsmusters müssen keine konkreten Datenstrukturen eingesetzt werden. Für die Evaluierung und den Einsatz im Rahmen des Kooperationsprojektes wurden je ein Ringpuffer und eine Warteschlange implementiert. Da die Ordnung der Elemente innerhalb der Datenstruktur erhalten bleiben sollte, wurden keine Kellerspeicher eingesetzt, auch wenn diese durch die oben genannten verwandten Arbeiten mit abgedeckt werden. Zusätzlich zu diesen beiden Zeigern wird ein Platzhalter

eingesetzt, ein sogenanntes *Dummy*-Element. Dadurch wird der Sonderfall vermieden, ein Element in eine leere Liste einfügen zu müssen.

Two-Locks-Algorithmus

Abbildung 5.2 veranschaulicht das Konzept einer Decoupling Datastructure auf der Basis eines modifizierten Two-locks-Algorithmus. In der Abbildung steht L für einen Zeiger auf das nächste zu lesende Element der Decoupling Datastructure und S für einen Zeiger auf das nächste zu schreibende Element. Die ursprüngliche Version von M. M. Michael und M. L. Scott [47] setzt Sperrmechanismen ein, um den Zugriff auf kritische Bereiche der Datenstruktur zu sperren. In der ursprünglichen Version wird dabei das zu lesende oder zu schreibende Element direkt gesperrt. Deswegen muss diese Version bezüglich der Problemstellung modifiziert werden. Dazu wird zum Einfügen ein zusätzlicher Dummy verwendet, zum Beispiel eine Mutexversion, und für den schreibenden Prozess eine permanente Sperre auf das letzte Element der Datenstruktur gesetzt, auf eben den Dummy. Ein potentiell schreibender Zugriff kann also zu jedem Zeitpunkt auf den Dummy zugreifen und entsprechend der Einfügevorschrift ein Element in die Datenstruktur einfügen. Da auch ein potentiell lesender Prozess nicht-blockierend überprüfen kann, ob die Datenstruktur leer beziehungsweise nur der einzelne Dummy in der Datenstruktur vorhanden ist, erfüllt diese Version die in Kapitel 3 und Abschnitt 5.1.2 zusammengefassten Anforderungen an eine echtzeitfähige Datenstruktur.

Der Schreibvorgang in Abbildung 5.2(a) zeigt, dass die beiden auf die Datenstruktur zugreifenden Prozesse die Sperren für die Elemente setzen können. Nach Initialisierung der leeren Liste existiert ein einzelner Dummy mit einer Schreibsperre, beide Zeiger stehen auf diesem Dummy. Um ein Element in die Datenstruktur einzufügen, erzeugt der schreibende Prozess nun einen weiteren Dummy, der nach Erzeugung ebenfalls durch diesen Prozess gesperrt ist, und fügt das Element hinter dem bereits existierenden Dummy ein. Der eigentlich einzufügende Wert wird nun in den ersten Dummy eingetragen und der schreibende Prozess gibt dieses jetzt vollwertige Element frei. Zuletzt wird noch der Zeiger des schreibenden Prozesses auf den neuen Dummy gesetzt. Abbildung 5.2(b) zeigt den Lesevorgang, der das Element aus der Datenstruktur entnimmt. Der lesende Prozess prüft nicht-blockierend, ob die Datenstruktur leer ist. Das ist aus Sicht des lesenden Prozesses auch dann der Fall, wenn der schreibende Prozess gerade ein Element einfügt. Kann ein Element gelesen werden, wird der Zeiger des lesenden Prozesses ein Element weitergesetzt, gegebenenfalls auf den Dummy der Datenstruktur, und das zu lesende Element aus der Datenstruktur entnommen.



(a) Schreiben

(b) Lesen (Entnehmen)

Abbildung 5.2.: *Two-locks*-Algorithmus

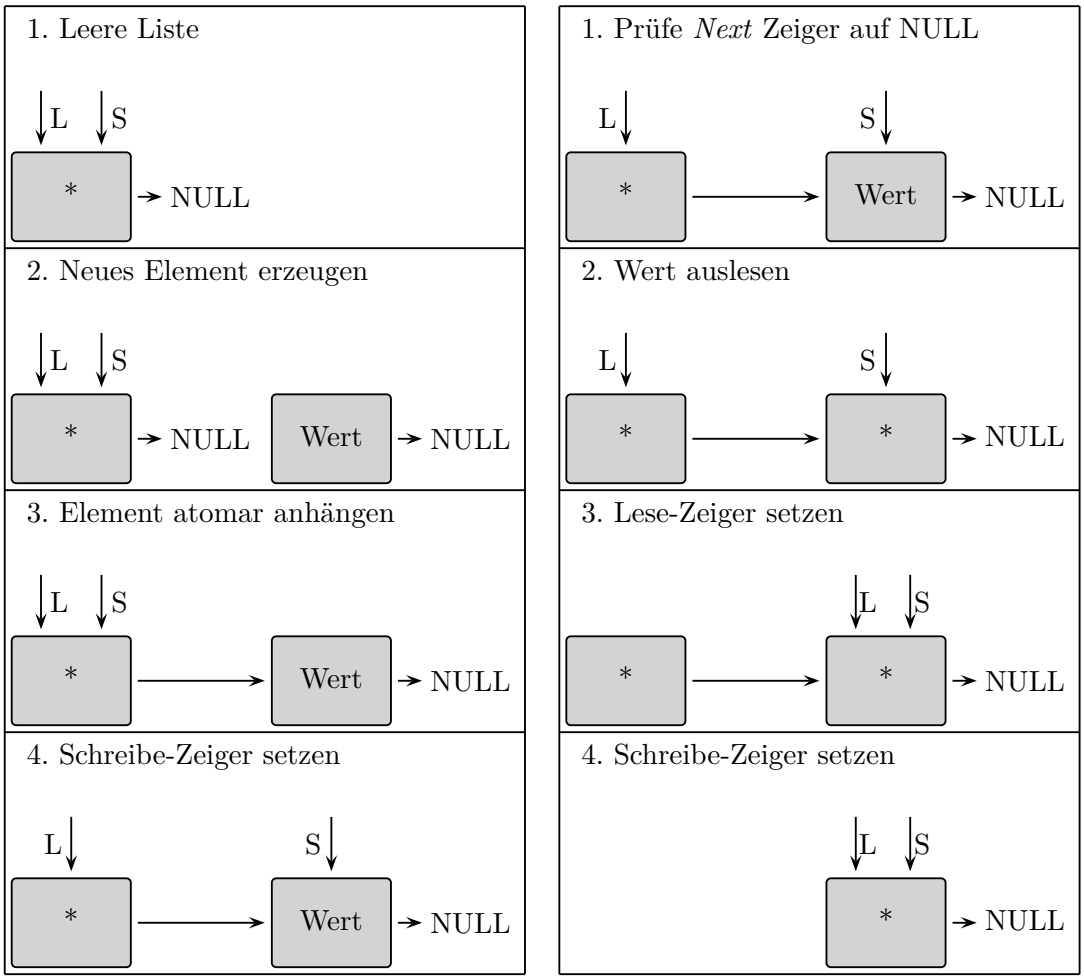
Compare-And-Swap-Algorithmus

M. M. Michael und M. L. Scott stellen in ihrem Papier [47] ebenfalls einen Compare-and-Swap-Algorithmus vor, der atomare Operationen anstatt eines Sperrmechanismus einsetzt. In diesem Algorithmus greift der lesende Prozess auf den Zeiger des schreibenden Prozesse zu, um die Ordnung des Lese- und Schreib-Zeigers zu überprüfen. Wird der schreibende Prozess durch den lesenden Prozess überholt, enthält die Datenstruktur keine Elemente. Mit Hilfe einer entsprechenden Modifikation kann jedoch auch dieser Algorithmus eingesetzt werden, um bezüglich der Problemstellung eine Datenstruktur zur Verfügung zu stellen. Abbildung 5.3 zeigt den Schreib- und den Lesevorgang des modifizierten Algorithmus.

Dieser modifizierte Algorithmus setzt einen Dummy zum Entnehmen von Elementen aus der Datenstruktur ein. Der Schreibvorgang, in Abbildung 5.3(a) dargestellt, erzeugt wieder als erstes ein neues und vollständiges Element. Dann wird dieses Element mit Hilfe der atomaren Operation wie in Abbildung 5.1 skizziert in die Liste eingefügt. Danach wird der Schreibe-Zeiger auf das neue Element gesetzt und der Vorgang ist abgeschlossen. Der Lesevorgang in Abbildung 5.3(b) prüft, ob die Datenstruktur leer ist. Dazu wird getestet, ob das durch den Lese-Zeiger referenzierte Element ein Nachfolgeelement hat. Ist der Nachfolger *NULL*, dann ist kein Element in der Datenstruktur vorhanden. Existiert das Nachfolgeelement, wird der Zeiger des lesenden Prozesses auf diesen Nachfolger gesetzt und dessen Vorgänger der Datenstruktur entnommen. Während dem Schreibvorgang wird der Zeiger des schreibenden Prozesses auf das neue Element gesetzt. Dieser Vorgang ist nicht mehr Teil der atomaren Operation. Aus diesem Grund kann die Überprüfung auf eine leere Datenstruktur potentiell ein *false positive* liefern, obwohl ein Element in die Datenstruktur bereits eingefügt werden. Dieses Problem kann durch ein Implementierungsdetail gelöst werden, in dem der interne Zeiger des schreibenden Prozesses als Hilfszeiger mit ausgewertet wird, wenn die Datenstruktur auf Elemente geprüft wird.

5.1.5. Evaluierung

Für das neue Entwurfsmuster Decoupling Datastructure ist keine konkrete Datenstruktur festgelegt worden, sondern wie Elemente aus einer generischen Struktur eingefügt und entnommen werden. Dabei überprüft eine potentielle Implementierung als Ringpuffer im Gegensatz zur Warteschlange nicht auf den Nachfolger des Lese-Zeigers, sondern direkt den Eintrag an der Speicherstelle, auf den dieser Zeiger steht. Das entspricht der üblichen Technik eines Ringpuffers, auf der Basis einer bestimmten Puffergröße den benötigten Arbeitsspeicher bereits vollständig vor dem Beginn einer Datenübertragung zu allozieren



(a) Schreiben (b) Lesen (Entnehmen)

Abbildung 5.3.: Compare-And-Swap-Algorithmus

und zu initiieren. Dementsprechend alloziert der schreibende Prozess keine Speicherbereiche und muss keine Zeiger auf Nachfolger erzeugen. Der Two-locks-Algorithmus kann also im Rahmen einer konkreten Ringpuffer-Implementierung nicht effizient eingesetzt werden. Damit muss im Fall eines Ringpuffers zwingend der Compare and Swap Algorithmus eingesetzt werden. Die Decoupling Datastructure setzt Platzhalter als initiale Werte innerhalb der generischen Datenstruktur ein. Da der für einen Ringpuffer reservierte Arbeitsspeicher während der ganzen Laufzeit alloziert bleibt, muss mindestens ein solcher Dummy nach jeder Leseoperation wieder durch den lesenden Prozess in den Ringpuffer eingefügt werden. Sonst kann durch den vorgestellten Algorithmus nicht sichergestellt werden, dass es zur Laufzeit keinen nicht-bemerkten Pufferüberlauf gibt.

Vergleich der beiden Algorithmen

Der wesentliche Unterschied der beiden Algorithmen ist, dass der Two-locks-Algorithmus eine reine Softwarelösung ist und damit plattformunabhängig. Diese Lösung kann POSIX[1] konform implementiert werden. Der auf Compare and Swap basierende Algorithmus setzt dagegen eine in Hardware realisierte atomare Operation ein.

Ein weiterer Unterschied der Algorithmen sind die verwendeten Zugriffe. In der Compare-and-Swap-Variante hat sowohl der lesende als auch der schreibende Prozess permanenten Zugriff auf die Datenstruktur. In der Two-locks Variante wird der Dummy durch den schreibenden Prozess permanent blockiert. Dennoch arbeiten beide Algorithmen nicht-blockierend. Der Compare and Swap Algorithmus benötigt weniger Speicherplatz als der Two-locks-Algorithmus. Im Umkehrschluss überprüft dieser Algorithmus auf der entsprechenden Datenstruktur die Konsistenz der Speicherstelle mehrfach.

Technische Beurteilung

Dominik Schmithausen [59] betrachtet in seiner Bachelorarbeit die Decoupling Datastructure Algorithmen auf der Basis konkreter Datenstrukturen. Diese Datenstrukturen sind die verlinkte Warteschlange, der verlinkte Ringpuffer sowie ein Array-basierter Ringpuffer. Die technische Evaluierung im Rahmen dieser Arbeit basiert auf Implementierungen der berücksichtigten Datenstrukturen. Die Evaluierung vergleicht die Decoupling Datastructure nicht in Abhängigkeit einer prozessualen beziehungsweise intra-prozessualen Datenübertragung, da die Allokation von gemeinsamen Speicher zwischen einzelnen Prozessen wesentlich aufwendiger ist als für privaten Arbeitsspeicher. In der hier vorgelegten Arbeit wird exemplarisch der Benchmark vorgestellt, wie er in der Arbeit von Dominik

[1] http://pubs.opengroup.org/onlinepubs/9699919799/

Schmithausen ausführlich und zusammen mit den weiteren Testszenarien beschrieben wird.

Die Beurteilung der Zugriffsalgorithmen wurde auf einem handelsüblichen Rechnersystem mit 2 GB Arbeitsspeicher mit Ubuntu-Realtime ausgeführt. Dazu wurde das Ubuntu Release 9.10 (karmic) auf dem Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66Ghz installiert. Danach wurde der Linux-Kernel durch den Realtime Patch 2.6.31-rt21[2] aktualisiert. Nach diesem Patch stellt das Betriebssystem eine POSIX konforme Schnittstelle bereit, um Prozesse in Echtzeit auszuführen. Für die Ausführung atomarer Operationen wurde die Bibliothek *libatomics-ops*[3] installiert, die auch eine Compare and Swap Operation zur Verfügung stellt.

In folgendem Benchmark werden die Schreiboperationen der Algorithmen für alle Datenstrukturen ausgemessen. Diese Werte helfen die Performanz der Algorithmen in Abhängigkeit der jeweils implementierten Datenstruktur zu vergleichen. Um den Testablauf zu vereinfachen, wurden die jeweils konstanten Parameter des Testaufbaus durch Platzhalter ersetzt, deren konkrete Werte für die einzelnen Tests in einer entsprechenden Textdatei vorgehalten wurden. Diese Platzhalter sind zum Beispiel *Empty*, ein Rückgabewert für lesende Prozesse, und *NIL*. Dieser Rückgabewert definiert einen nicht verfügbaren Wert und wird auch als Dummy eingesetzt.

Beide Algorithmen wurden jeweils mit allen drei Datenstrukturen implementiert und der Benchmark auf diesen Implementierungen ausgeführt. Ein einzelner Schreibvorgang, der hier gemessen werden soll, ist sehr schnell, die Genauigkeit der Messung ist deswegen nicht sehr genau. Aus diesem Grund werden immer die Zeitintervalle für eine konstante Anzahl an Schreibvorgängen als Block gemessen und dann operationalisiert. Abbildung 5.4 zeigt die Blockbildung bei der Durchführung eines Benchmark. Ein solcher Schreibblock ist in der Abbildung mit drei einzelnen Schreibvorgänge dargestellt, während der tatsächlichen Durchführung wurden 100000 solcher Vorgänge durchgeführt. Die Zeit T_S wird dabei für jeden Benchmark 50 mal gemessen, wobei immer 40 Blöcke geschrieben werden, und dann die mittleren, minimale und maximale Durchführungsdauer für einen Schreibblock ermittelt. Alle Ergebnisse werden in einer Textdatei persistent gespeichert. Für die Evaluierung einer prozessualen Datenübertragung können die Testparameter auf 20 Blöcke zu je 10000 Schreibvorgängen reduziert werden, ohne die statistischen Ergebnisse zu beeinflussen.

Das Ergebnis der Messungen war eine nahezu konstante Zeit für einen Schreibvorgang, sowohl für die Durchschnittszeit als auch den Worst-Case. Dieses deterministische Verhalten ermöglicht in einem nachgelagerten Echtzeitszenario, das ebenfalls in der

[2] http://www.osadl.org
[3] http://packages.debian.org/de/sid/libatomic-ops-dev

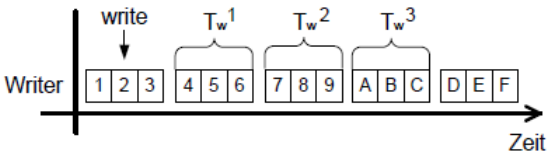


Abbildung 5.4.: Ablauf des Benchmark

Bachelorarbeit von Dominik Schmithausen [59] evaluiert wird, die erfolgreiche Datenübertragung eines echtzeitfähigen, schreibenden Prozess zu einem nicht-echtzeitfähigen, lesenden Prozess. Außerdem arbeitet die Ringpuffervariante ab dem zweiten Schreibvorgang wie erwartet schneller als die Warteschlange, da dann die Allokationsfunktion den Speicherbereich des Ringpuffers in einer Tabelle vorhält, die während dem ersten Schreibvorgang angelegt wurde. Die Evaluierung zeigt auch, dass der Compare and Swap tatsächlich ein besseres Zeitverhalten hat als der Two-locks-Algorithmus. Weiterhin ist die Ringpuffervariante schneller als Algorithmen, die auf einer Warteschlange basieren.

Im Rahmen der Evaluierung der prozessualen Datenübertragung wurden zwei Besonderheiten identifiziert. Erstens war die Zeit für einen Schreibvorgang bei der Compare and Swap Variante auf der Basis eines Ringpuffers (Array) im Gegensatz zu den anderen Kombinationen aus Algorithmen und Datenstrukturen nicht-deterministisch. Zweitens sind die Ergebnisse für den Compare and Swap Algorithmus auf Basis der Warteschlange schlechter als die Ergebnisse der Two-locks Variante.

5.1.6. Fazit

Die Decoupling Datastructure definiert zwei Algorithmen über einer generischen Datenstruktur, um eine nicht-blockierende Datenübertragung inklusive einer oberer Schranke für das zeitliche Verhalten der Schreibvorgängen in die entsprechende Datenstruktur zu ermöglichen. Dieses Entwurfsmuster kann zwischen Prozesses, Subprozessen oder Applikationen eingesetzt werden. Die Ergebnisse zeigen allgemein, dass die Compare and Swap Varianten zeiteffizienter sind als die Two-locks Varianten. Kann hardwareabhängig ein solcher Algorithmus eingesetzt werden, ist diese Variante besser geeignet. Findet die Datenübertragung über mehrere Plattformen verteilt statt oder die Hardware keine passende atomaren Operationen zur Verfügung stellt, muss die Two-locks Variante verwendet werden. Diese Implementierung ist POSIX-konform. Ist die Größe der Datenstruktur vorhersagbar, kann der effizientere Ringpuffer eingesetzt werden. Ansonsten kann die Warteschlange mit flexibler Größe eingesetzt werden. Eine Reinitialisierung des Ringpuffers ist dagegen während zeitkritischer Laufzeiten der Applikation nicht möglich, da der Zeitpunkt der Reinitialisierung nur schlecht vorhergesagt werden kann.

5.2. Active Cross-Layer Balancing

In diesem Abschnitt wird das Verhaltensmuster *Active Cross-Layer Balancing* vorgestellt, das den Zugriff bei der Datenübertragung in die Schichten einer Softwarearchitektur verlagert und aktiv den Datendurchsatz zwischen diesen Schichten balanciert. Der Datenzugriff erfolgt dabei im ersten Schritt durch ein synchrones Signal über die Schichtgrenze hinweg und im zweiten Schritt asynchron durch eine Verarbeitung innerhalb der über das Signal informierten Schicht. Dieses Verhaltensmuster wurde in gemeinschaftlicher Arbeit mit Tim Lange entwickelt und im Rahmen seiner Bachelorarbeit [39] veröffentlicht. Die Active Cross-Layer Balancing wurde zuerst im Kooperationsprojekt mit einem Unternehmen aus der Licht- und Signaltechnikindustrie entwickelt und erfolgreich eingesetzt. Im Anschluss wurde zu diesem Verhaltensmuster ein Papier [64] veröffentlicht.

In Abschnitt 5.1 wurde eine auf die Problemstellung aus Kapitel 3 zugeschnittene Datenstruktur vorgestellt. In diesem Abschnitt wird nun eine verlustfreie Datenübertragung und rechtzeitige Datenübertragung vorgestellt, in der diese Datenstruktur zwischen einzelnen Softwareschichten zur Datenübertragung eingesetzt wird. Diese neue Datenübertragung wurde in dem Kooperationsprojekt mit einem Unternehmen aus der Licht- und Signaltechnikindustrie entwickelt, in dem eine Softwaresimulation realisiert wurde, um Stellwerke intensiv gegen eine entsprechend simuliertes Signalnetzes zu prüfen.

Die Stellwerke steuern zum Beispiel in Bahnhöfen hunderte von vernetzten eingebetteten Systemen. Empfängt ein Signalelement-Controller als Teil eines solchen Netzwerks ein Datenpaket, muss zuerst überprüft werden, ob der Controller der Adressat oder einer der Adressaten ist. Diese Funktion kann oftmals schon auf der Netzwerkebene des Controllers durchgeführt werden. Soll das Datenpaket durch den Controller verarbeitet werden, wird das Paket an die unterste Schicht der Softwarearchitektur weitergegeben. Von dort aus wird nun auf verschiedenen Schichten entschlüsselt, gegebenenfalls ein Plausibilitätstest durchgeführt, die Dateninformation ausgewertet und eine Empfangsbestätigung zum Beispiel aufgrund von Zuverlässigkeitsanforderungen an das Netzwerk gesendet. Diese Bestätigung wird wieder durch alle Schichten der Softwarearchitektur bis zur Netzwerkschicht weitergegeben. Zwischen zwei Schichten der Softwarearchitektur muss jeweils ein entsprechendes Protokoll angewendet werden, um zu entscheiden, ob die Dateninformation weitergegeben oder innerhalb der aktuellen Schicht verarbeitet wird. Mit Hilfe des im Netzwerk angewendeten *Broadcast*-Mechanismus wird das Datenpaket zum nächsten Netzwerkknoten gesendet.

Eine Softwaresimulation des beschriebenen Vorgangs belastet die Prozesseinheit und den Speicher der Simulationsplattform, dabei ist dieses Problem bei busbasierten Simulationen häufig der Fall. Obwohl die verwendeten Softwarekomponenten je nach simulierten

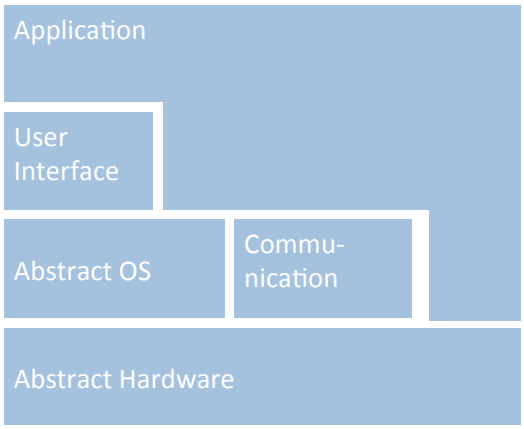


Abbildung 5.5.: Douglass *Five-Layer-Pattern* Architektur

Netzwerkknoten stark variieren können, ist diese Problemstellung präsent und wird durch den hier vorgestellten Ansatz adressiert. Durch das Verhaltensmuster Active Cross-Layer Balancing wird die Kommunikationslast zwischen einzelnen Schichten einer Softwarearchitektur reduziert und in die Rechenlast in die Schichten selbst verlagert. Dazu wird eine aktive Kommunikation basierend auf einem separaten Prozess, beschrieben in den folgenden Abschnitten 5.2.3, 5.2.3, und eine zwei-stufige Modifikation des *Observer* dieses Entwurfsmusters in Abschnitt 5.2.3 eingesetzt. Im Vorfeld werden die Voraussetzungen für das neue Verhaltensmuster in Abschnitt 5.2.1 beschrieben und in Abschnitt 5.2.2 in Beziehung zu verwandten Arbeiten gesetzt. Das Fazit in Abschnitt 5.2.4 schließt die Beschreibung des Active Cross-Layer Balancing Verhaltensmusters mit einer Evaluierung ab.

5.2.1. Voraussetzungen

Abbildung 5.5 zeigt die von Douglass vorgeschlagene Softwarearchitektur [12] für den Einsatz in zeitkritischer Software. In diesem Abschnitt werden erst die Nachteile der Architektur bezüglich der Problemstellung der vorgelegten Arbeit aufgezeigt und dann vorgestellt, wie diese Nachteile durch den Einsatz des Active Cross-Layer Balancing Verhaltensmusters aufgelöst werden können. Die in der Abbildung gezeigte Schichtenarchitektur wurde hauptsächlich für echtzeitfähige Software entworfen. Active Cross-Layer Balancing erhöht die Effizienz der Datenübertragung zwischen den einzelnen Schichten

und erhöht damit die Performanz der Software, ohne die zeitkritischen Eigenschaften der Architektur zu negieren. Eine Software basierend auf der resultierenden Softwarearchitektur kann in zeitheterogenen Netzwerken eingesetzt werden, um die in Kapitel 3 vorgestellte Problemstellung für Datenübertragung, insbesondere Datenfluss, zu lösen.

Die ersten beiden Schichten der ursprünglichen Architektur sind die Hardwareabstraktion und die Abstraktionsschicht der Systemfunktionen. Beide Schichten werden nicht durch das neue Verhaltensmuster modifiziert. Die Hardwareabstraktion kapselt entsprechende Funktionalität vor den höheren Schichten und enthält in einer konkreten Implementierung alle Treiber für Sensoren, Aktoren und die Protokolle der Netzwerkkommunikation. Auf der Ebene der Systemfunktionen werden wesentlichen Funktionen zur Ablaufplanung, Speicherzugriff und Zeitgeber veröffentlicht und den höherer Schichten der Software zur Verfügung gestellt. Die Kommunikationsschicht veröffentlicht Funktionen für die Datenübertragung, das Marshalling von Nachrichten und die Verbindungsverwaltung. Über dieser Schicht befinden sich die Benutzerschnittstelle und die Anwendungsschicht. In der Definition nach Douglass ist den einzelnen Schichten jeweils nur der *top-down*-Zugriff auf andere Schichten gestattet. Für jede Schicht ist deswegen nur die nächstniedrigere Schicht sichtbar. Solch eine Architektur bietet eine gute Kapselung der einzelnen Abstraktionsschichten und erhöht damit die Austauschbarkeit der Schichten untereinander. Gerade im Bereich eingebetteter Systeme ist das ein essentieller Vorteil, wenn Software- und Hardwarekomponenten oft ersetzt oder erweitert werden. Im Umkehrschluss wird die Effizienz gesenkt, zum Beispiel ist nur unidirektionale Kommunikation möglich und es sind keine Daten aus einer übergeordneten Schicht sichtbar. Aus diesem Grund muss beim Entwurf konkreter Software für eingebettete Systeme festgelegt werden, ob eine solche Schichtenarchitektur oder eine andere Architektur mit besserer Performanz verwendet wird. Der hier vorgestellte Ansatz löst diese Gegensätze auf. Um Wiederverwendbarkeit und Modularität für Softwarearchitekturen eingebetteter Systeme zu unterstützen, besteht oft nur eine lose Kopplung zwischen den einzelnen Schichten. Für die Kommunikation zwischen diesen Schichten werden gemeinsame Protokolle definiert. Dabei müssen diese Protokolle bidirektionale Datenübertragung realisieren. Einerseits werden Dateninformationen von der untersten Schicht bis in die Anwendungsebene weitergegeben, wenn zum Beisiel die Software Daten aus dem Netzwerk empfängt. Andererseits muss die Software initiativ zum Beispiel Sensordaten über das Netzwerk versenden. Dann werden Daten von Anwendungsschicht bis runter in die Hardwareabstraktion übertragen. Muss für einen konkreten Entwurf eine bidirektionale Datenübertragung berücksichtigt werden, sind essentielle Strukturen der resultierenden Software nicht durch Douglass Ansatz abgedeckt. Innerhalb des Protokollentwurfs muss auch berücksichtigt werden, ob eine Schicht mit allen Schichten auf denen diese basiert oder nur mit der nächstniedrigeren kommunizieren können soll. Der weniger stringente Ansatz führt zu

einer sogenannten offenen Architektur, bei strenger Interpretation von Douglass Ansatz handelt es sich um eine geschlossene Architektur. Der Vorteil einer offenen Architektur im Vergleich zu einer geschlossenen Architektur ist die geringere Latenz beim Versenden von Nachrichten zwischen weiter entfernten Schichten. Der Nachteil ist die stärkere Kopplung, beim Ersetzen oder Erweitern einer Schicht zu einem erhöhten Aufwand führt. Das Active Cross-Layer Balancing Verhaltensmuster definiert eine geschlossene Architektur, um die Modularität zu erhalten. Eine essentielle Schwierigkeit der ursprünglichen Architektur ist das stetige *Polling*, wenn der Datenfluss oder Kontrollfluss nicht von der Anwendungsschicht nach unten sondern umgekehrt entworfen werden muss. Diese aktive Abfrage senkt die Effizienz der Softwarearchitektur, wenn eine Datenübertragung entgegen der von Douglass vorgeschlagenen Richtung erfolgt. Das Verhaltensmuster Active Cross-Layer Balancing führt eine bidirektionale einsetzbare Datenübertragung ein, um dieses Problem zu lösen.

Eine weitere Verbesserung der ursprünglichen Softwarearchitektur ist die Definition eines gemeinsamen Protokolls zwischen den einzelnen Schichten. Eine Protokolldefinition ist in architektonischen Ansätzen nicht unbedingt vorhanden und wird auch in der Arbeit von Douglass nicht erwähnt. Durch die Einführung eines leichtgewichtigen Protokolls zur Datenübertragung zwischen einzelnen Schichten wird die Kapselung, die Modularität und die Erweiterbarkeit innerhalb der Architektur weiter erhöht.

Das Ergebnis der zusammengefassten Probleme und Lösungen ist eine ereignisgesteuerte und bidirektionale Datenübertragung zwischen zwei benachbarten Schichten. Abschnitt 5.2.3 und folgende Abschnitte stellen das Active Cross-Layer Balancing Verhaltensmuster im Detail vor.

5.2.2. Verwandte Arbeiten

In dem Papier [71] präsentieren Srivastava und Motani verschiedene Ansätze für den Entwurf einer Schichten-übergreifenden Architektur und Kommunikation. Diese Arbeit fokussiert auf drahtlose Netzwerke und der Begriff *cross-layer* bezieht sich in ihrer Arbeit nur auf den Kellerspeicher der Kommunikationsprotokolle. Das Verhaltensmuster Active Cross-Layer Balancing ist nicht auf drahtlose Netze limitiert und auch nicht auf die Protokollspeicher beschränkt. Es kann allgemein eingesetzt werden, sobald die Softwarearchitektur in einzelnen Schichten aufgebaut ist.

Das Papier [68] zeigt einen vielversprechenden schichtenübergreifenden Ansatz für drahtlose Netzwerke. Durch die Verfügbarkeit von Informationen aus den physikalischen und MAC Schichten in den höheren Schichten einer Softwarearchitektur wird in diesem Ansatz die Effizienz des paketbasierten Datentransports erhöht. Der Fokus liegt hier entweder auf *Accesspoints* von Zellen oder Ethernet-basierter Datenübertragung.

Dieser Fokus auf Netzwerkschichten unterscheidet diesen Ansatz von dem hier vorgestellten Verhaltensmuster. Das Entwurfsmuster Active Cross-Layer Balancing minimiert die Menge an Dateninformationen, die von verschiedenen Schichten geteilt werden muss und entkoppelt damit die einzelnen Schichten soweit wie möglich voneinander.

Michelson [49] präsentiert den Einsatz eines ereignisgesteuerten Entwurfs einer *Service-Orientierten Architektur (SOA)*. Michelson zeigt, wie mächtig solche Systeme sein können und welche Vorteile sich durch den Einsatz ereignisgesteuerter Prozesse ergeben. Dieser Ansatz ist allerdings an SOA in Enterprise Applikationen gebunden und nur schwer für den Einsatz in eingebetteten Systemen portierbar. Das hier vorgestellte Entwurfsmuster fokussiert mehr auf eine leichtgewichtige Architektur und möglichst hohe Performanz.

Chenyang et al. präsentieren in [44] eben solche eine leichtgewichtige und performante Architektur für Sensornetzwerke. Durch Fokus auf Anfragen und Ereignis-verarbeitende Dienste innerhalb solcher Sensornetzwerke optimieren die Autoren das paketbasierte Scheduling. Solche Pakete werden unter Berücksichtigung ihrer Deadline, voraussichtliche Zeit bis zum Erreichen des Zielknotens und der Entfernung zu diesem Knoten geplant. Da Sensorknoten in solchen Netzwerken zumeist nur eine einzige Funktion realisieren, zum Beispiel die Übertragung einer gemessenen Temperatur zur Basisstation, sind ihre Architekturen flach und einfach strukturiert. Aus diesem Grund werden die besten Ergebnisse mit RAP durch Verbesserungen im Netzwerk *Protocol Stack* realisiert. Der Fokus des Entwurfsmusters Active Cross-Layer Balancing liegt auf vielschichtigen Architekturen für eingebettete Systeme. Netzwerkkommunikation wird in diesem Ansatz nicht berücksichtigt. Das Verhaltensmuster zielt auf das zeitliche Verhalten und die Leistung in diesen vielschichtigen Architekturen und der Kommunikation innerhalb der entsprechenden Softwarearchitektur.

5.2.3. Architektur

Ein wichtiger Teil des neuen Verhaltensmusters ist eine nicht-blockierende unidirektionale Kommunikation zwischen den einzelnen Schichten der Softwarearchitektur. Je nach Softwareplattform sind passende Datenstrukturen bereits implementiert, zum Beispiel die *WaitFreeDequeue* des *Java Real-time System* [23]. In Abschnitt 5.1 stellt die vorliegende Arbeit die neu entwickelte Decoupling Datastructure vor, die für diesen Anwendungsfall entwickelt wurde. Der Einsatz dieses Verhaltensmusters entkoppelt auch die Zuteilungen der einzelnen Prozesse auf Prozessoren zum Beispiel eines Mehrkernsystems voneinander, da jeder Prozess nur innerhalb einer Softwareschicht arbeitet. Es werden durch den Prozess selber keine Methoden anderer Schichten aufgerufen, da adjazente Schichten nur über die neue Datenstruktur adressiert werden dürfen. Um eine solche

Methode auszuführen, muss der Prozess mit Hilfe eines konkreten Protokolls ein entsprechendes Ereignis oder Kommando senden, das dann innerhalb der anderen Schicht ausgeführt wird. Das ist die typische Vorgehensweise in ereignisgesteuerten Systemen.

Nachrichtenverteilung

Abbildung 5.6 zeigt das Kommunikationsmodell, wie es das Verhaltensmuster Active Cross-Layer Balancing definiert. In der untersten Schicht der Softwarearchitektur wird eine beliebige Information erzeugt, muss verarbeitet werden oder wird durch das Netzwerk empfangen. Um diese Dateninformation in eine höhere Architekturschicht zu übertragen, wird diese in einer entsprechenden Datenstruktur, zum Beispiel eine Decoupling Datastructure, gespeichert. Konzeptionell entspricht die gewählte Datenstruktur einem Kommunikationskanal in der Grenze zwischen zwei benachbarten Schichten. Die Verarbeitung der in der Datenstruktur eingefügten Information erfolgt nicht direkt in der nächsten Schicht der Softwarearchitektur, sondern durch einen *Nachrichten Dispatcher*. Dieser Nachrichten Dispatcher ist ein Prozess, der innerhalb der Softwareschicht ausgeführt wird, in welcher die Dateninformationen ausgewertet werden müssen. Dabei werden die einzelnen Dateninformationen durch den Dispatcher mit Hilfe einer für diese Datenübertragung dedizierten *Policy* ausgewertet. Der Dispatcher dekoriert die Dateninformation entsprechend der Policy in eine *MetaType* Instanz und fügt diese in eine Warteschlange ein. Danach wird ein Ereignis innerhalb der Softwareschicht propagiert. Potentielle Empfänger der Dateninformation werden über dieses Ereignis informiert, dass eine neue Dateninformation darauf wartet, verarbeitet zu werden. Die Verarbeitung erfolgt asynchron zum Eintreffen der Information in der entsprechenden Softwareschicht.

Die für die Realisierung des Verhaltensmusters Active Cross-Layer Balancing benötigten Klassen und ihre Abhängigkeiten werden in Abbildung 5.7 vorgestellt. Als zentrale Komponente hat der Nachrichten Dispatcher Zugriff auf den Kommunikationskanal über die Methode `read()`. Die `write()` Methode des Kommunikationskanals wird nicht durch den Dispatcher, sondern durch einen Prozess aus derjenigen Softwareschicht ausgeführt, die Dateninformationen in die Softwareschicht des Dispatchers übertragen soll. Der Nachrichten Dispatcher instanziiert auch die MetaType Objekte anhand einer Policy. Diese Vorschrift ist System-spezifisch zu implementieren. Zum Beispiel könnte eine Policy festlegen, in wie fern durch das Netzwerk empfangene Dateninformationen dekodiert und an welche Empfänger innerhalb der Applikation gesendet werden müssen. Der Nachrichten Dispatcher bestimmt mit Hilfe einer applikationsweiten Registrierung, welche Empfänger die in der Datenstruktur eingefügte Dateninformation erhalten sollen. Eine entsprechende Empfängerliste wird in jeder MetaType Instanz vorgehalten. Die Klasse MetaType stellt weitere Methoden zur Verarbeitung der dekorierten Daten-

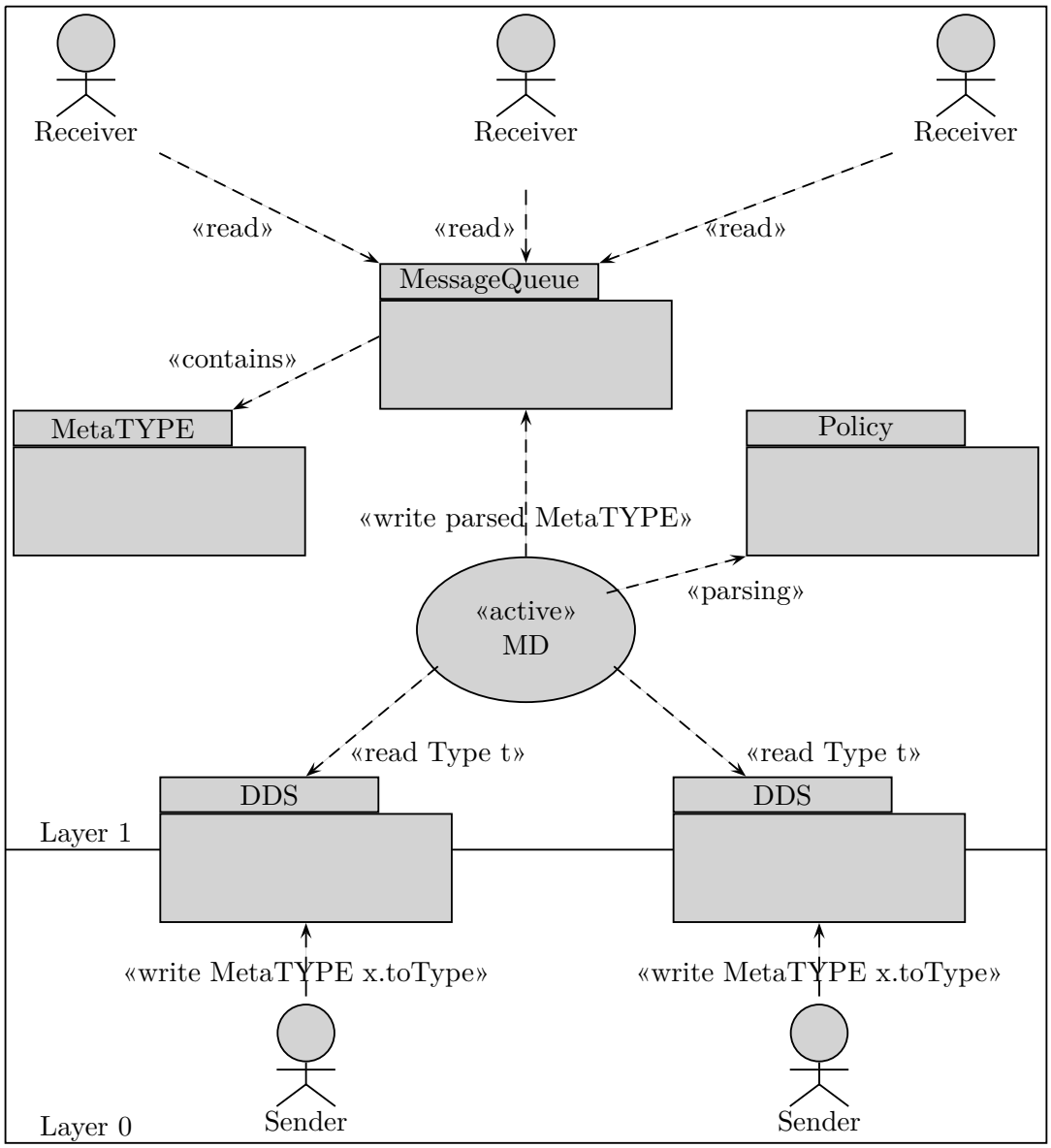


Abbildung 5.6.: Modell der Kommunikation

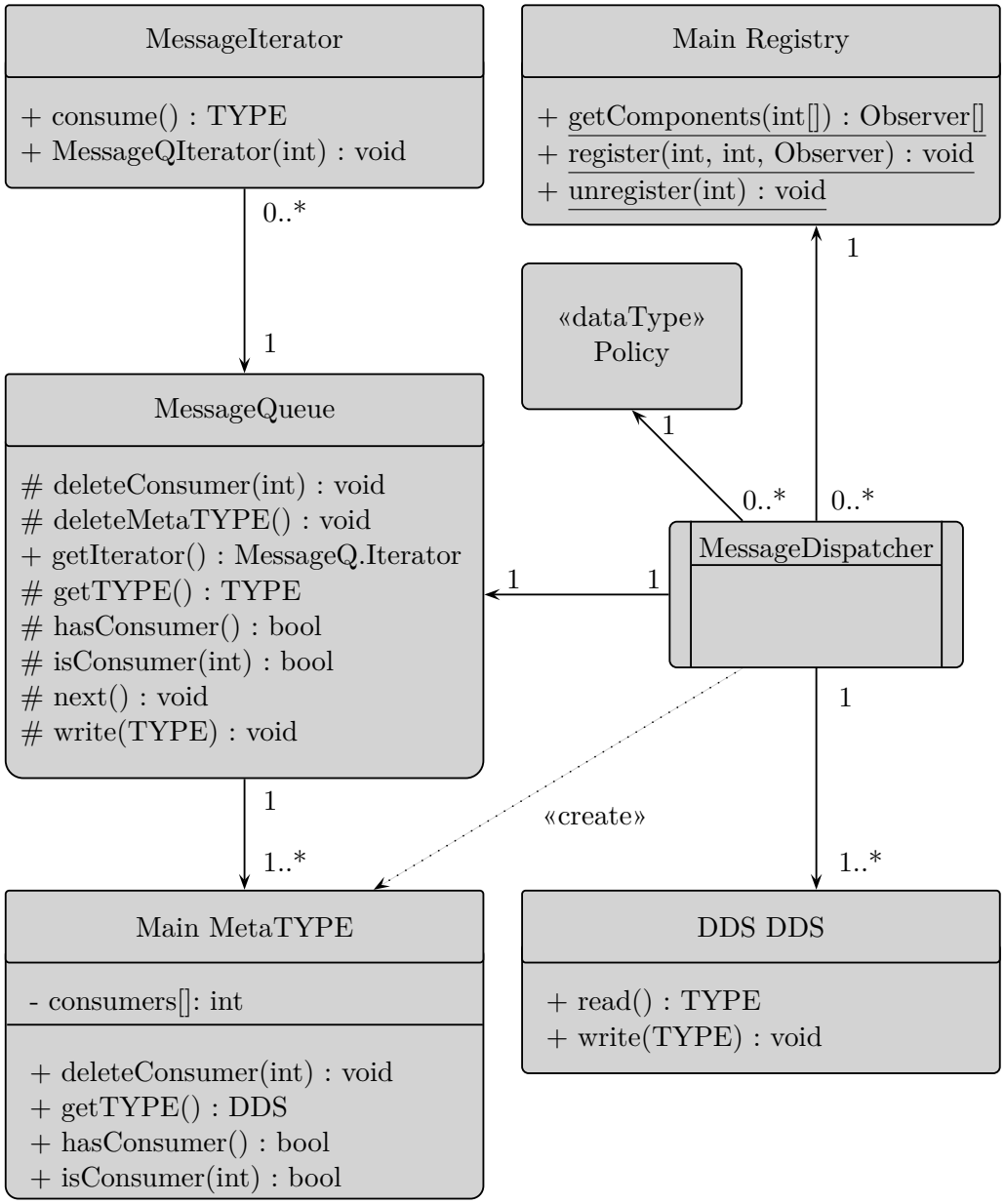


Abbildung 5.7.: Klassendigramm des Kommunikationskanals

informationen zur Verfügung, die durch die in der Empfängerliste vorgehaltenen Verbraucher aufgerufen werden können, um die Dateninformation zu verarbeiten. Nachdem der Dispatcher eine Dateninformation dekoriert hat, wird die MetaType Instanz in eine Warteschlange eingefügt und dort vorgehalten, bis alle Verbraucher diese Information verarbeitet haben. Ein durch das Verhaltensmuster Active Cross-Layer Balancing vorgegebener *Nachrichten Iterator* liefert die Schnittstelle für Verbraucher, um die einzelnen MetaType Instanzen innerhalb der Warteschlange zu iterieren. Die Warteschlange selber ist generisch, sollte aber für den konkreten Anwendungsfall eine ausreichende Größe und Durchsatz haben. MetaType Instanzen werden in dieser Warteschlange vorgehalten, bis alle Prozesse beziehungsweise Komponenten der entsprechenden Softwareschicht die dekorierten Dateninformationen verarbeitet haben. Zum Beispiel könnte ein aus dem Netzwerk empfangenes Datenpaket in einer Softwareschicht verarbeitet werden, aber in einer anderen Schicht durch einen *Logger* Prozess aufgezeichnet werden. Um die Dateninformation schließlich durch die einzelnen Verbraucher zu verarbeiten, wird das Beobachter-Entwurfsmuster um den Nachrichten Iterator erweitert. Diese Vorgehensweise entlastet den Nachrichten Dispatcher, der seine zur Verfügung gestellte Prozessorzeit für das Auslesen des Kommunikationskanals, das Dekorieren der Dateninformationen und Einfügen in die Warteschlange nutzen kann.

Nachrichten Dispatcher und die einzelnen Beobachter beziehungsweise Verbraucher sind lokale Instanzen mit begrenzten Gültigkeitsbereichen. Damit unterstützt das Verhaltensmuster *Information Hiding*, muss aber mit einer globalen Registrierung arbeiten, die im Allgemeinen als *Singleton* implementiert werden kann. In diese Registrierung können Verbraucher angemeldet werden, die spezifische MetaType-Objekte verarbeiten müssen. Für jede Dateninformation, die dekoriert werden muss, kann ein Nachrichten Dispatcher über die Methode `getComponents()` die aktuelle Liste der für diesen MetaType angemeldeten Verbraucher ermitteln. Dem Beobachtermuster folgend muss der Nachrichten Dispatcher observierbar implementiert sein. Dies geschieht im Allgemeinen durch den synchronen und sequentiellen Aufruf der `update()` Methode aller angemeldeten Verbraucher, die für eine entsprechende Dateninformation angemeldet sind. Solche Verbraucher können beliebige Softwarekomponenten oder Prozesse sein. Durch den Aufruf der `update()` Methode wird synchron nur ein dediziertes Zählerfeld der Verbraucher durch den Aufruf der Methode `inc()` erhöht. Dadurch wird das Ereignis des Eintreffens einer Dateninformation von der Verarbeitung dieser Information entkoppelt. Die Verarbeitung der Dateninformation wird anschließend durch das Scheduling der Verbraucher bestimmt.

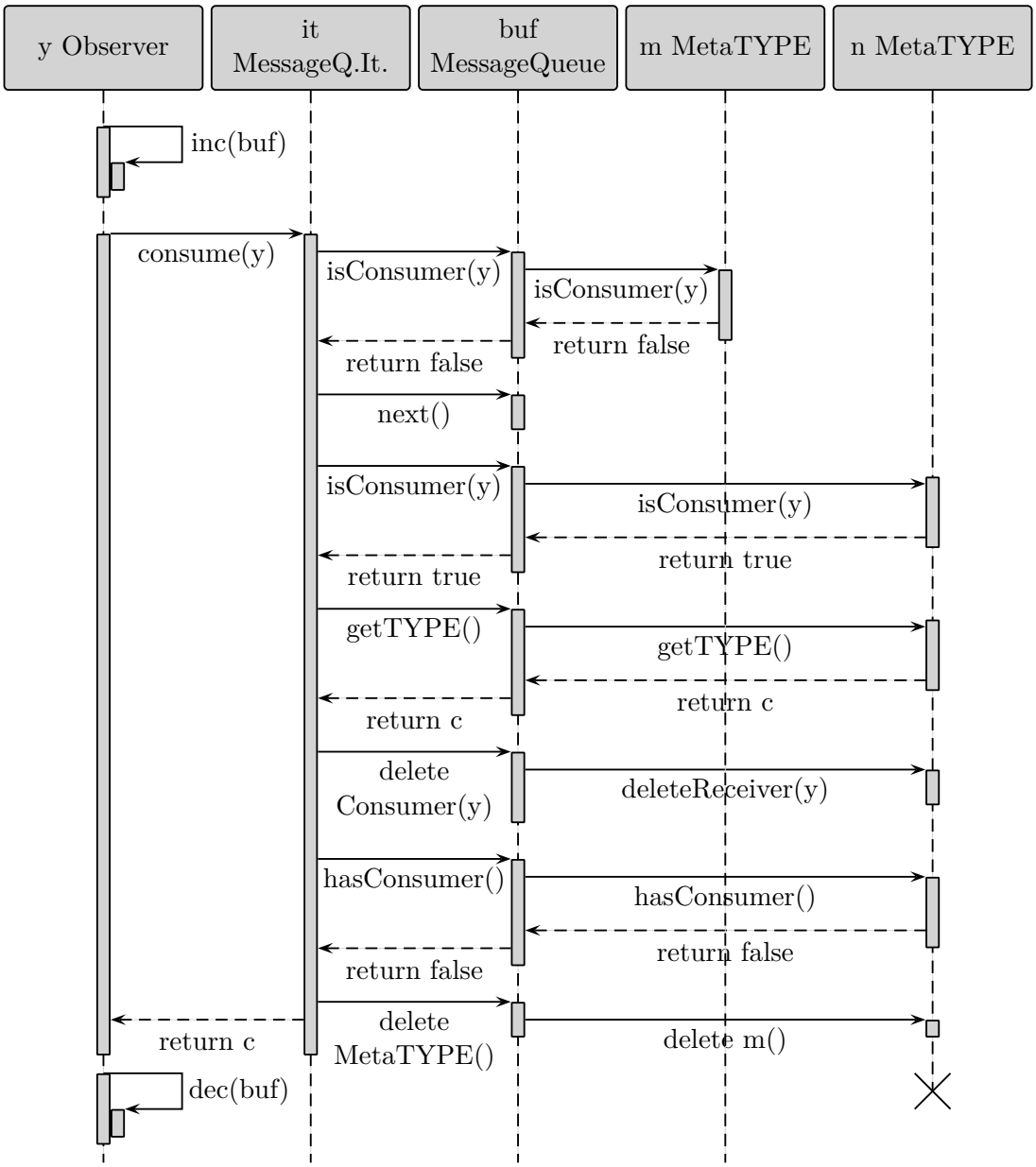


Abbildung 5.8.: Sequenz der Nachrichtenverarbeitung

Nachrichten verarbeiten

Nachdem die dekorierte Dateninformation in der Warteschlange gespeichert wurde und die potentiellen Verbraucher dieser Information durch den Dispatcher aktualisiert wurden, obliegt es dem jeweiligen Verbraucher, diese Dateninformation asynchron zu dem eigentlichen Verteilungsvorgang zu verarbeiten. Zu diesem Zweck wird ein Nachrichten Iterator eingeführt, der die Iteration über die Dateninformationen in der Warteschlange kapselt. Abbildung 5.8 zeigt den Ablauf der Nachrichtenverarbeitung. Der Nachrichten Dispatcher speichert jede dekorierte Dateninformation in einer einzigen Warteschlange. Aus diesem Grund führt ein für einen bestimmten Verbraucher dedizierter Iterator eine gefilterte Iteration über dieser Warteschlange aus. Ob eine spezifische Dateninformation durch seinen Verbraucher verarbeitet werden muss, kann eine Instanz des Iterators über die Methode `isConsumer()` der Warteschlange überprüfen. Dieser Aufruf wird an die MetaType Instanzen delegiert, über die gerade iteriert wird. Ist der Verbraucher kein Adressat des gerade iterierten MetaType, wird die `next()` Methode des dedizierten Iterators eingesetzt, um auf die nächste MetaType Instanz in der Warteschlange zu wechseln. Muss die aktuelle MetaType Instanz verbraucht werden, wird die Methode `getType()` aufgerufen, die wieder an die Instanz delegiert wird. Der Rückgabewert dieser Methode entspricht der ursprünglichen Dateninformation. Bevor jetzt der Iterator die Dateninformation an den Verbraucher weitergibt, wird dieser Verbraucher aus der Liste der Adressaten, die in der Metatype Instanz vorgehalten werden, gelöscht. Dann überprüft der Iterator, ob noch weitere Adressaten in der Liste sind. Ist dies nicht der Fall, wird die MetaType Instanz und damit die Dateninformation aus der Warteschlange gelöscht und dekonstruiert. Als letzter Schritt wird das Zählerfeld des Verbrauchers durch den Iterator um eins dekrementiert.

Nachricht-Dispatching in Netzwerken

In diesem Abschnitt wird der Spezialfall betrachtet, wenn mehrere Prozesse innerhalb eines Netzwerks zur Laufzeit Dateninformationen an das Netzwerk senden. Zum Beispiel muss in dem industriellen Szenario des Kooperationsprojekts mit einem Unternehmen aus der Licht- und Signaltechnikindustrie spezifische Netzwerkknoten periodische *Keep Alive* Informationen an eine *Watchdog* Instanz senden und gleichzeitig weitere Netzwerkknoten miteinander Daten austauschen, während eine dritte Komponente Dateninformationen generiert und versenden muss. Abbildung 5.9 skizziert das beschriebene Szenario unter Berücksichtigung des Verhaltensmusters Active Cross-Layer Balancing. Der Nachrichten Dispatcher Prozess entnimmt Dateninformationen aus allen Kommunikationskanälen, die durch Softwarekomponenten aus höheren Schichten der Architektur

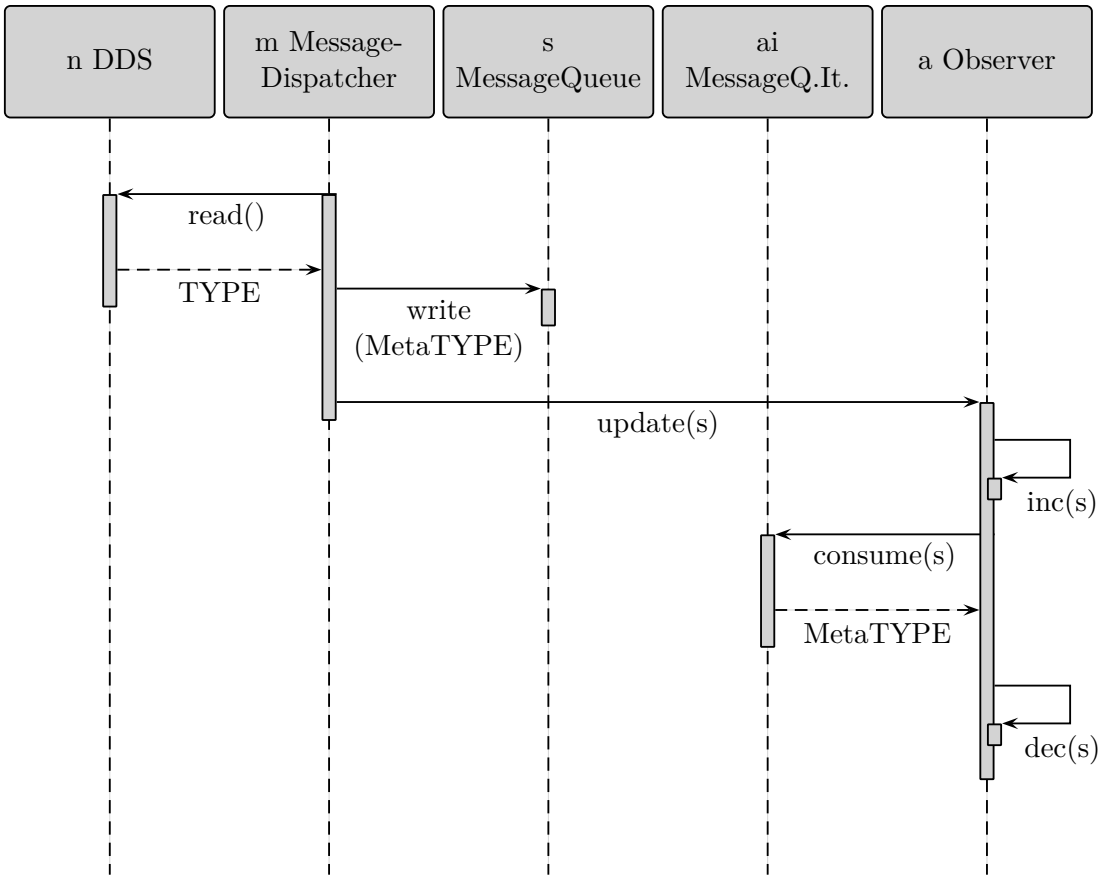


Abbildung 5.9.: Nachrichtenverteilung in das Netzwerk

parallel eingefügt werden, und speichert diese in seiner Warteschlange ab. Ein einzelner Verbraucher kann nun diese Warteschlange iterieren, verarbeitet die Informationen gemäß dem implementierten Netzwerkprotokoll, und sendet an das Netzwerk. Im Rahmen des industriellen Szenarios war das ein *CAN Controller*, der die Nachrichten sequentiell in das Bussystem sendet. Während also Komponenten höherer Schichten parallel arbeiten können, erlaubt das Verhaltensmuster auch weiterhin eine serielle und fehlerfreie Netzwerkkommunikation.

5.2.4. Fazit

Andere Softwarearchitekturen zielen auf eine verbesserte Zuverlässigkeit innerhalb der Architektur. Aber keine dieser Strukturen liefert einen integrierten Ansatz, um einerseits die Zuverlässigkeit und andererseits die Problemstellung zu lösen, die durch eine hohe Kommunikationslast in einem Netzwerk entstehen kann. Das Verhaltensmuster Active Cross-Layer Balancing löst dieses Problem durch den Ansatz, über die Grenzen von Softwareschichten hinweg eine aktive Kommunikation zu etablieren. Dateninformationen zum Beispiel aus der Netzwerkebene werden nach der Anwendung des Entwurfsmusters schneller und effizienter auf die Softwareschicht übertragen, auf der die Informationen verarbeitet werden soll. Dabei bleibt die strikte Trennung der einzelnen Softwareschichten durch die Definition der MetaType Policy erhalten. Das Verhaltensmuster Active Cross-Layer Balancing kann basierend auf einer passenden Datenstruktur, wie zum Beispiel die Decoupling Datastructure aus Abschnitt 5.1, eine bidirektionale prinzipiell asynchrone Datenübertragung realisieren. Dadurch wird die auf der Netzwerkschicht entstehende Kommunikationslast schneller an die höheren Softwareschichten weitergegeben. Das führt zu einer Entlastung der Netzwerkschicht und erlaubt eine höhere Gesamtauslastung des Netzwerks oder eine erhöhte Verlässlichkeit der Datenübertragung. Dieser Ansatz konnte in dem Kooperationsprojekt mit der Licht- und Signaltechnikindustrie erfolgreich eingesetzt werden.

Das vorgestellte Verhaltensmuster erhält die modulare Struktur und Kapselung der ursprünglichen Architektur von Douglass. Über eine konkrete Implementierung zum Beispiel auf einer Hardwareplattform mit einem Mehrkernprozessor kann das Entwurfsmuster skaliert werden, wenn zwischen zwei Softwareschichten eine besonders hohe Auslastung zu erwarten ist. Hier können zum Beispiel die einzelnen Nachrichten Dispatcher auf separaten Kernen ausgeführt werden.

5.3. Adaptive Controlflow Transitions

In diesem Abschnitt wird das Verhaltensmuster *Adaptive Controlflow Transitions* vorgestellt. Dieses Entwurfsmuster ermöglicht es, vernetzten eingebetteten Systemen einen echtzeitfähigen Wechsel des Netzwerkbetriebs auf allen Netzwerkknoten durchzuführen und dabei insbesondere die dort benötigten Ressourcen unmittelbar frei zugeben. Damit liefert das Verhaltensmuster eine auf die Problemstellung in Kapitel 3 zugeschnittene Variante eines netzwerkweiten Kontrollflusses. Dieses Verhaltensmuster wurde in gemeinschaftlicher Arbeit mit Thomas Gerlitz entwickelt und prototypisch auch im Kooperationsprojekt mit der Avionikindustrie getestet. Im Anschluss wurde zu diesem Verhaltensmuster ein Papier [62] veröffentlicht. Im Kooperationsprojekt mit einem Unternehmen aus der Avionikindustrie wurde eine Softwaresimulation entwickelt, in der mehrere Applikationen auf einer einzelnen Softwareplattform ausgeführt werden. Die einzelnen Applikationen mussten dabei dynamisch Ereignisse auswerten und Daten übertragen können. Deswegen war das erklärte Entwurfsziel der Softwaresimulation keine Architektur, die eine statisch vorhersagbare Laufzeit ermöglicht, sondern eine dynamische *Quality of Service*-Anpassung auf Anwendungsebene. Da die Softwaresimulation auf einem handelsüblichen Rechnersystem ausgeführt wird, ist ein mögliches Laufzeitszenario, dass eine oder mehrere der Applikationen, welche die Softwaresimulation realisieren, ausfällt und nicht mehr verfügbar sind. Das Verhaltensmuster Adaptive Controlflow Transitions kann natürlich auch allgemein in vernetzten eingebetteten Systemen eingesetzt werden.

Zum Beispiel wird ein Elektrofahrzeug vollständig *Drive-By-Wire* gesteuert. Dieses Fahrzeug besitzt eine verteilte Architektur für die Steuerung der Motoren und Räder, bei der jedes der vier Räder durch jeweils einen Elektromotor angetrieben wird, von denen jeder wiederum durch jeweils einen Controller gesteuert wird. Obwohl hier Robustheit ein wichtiges Entwurfsziel ist, muss bei der Entwicklung des Fahrzeugs auch Ausfallsicherung zum Beispiel durch *Fault Tolerance* berücksichtigt werden. Um hier einen dynamischen Quality of Service zu realisieren, können verschiedene Betriebsarten innerhalb des Netzwerkes vorgesehen werden, die den Ausfall einzelner Komponenten berücksichtigen. Fällt zum Beispiel ein Radcontroller im Next Generation Fahrzeug aus, muss dessen Aufgabe entweder auf die anderen Controller im Netzwerk umgesetzt werden oder das gesamte Netzwerk wechselt in eine andere Betriebsart, die weniger Rechenkapazitäten verbraucht.

Adaptive Controlflow Transitions definiert eine generische Architektur, mit der verschiedene Betriebsmodi implementiert werden können, um eine dynamischen Quality of Service Anpassung für die Software oder Softwareplattform zu realisieren. Dieses Verhaltensmuster erweitert das Konzept *Graceful Degradation* [1, 30] auf einen flexiblen Ansatz, der adaptive Transitionen definiert, um entweder die Rechenlast des Netzwerkes

zu erhöhen oder die Qualität der gemeinschaftlichen Berechnung aller Netzwerkknoten zu reduzieren. Das führt zu einem bezüglich der Rechenlast und Qualität der Berechnung kooperativ arbeitenden Netzwerk. In Abschnitt 5.3.1 wird die Problemstellung vertieft und dann in Abschnitt 5.3.2 ein Konzept vorgestellt, wie ein kooperatives Netzwerk aufgesetzt werden kann. Dieses Konzept wird in Abschnitt 5.3.6 evaluiert und in Abschnitt 5.3.7 in Bezug zu verwandten Arbeiten gesetzt. Der Abschnitt 5.3.8 schliesst die Vorstellung des Verhaltensmusters Adaptive Controlflow Transitions ab.

5.3.1. Kooperation von Last und Qualität

Zur Laufzeit verteilter eingebetteter Systeme können dynamische Ereignisse eintreten, deren zeitliches Verhalten nicht vorhersagbar ist. Die Verarbeitung dieser Ereignisse dürfen die zeitkritischen Steuerungsvorgänge nicht unterbrechen. In dem oben beschriebenen Next Generation Elektrofahrzeug könnte zum Beispiel einer der Controller ausfallen. In der aktuellen Generation von Elektrofahrzeugen führt dieser Fehler zu Graceful Degradation, die ein bekanntes Konzept für sicherheitskritische Systeme ist. In diesem sicheren Betriebsmodus würde das Fahrzeug kontrolliert stoppen, im Idealfall ohne Insassen zu gefährden. Dieses Konzept kann natürlich auch in nächsten Generationen verwendet werden. Aber das Verhaltensmuster Adaptive Controlflow Transitions erweitert diesen Ansatz um die Interpretation, dass verteilte eingebettete Systeme ein kooperatives Netzwerk bilden, um ein gemeinsames Ziel zu erreichen. Wenn in Abhängigkeit von einem dynamisch eintreten Ereignis mehr Ressourcen benötigt werden oder bisher verwendete Ressourcen nicht mehr zur Verfügung stehen, müssen die restlichen Netzwerkknoten so adaptiert werden, dass in Kooperation dieser Knoten das gemeinsame Zeil weiterhin erreicht wird. Zur Beschreibung des Verhaltensmusters wird ein kooperatives Netzwerk wie folgt definiert:

Definition 7: Kooperative Netzwerke Ein kooperatives Netzwerk ist eine Menge C von verteilten eingebetteten Systemen, die physikalisch oder logisch über eine Netzwerktopologie T verbunden sind. Elemente dieser Menge erfüllen kooperativ eine spezifische Hauptfunktion. ◇

Die Qualität der Ausführung der Hauptfunktion eines einzelnen eingebetteten Systems ist deswegen abhängig von der Qualität der anderen Elemente des Netzwerks.

Das gemeinsame Ziel des Netzwerks im Elektrofahrzeug wäre zum Beispiel die allgemeine Funktionsfähigkeit des Fahrzeugs. Fällt ein Controller in diesem Netzwerk aus, dann müssen die verbliebenen drei Controller und damit Motoren eingesetzt werden, um dieses Ziel auch weiterhin zu erreichen. Einerseits kann die Rechenlast des ausgefal-

lenen Controllers auf die verbliebenen Controller aufgeteilt werden. Damit erhöht sich die Rechenlast dieser Controller entsprechend und der Energieverbrauch steigt. Auf diesem Weg werden die vier Motoren durch insgesamt drei Controller ohne Qualitätsverlust steuerbar, wenn die Ergebnisse der einzelnen Systemen zum Beispiel über das Bussystem des Fahrzeugs ausgesendet werden. Andererseits kann die Qualität der Ergebnisse reduziert werden. Dabei bleibt die Rechenlast auf den einzelnen verbliebenen Controllern konstant. Im obigen Beispiel könnte die Frequenz der Iterationen verringert werden, in denen die Motorenleistung innerhalb eines Regelkreises, vergleiche Abbildung 3.2, aktualisiert wird. Das Konzept sollte auch vorsehen, dass ein als Standard festgelegter Operationsmodus nach der Verarbeitung des Ereignisses wieder angenommen werden kann, zum Beispiel wenn im Next Generation Elektrofahrzeug der vierte Controller nach einem Neustart wieder zur Verfügung steht.

5.3.2. Adaptive Transitionen

In diesem Abschnitt wird das Verhaltensmuster Adaptive Controlflow Transitions vorgestellt, das die im letzten Abschnitt beschriebenen Anforderungen erfüllt. Das Verhaltensmuster wird anhand der Berechnung geometrischer Figuren dargestellt. Dabei werden die drei folgenden Betriebsmodi berücksichtigt.

- Standard
- Erhöhte Rechenlast
- Verringerte Qualität

In Abbildung 5.10 wird der Vorgang einer Transition ausgehend vom Standardmodus des Netzwerks in einen adaptierten Modus dargestellt. In dieser Kooperation arbeiten die Netzwerkknoten zusammen, um die Ränder einer geometrischen Figur zu berechnen. Das gemeinsame Ziel der einzelnen Knoten ist also die Berechnung eines geschlossenen Polygons. Die Aufgabe wird durch vier separate Prozesse bearbeitet, die zum Beispiel auf vier einzelnen Prozessoren ausgeführt werden oder über das Ethernet kommunizieren. Im Standardmodus wird jede der vier Kanten eines Quadrats durch einen Netzwerkknoten berechnet. Fällt der Knoten, der L_1 berechnet, aus, wird die entsprechende Kante nicht mehr berechnet. Das Netzwerk befindet sich in einem Fehlerzustand. Jetzt kann das Netzwerk entweder eine *Computational Promotion* oder eine *Quality Degradation* durchführen, um den Betriebszustand anzupassen und so den Fehlerzustand wieder zu verlassen. Bei einer Computational Promotion könnte die Berechnung von Kante L_1 zu gleichen Teilen durch die noch lauffähigen Prozesse übernommen werden. Findet eine Quality Degradation statt, berechnet das System ab diesem Zeitpunkt ein Dreieck als

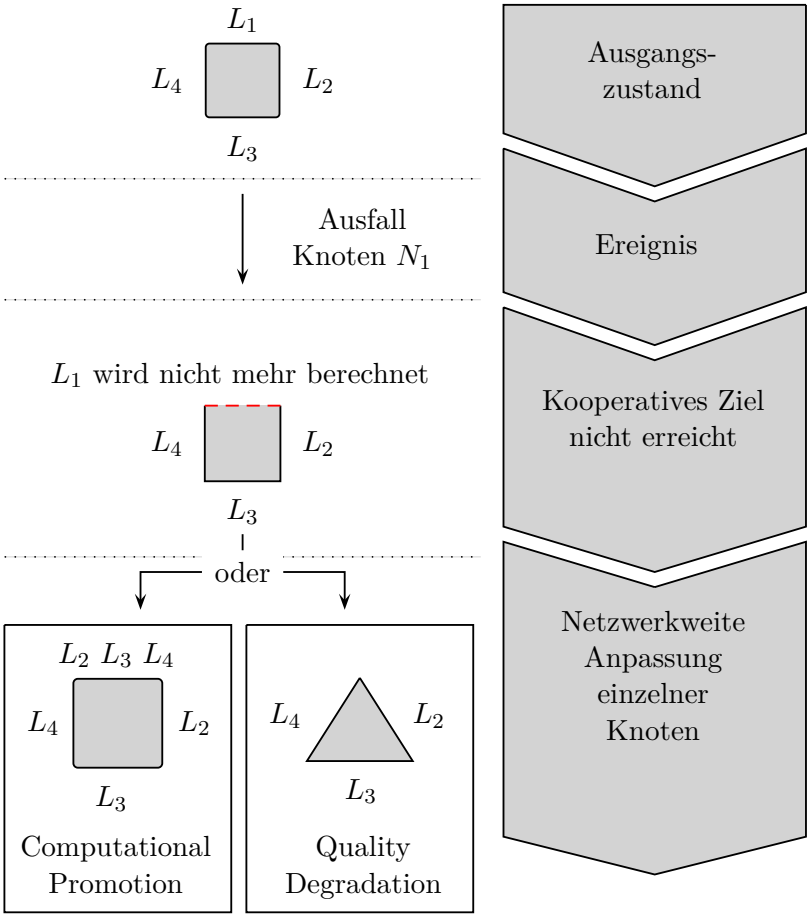


Abbildung 5.10.: Beispiel einer Kontrollflusstransition

ein ebenfalls geschlossenes Polygon. Die Auswahl der Strategie ist abhängig von der konkreten Anwendung. Das hier vorgestellte Verhaltensmuster definiert eine adaptive Transition wie folgt:

5.3.3. Adaptive Transitionen

Ein kooperatives Netzwerk C wechselt von einem Betriebsmodus in einen anderen Modus durch einen der folgenden adaptiven Transitionen:

- *Computational Promotion*: Die Rechenlast einer Teilmenge von C wird erhöht, um die Qualität des Berechnungsergebnisses des gesamten Netzwerkes konstant zu halten. Rechenlast wird hier als Synonym für erhöhten Ressourcen- oder Energieverbrauch eingesetzt.
- *Quality Degradation*: Reduktion der Qualität des Berechnungsergebnisses des gesamten Netzwerkes, um die Rechenlast einer Teilmenge von C konstant zu halten, wenn eine weitere Teilmenge zeitweise oder permanent nicht zur Berechnung zur Verfügung steht.

Einem kooperativen Netzwerk wird genau in den drei folgenden Fällen ermöglicht, adaptive Transitionen durchzuführen, wenn:

1. Innerhalb des Netzwerkes existiert ein Koordinatorknoten. Über diesen Knoten werden die zu verarbeitenden dynamischen Ereignisse des Netzwerkes koordiniert.
2. Jeder Knoten des kooperativen Netzwerkes implementiert eine Schnittstelle, über die der Koordinatorknoten Transitionen in verschiedene Betriebsmodi erzwingen kann.
3. Eine dedizierte Verhaltenstabelle wird von jedem Netzwerkknoten implementiert. Wenn durch den Koordinatorknoten eine Transition in einen anderen Betriebsmodus erzwungen wird, gibt diese Tabelle den zu einzusetzenden Algorithmus vor.

Die Anzahl Promotionen oder Degradierungen wird nur durch die Anzahl implementierter Betriebsmodi begrenzt. Der Ansatz geht dabei davon aus, dass die Aufrechterhaltung der Qualität des netzwerkweiten Ergebnisses proportional zur Erhöhung der Rechenlast zur Folge hat.

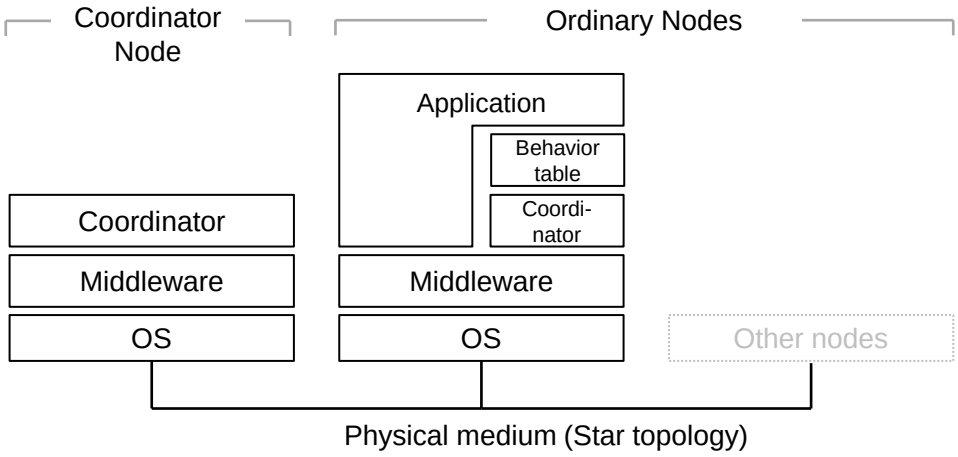


Abbildung 5.11.: Einfache kooperative Architektur

5.3.4. Koordination

Verteilte eingebettete Systeme werden typischerweise in Netzwerktopologien wie der Stern-, Ring oder Bustopologie organisiert, können aber auch in anderen Topologien organisiert sein. Die verwendete Topologie hat zusammen mit der realisierten Kommunikation einen großen Einfluss, wie das kooperative Netzwerk und die adaptiven Transitionen zu implementieren sind. Abbildung 5.11 zeigt eine potentielle Architektur eines kooperativen Netzwerkes als OSI-Modell[4]. Die vorgeschlagene Architektur beruht auf der Anwendungsschicht der jeweiligen Knoten und einer ausreichenden Effizienz der Schedulingmechanismen und Netzwerkdienste. Wenn ein Ereignis eintritt, muss dieses deterministisch und im Rahmen eines spezifischen Zeitfensters über diese Schichten hinweg verarbeitet werden. Essentieller Teil des hier vorgestellten Netzwerks ist der Koordinatorknoten, verantwortlich für die Berechnung des gemeinsamen Zielergebnisses. Teilergebnisse werden dabei durch gewöhnliche Knoten berechnet, die eine passende Schnittstelle zur Transition der Betriebszustände und eine Verhaltenstabelle implementieren, über die festgelegt wurde, welcher Algorithmus abhängig vom Betriebszustand zur Berechnung eines Teilergebnisses eingesetzt werden soll.

Definition 8: Koordinatorknoten Ein Knoten innerhalb eines kooperativen Netzwerkes C heißt Koordinatorknoten, wenn er dafür verantwortlich ist, die Transitionen der Betriebsmodi über das Netzwerk zu propagieren. ◇

[4] http://de.wikipedia.org/wiki/OSI-Modell

Für eine verteilte Berechnung des geschlossenen Polygons, im Standardmodus die Berechnung der äußeren Kanten eines Quadrates, muss ein Koordinatorknoten wie folgt arbeiten, wenn einer der vier Prozesse ausfällt:

- Das Ereignis empfangen und verarbeiten
- Eine passende adaptive Transition im Netzwerk propagieren, die durch die einzelnen kooperativen Knoten ausgeführt werden muss. Nach dieser Transition befindet sich das kooperative Netzwerk in einem anderen Betriebsmodus.

Entspricht das zu verarbeitende Ereignis zum Beispiel dem Ausfall eines der Prozesse, die das geschlossenen Polygon berechnen, oder - übertragen auf ein Netzwerk eingebetteter Systeme - dem Ausfall einer Unterkomponente, kann das Ereignis wie folgt durch den Koordinatorknoten ausgewertet werden: Ist die Komponente selbst noch soweit funktionsfähige Daten zu übertragen, wird ein entsprechendes Ereignis direkt durch diesen Netzwerkknoten an den Koordinator gesendet. Dieser Koordinator kann dann das Ereignis verarbeiten und die Transition durchführen. Besteht dagegen die Möglichkeit, dass die Komponenten zur Laufzeit nicht mehr funktionsfähig sind und auch nicht mehr mit dem Koordinatorknoten kommunizieren kann, dann sollte das Netzwerk ein *Watch-Dog* Verhaltensmuster im Koordinator und/oder allen Standardknoten implementieren. Der konkrete Standort der instanziierten Watch-Dogs ist abhängig von der verwendeten Netzwerktopologie. Wenn zum Beispiel ein Standardknoten nur mit dem Koordinator kommuniziert (Sterntopologie), sind Wachhundeprozesse für weitere Standardknoten obsolet. Über konkrete Instanzen der Wachhunde kann nun der Ausfall eines kompletten Netzwerkknotens bemerkt werden und ein entsprechendes Ereignis ausgelöst werden.

Ein Sonderfall ist zu betrachten, wenn der Koordinatorknoten selber ausfällt. In diesem Fall könnte ein die Qualität der netzwerkweiten Berechnung beeinflussendes Ereignis nicht mehr verarbeitet werden. Das ist keine für das Entwurfsmuster Adaptive Controlflow Transitions spezifische Problemstellung. In drahtlosen Sensornetzwerken zum Beispiel müssen ähnliche Problemstellungen gelöst werden. Fällt ein Koordinatorknoten aus, wird eine der noch funktionsfähigen Standardknoten über eine passende Heuristik ausgewählt und als Koordinator dekoriert re-instanziiert. Das ist nur dann möglich, wenn die im Netzwerk verbleibenden Knoten nach dem Ausfall des Koordinators noch miteinander kommunizieren können. In einer Sterntopologie müsste also zumindest die physikalische Schicht einer Komponente noch funktionsfähig sein. Für das Software-basierte Arbeitsbeispiel ist diese Voraussetzung natürlich erfüllt.

Auf logischer Ebene kann das Verhaltensmuster Adaptive Controlflow Transitions von der Sterntopologie auf alle anderen Topologien portiert werden. Allerdings erzeugt das Verhaltensmuster bei anderen Topologien unter Umständen einen großen Mehraufwand,

so dass andere Ansätze besser geeignet wären; zum Beispiel die Selbstorganisation zwecks Koordination wie sie in Sensornetzwerken angewendet wird.

5.3.5. Instanziieren von Betriebsarten

Wenn der Koordinatorknoten die noch funktionsfähigen Standardknoten über die implementierte Schnittstelle zu einer Transition zwingt, müssen diese Knoten die Transition lokal durchführen und über ihre Verhaltenstabelle einen neuen Algorithmus auswählen, der in der folgenden Laufzeit beziehungsweise bis zur einer weiteren Transition eingesetzt wird, um ein Teilergebnis für das Netzwerk bereitzustellen. Dieser Übergang führt entweder zu einer Steigerung der Rechenleistung des jeweiligen Knotens, was eine Neuverteilung der zu leistenden Berechnung über das gesamte Netzwerk propagiert, oder einer Qualitätsreduktion des Knotens, was einer funktionalen Graceful Degradation entspricht. Mit diesem Ansatz verglichen ist das Verhaltensmuster Adaptive Controlflow Transitions kein fundamental neuer Ansatz, bietet aber eine flexiblere Ereignis basierte Lösung. Die dafür benötigten Betriebsmodi werden innerhalb der Verhaltenstabellen der einzelnen Netzwerkknoten zur Entwurfszeit festgelegt.

Definition 9: Verhaltenstabelle Eine Verhaltenstabelle ist eine zwei dimensionale Matrix, in der die Zeilen die verfügbaren Betriebsmodi definieren und die Spalten die konkrete Rechenlast für den jeweiligen Betriebsmodus vorgeben, wenn der entsprechende Algorithmus zur Anwendung kommt. ◇

In Zusammenarbeit mit der Schnittstelle zum Koordinatorknoten wird durch diese Tabelle ein parametrierbarer Moduswechsel möglich. Die einzelnen Einträge in den Matrixzellen definieren eine Reihe von Algorithmen, zum Beispiel Funktionszeiger in der Programmiersprache C.

Die zu erwartende Rechenlast der verwendeten Algorithmen muss zur Entwurfszeit abgeschätzt werden oder dynamisch zuverlässig berechnet werden, um nur effektive Transitionen zu erlauben. Eine dynamische Berechnung ist dabei eine NP-harte Problemstellung. Abhängig von den dedizierten Verhaltenstabellen führen die einzelnen Standardknoten eine Transition in einen neuen Betriebsmodus durch. Danach befindet sich das gesamte Netzwerk in einem anderen Betriebsmodus. Wenn bei der Polygonberechnung zum Beispiel der Prozess für die Kante L_1 ausfällt, könnte die Verhaltenstabelle für die anderen Prozesse wie folgt entworfen sein:

- Der Algorithmus zur Berechnung von Kante L_1 wird zusätzlich durch den Prozess zur Berechnung der Kante von L_2 ausgeführt. Das geht einher mit einer erhöh-

ten Priorität des Prozesses, der nun doppelt so oft ausgeführt werden muss, wie Prozesse, die nur einzelne Kanten berechnen.

- Die Prozesse für die Kanten L_3 und L_4 verfahren wie im Standardbetriebsmodus.

5.3.6. Evaluierung

Dieses Entwurfsmuster wurde prototypisch in zwei Fallstudien und dem Kooperationsprojekt mit einem Unternehmen aus der Avionikindustrie eingesetzt und evaluiert. Im ersten Szenario wurde ein einfacher Ventilator konstruiert, der durch insgesamt drei Elektromotoren angetrieben wird. Die Übersetzung erfolgt über zwei Differenziale. Dieses Szenario entspricht einem Industrieventilator, der eine konstante Kühlleistung erzeugen soll und dessen Einzelkomponenten ohne eine vollständige Betriebsstörung ausgetauscht werden sollen. Die Steuerung der einzelnen Motoren erfolgt über drei separate Prozesse auf einer Mikrocontroller Plattform (ARM-Prozessor mit 48MHz AT91SAM7S256 und 64kB RAM) und eine Koordinatorinstanz, die wie durch das Verhaltensmuster vorgegeben Ausfälle einer der drei Motoren und/oder Controllern registriert und eine Computational Promotion des Netzwerks auslöst. Deswegen fällt die Ventilatorleistung nicht ab, hier als Drehzahl gemessen, wenn einer der Motoren ausfällt. Stattdessen werden die verbliebenen Motoren ihre Ausgangsleistung erhöhen. Das bewirkt über die Umsetzung eine gleichbleibende Ventilatorleistung. Das Verhaltensmuster erlaubt dem Netzwerk auch eine Rückkehr in den Standardbetriebsmodus, damit zum Beispiel alle Motoren gleichmäßig verschleißen, nachdem der Fehler an dem betroffenen Motor behoben werden konnte. Während des Tests des Ventilatormodells wurden die Umdrehungen pro Minute mit Hilfe eines inkrementellen Kodierers (Rotation) gemessen. Die Auswertung der Ergebnisse zeigt, das durch die effiziente Adaption des Netzwerkes, ermöglicht durch das Verhaltensmuster, der Betriebszustand des Ventilators konstant gehalten werden konnte.

Im ersten Szenario wurde das beschriebene Szenario auf das Betriebssystem Linux portiert. Tests auf diesem System zeigten ebenfalls, dass das Verhaltensmuster Adaptive Controlflow Transitions anwendbar ist. Auf Basis der echtzeitfähigen Variante konnten die Ergebnisse noch einmal verbessert werden, da die effizienten Transitionen jetzt auch deterministisch vorhersagbar wurden. Am Ende der prototypischen Evaluierung wurde das Verhaltensmuster auch im Kooperationsprojekt mit der Avionikindustrie eingesetzt, um Betriebsmodi der Softwaresimulation zu wechseln.

5.3.7. Verwandte Arbeiten

Graceful Degradation ist ein wichtiges Entwurfsmuster für sicherheitskritische Systeme [1, 30], insbesondere für fehlertolerante Systeme. Das vorgestellte Verhaltensmuster

Adaptive Controlflow Transitions erweitert diesen statischen Ansatz um ein flexibles Konzept, um dynamisch auf Ereignisse reagieren zu können. Dazu wird dem bekannten Entwurfsmuster die Möglichkeit zugefügt eine Computational Promotion durchzuführen. Das ermöglicht implementierenden Systemen adaptive Transitionen durchzuführen anstatt nur eine Reduktion der Leistung anzuwenden [69]. Das Konzept der Graceful Degradation selber wurde auch optimiert [21]. Dabei erfolgt die Optimierung aus sowohl in Struktur aus auch im Verhalten dezentral. Im Gegensatz zum Verhaltensmuster Adaptive Controlflow Transitions erlaubt dieser Ansatz keine Optimierung des Energieverbrauchs oder anderer Effekte wie den Verschleiß von Komponenten. Ein verwandter Ansatz ist die Rekonfiguration von eingebetteten Systemen [28]. Der Ansatz migriert Rechenaufgaben in ein Netzwerk von rekonfigurierbaren Hardwarekomponenten wie *Programmable Logic Controller*, um auf den Ausfall von Hardwarekomponenten zu reagieren. Im Gegensatz zu [28] fokussiert sich das Verhaltensmuster Adaptive Controlflow Transitions auf Softwarekomponenten, die heutzutage im Bereich eingebetteter Systeme immer wichtiger werden. Nachteil der neuen Verhaltensmuster ist aber, dass der Einsatz der Verhaltenstabelle zur Laufzeit garantiert werden können muss.

5.3.8. Fazit

Das Verhaltensmuster Adaptive Controlflow Transitions liefert eine Softwarearchitektur für verteilte Systeme, die kooperativ ein gemeinsames Ziel bearbeiten. Wird das Verhaltensmuster auf geeigneten Plattformen implementiert, wird aus einem Kontrollfluss ein deterministischer Kontrollfluss über das gesamte Netzwerk. Adaptive Controlflow Transitions bietet die Möglichkeit, entweder das kooperative Netzwerk in einen Betriebsmodus mit reduzierter Qualität der gemeinsam durchgeführten Berechnung oder einen Modus zu versetzen, in dem einzelne Netzwerkknoten eine erhöhte Rechenlast einsetzen, um die Qualität der gemeinsamen Berechnung zu erhalten. Das Verhaltensmuster kann dynamisch angewendet werden, wenn zur Laufzeit innerhalb des kooperativen Netzwerkes zeitlich unvorhersagbare Ereignisse eintreten. Die zur Vorstellung und Evaluierung eingesetzte Sterntopologie des Netzwerkes lässt sich zwar als *Single-Point-of-Failure* interpretieren, was aber in reinen Softwarelösungen vernachlässigbar ist. Zudem ist Architektur skalierbar. Adaptive Controlflow Transitions erlaubt eine dynamische Adaptierung der Netzwerkverhaltens in Abhängigkeit gerade verfügbarer Ressourcen und zu bearbeitender Aufgaben. Jede Transition ist umkehrbar, sobald die konkrete Implementierung dies zur Verfügung stellt; ein Vorteil der mehr oder weniger vor allem für Softwareplattformen gilt. Die Effizienz einer Implementierung des Verhaltensmusters ist abhängig von der zugrunde liegenden Plattform.

Im Rahmen der vorgestellten Arbeit wurde nicht untersucht, ob sich Adaptive Con-

trolflow Transitions auch für prioritätsbasierte Netzwerke eignet und wie gut die Skalierbarkeit für große Netze bezüglich des *protocol stacks*[5] ist. In der jetzige Version wird außerdem keine verschachtelte Adaptierung in separaten Teilen desselben kooperativen Netzwerkes unterstützt, deren Schnittmenge nicht leer ist.

5.4. Resume on Exception

In diesem Abschnitt wird das Verhaltensmuster *Resume on Exception* vorgestellt, das es ermöglicht in beliebigen Schichten einer Softwarearchitektur eine Ausnahmebehandlung durchzuführen und den Kontrollfluss des betroffenen Prozesses in Abhängigkeit des konkreten Szenarios fortzusetzen. Dieses Entwurfsmuster wurde mit Unterstützung von Jan Garcia entwickelt und prototypisch evaluiert. Motivation bei der Entwicklung des Entwurfsmusters ist der Bedarf an einer Ausnahmebehandlung, die einerseits die Effizienz zur Laufzeit erhöht und andererseits die Zuverlässigkeit des Netzwerkes erhöhen kann. Abschnitt 5.4.1 liefert einen Überblick über den aktuellen Stand der Ausnahmebehandlungen. Dann wird in Abschnitt 5.4.2 der Ansatz Resume on Exception motiviert und dann in Abschnitt 5.4.3 das Verhaltensmuster vorgestellt. Die Beurteilung des Verhaltensmusters beginnt in Abschnitt 5.4.4 mit der Darstellung verwandten Entwurfsmuster. Danach wird die Evaluierung des Ansatzes in Abschnitt 5.4.5 und in Abschnitt 5.4.6 abgeschlossen.

5.4.1. Voraussetzungen

Aktuelle Programmiersprachen erlauben Ausnahmebehandlungen durch Blöcke aus den Schlüsselwörtern *try* und *catch*. Tritt in der Schicht einer Softwarearchitektur allerdings eine Ausnahme auf, dann wird der Kontrollfluss des entsprechenden Programms unterbrochen und der Kontrollfluss kann nicht mehr zu dieser Ausführungsstelle zurückkehren. Dieses Problem kann prinzipiell durch exzessiven Einsatz der Ausnahmebehandlung vermieden werden, wenn zum Beispiel jede Programmzeile einen eigenen Ausnahmeblock hat. Das wird in konkreten Systemen aber schnell unübersichtlich und unpraktikabel. Zur Entwurfszeit wird je nach Programmiersprache zusätzlicher Quelltext zum Deallokation lokaler Objektinstanzen generiert. In diesem Fall steigt der Bedarf an Arbeitsspeicher mit jedem *throw* Schlüsselwort, das eine potentielle Ausnahme zur Entwurfszeit markiert.

Abbildung 5.12 zeigt den typischen Ablauf einer Ausnahmebehandlung. Ausgehend vom Programmeinstiegspunkt wird erst Subroutine A, dann Subroutine B und schließlich Subroutine C aufgerufen. Bei der Abarbeitung der Subroutine C tritt dann eine

[5] http://de.wikipedia.org/wiki/Protokollstapel

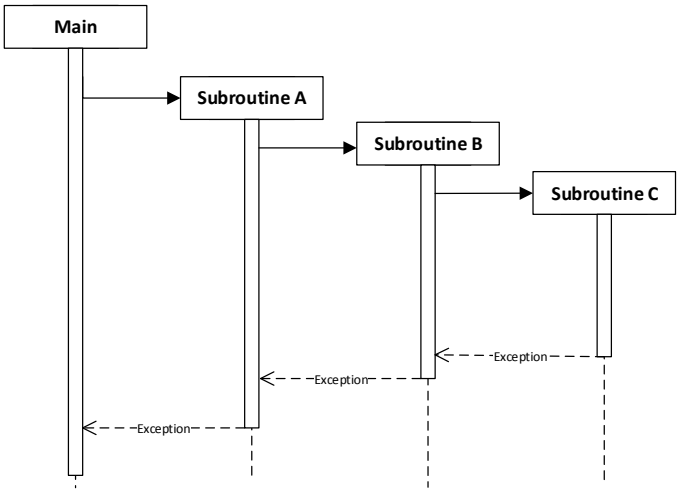


Abbildung 5.12.: Beispielablauf einer Ausnahmebehandlung

Ausnahme auf, die entsprechend behandelt werden muss. An dieser Stelle des Kontrollflusses ist nicht erkennbar, wo diese Ausnahme geschieht, sobald dies nicht mehr lokal innerhalb der Subroutine selber implementiert ist. Im Beispiel wird die Ausnahme wieder bis in die Einstiegsmethoden zurückgeworfen; in allen Subroutinen müssen dabei lokale Objekte alloziert und wieder dealloziert werden. Der geplante Kontrollfluss wird durch die Ausnahme in Subroutine C unterbrochen.

5.4.2. Motivation

Das Verhaltensmuster Resume on Exception soll ermöglichen, dass Ausnahmen in höheren Schichten einer Softwarearchitektur verarbeitet werden können und dann das Programm direkt hinter der Programmstelle fortgeführt werden kann, an der die Ausnahme aufgetreten ist. Die Integration des Entwurfsmusters soll das vorhandene Laufzeitverhalten der Software nicht beeinflussen. Speziell soll es dem Programmierer möglich sein, auf Benutzerlevel die eigentliche Ausnahmebehandlung zur Laufzeit zu konfigurieren. Resume on Exception ermöglicht eine neue Ausnahmebehandlung durch die Integration eines *Exception Handler* Mechanismus, der das Beobachtermuster [19] umsetzt. Nach der entsprechender Adaptierung einer Softwareplattform unterstützt diese Plattform die Instanziierung und Anmeldung von Exception Handlern. Im Rahmen der erweiterten

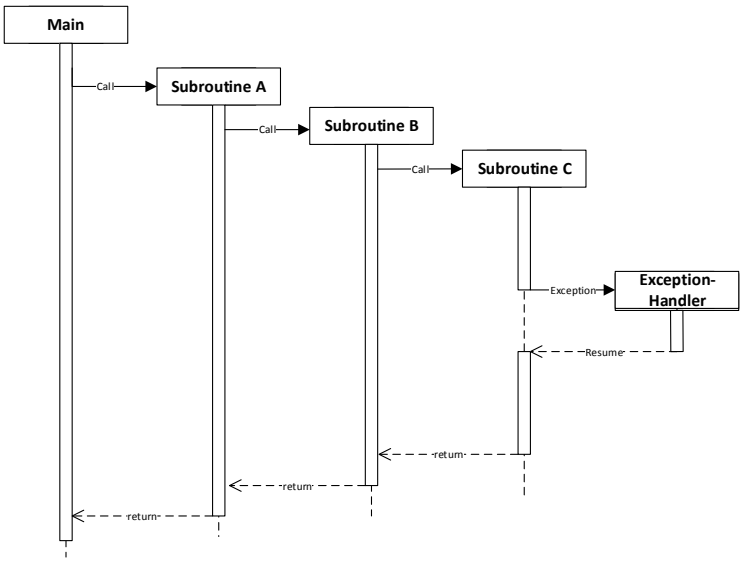


Abbildung 5.13.: Beispielablauf einer Resume on Exception Ausnahmebehandlung

Ausnahmebehandlung werden beim Auslösen jeder Ausnahme alle angemeldeten Handler über die übliche Methode *update* aktualisiert. Dabei werden der Methode für die Manipulation der Ausnahme dedizierte Parameter übergeben. Die Abbildung 5.13 zeigt den potentiellen Ablauf einer Ausnahmebehandlung auf einer adaptieren Softwareplattform. Wie im ersten Beispiel werden die Subroutinen ausgeführt, wobei jetzt auch ein Exception Handler zum Beginn der Laufzeit angemeldet wurde. Tritt nun in Subroutine C eine Ausnahme auf, wird dieser Handler zuerst aufgerufen. Jetzt hat das Programm auf Benutzerlevel die Gelegenheit auf diese Ausnahme passend zu reagieren und anschließend das Programm direkt an der lokalen Stelle innerhalb der Subroutine C auszuführen. Während der Abarbeitung des Exception Handler ist es möglich, die Ausnahme systemweit zu widerrufen und damit die ursprüngliche Abarbeitung zu unterbrechen. Wird diese Möglichkeit nicht eingesetzt, wird nach der Ausführung aller Handler, die Ausnahme wie ursprünglich vorgesehen verarbeitet.

In Abschnitt 5.3 wurde ein Elektrofahrzeug als Beispiel verwendet. Die vier Elektromotoren, welche das Fahrzeug bewegen, werden durch vier einzelne Controller gesteuert. Ein typischer Ablauf ist dabei, dass ein Fahrer dem Fahrzeug mitteilt, dass er beschleunigen möchte. Dazu muss eine hohe Beschleunigung an jedem der vier Motoren realisiert werden. Das Fahrzeug überträgt deshalb den Beschleunigungsbefehl an die vier dedi-

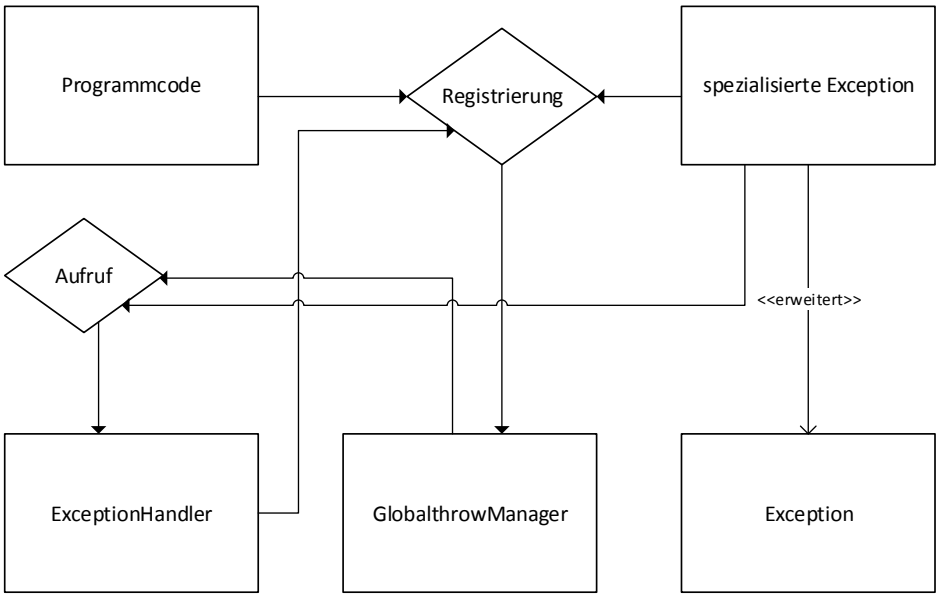


Abbildung 5.14.: Ausnahmebehandlung mit Resume on Exception

zierten Controller. Tritt nun zum Beispiel eine Ausnahme in einem der Controller auf, könnte dieser einzelne Controller die Ausnahme lokal behandeln und wieder einen sicheren Zustand einnehmen. Während der Laufzeit, die der Controller dafür benötigt, die Ausnahme zu behandeln, reagiert der Controller nicht-deterministisch auf weitere zum Beispiel durch den Fahrer verursachte, netzwerkweite Wechsel des Betriebszustandes. Insbesondere, wenn eine Ereignis basierte Datenübertragung implementiert wurde. Durch den Einsatz eines Exception Handler kann diese Ausnahme netzwerkweit behandelt werden. Das bedeutet im Umkehrschluss auch, dass diese Ausnahme netzwerkweit berücksichtigt werden kann, wenn ein Wechsel des Betriebszustandes berechnet wird. Zum Beispiel könnte eine Traktionskontrolle die fehlende negativ Beschleunigung eines Motors für die betroffene Laufzeit berechnen.

5.4.3. Konzept

Das Verhaltensmuster Resume on Exception definiert zwei weitere Teilnehmer in den Laufzeitkontext einer Softwareplattform, das sind *GlobalException* und *GlobalThrowManager*. Dazu muss die verwendete Programmiersprache um das Schlüsselwort `GlobalException` erweitert und der betreffende Kompilierer angepasst werden. Die Syntax es neuen Schlüsselwortes entspricht derjenigen Syntax von `throw`. Zusätzlich wird die

Laufzeitumgebung, zum Beispiel die *Java Runtime*, oder das generierte Maschinenprogramm so angepasst, dass das neue Schlüsselwort durch die einzige Instanz eines `GlobalThrowManager` verarbeitet werden kann. Diese Änderung ist für den Programmierer, der das Verhaltensmuster Resume on Exception für eine Applikation einsetzen will, nicht sichtbar. Das Verwendung des Schlüsselwortes `GlobalThrow` bestimmt, welche Instanzen von *Exception* durch das neue Entwurfsmuster verarbeitet werden sollen. An den `GlobalThrowManager` angemeldete Instanzen des `ExceptionHandler` vervollständigen den Mechanismus.

Abbildung 5.14 zeigt den Ablauf, wenn zur Laufzeit einer Applikation eine global zu verarbeitende Ausnahme geworfen und dann verarbeitet werden soll. Ein Exception Handler wurde implementiert und an den Manager angemeldet. Wird nun eine Ausnahme geworfen, übergibt das Programm diese Instanz von `Exception` an den Manager. Dieser prüft, ob entsprechende Handler angemeldet sind und aktualisiert diese Handler gegebenenfalls. Dabei wird die Instanz der Ausnahme dekoriert und den Handler als Parameter übergeben. Diese spezialisierte Ausnahmeinstanz kann dann in der `update` Methode des Handler eingesetzt werden, um auf Benutzerlevel Einfluss auf die ursprüngliche Ausnahmebehandlung zu nehmen. Der Dekorateur ist eine Instanz von `GlobalException`, welche die Methode `continue` veröffentlicht. Mit Hilfe dieser Methode kann festgelegt werden, an welcher Stelle des Programms der Kontrollfluss nach der Ausnahmebehandlung fortgesetzt wird. Ein potentielles Schema wäre, dass durch ein parameterloses Aufrufen der Methode, der Kontrollfluss direkt nach der Programmzeile, welche die Ausnahme geworfen hat, fortgeführt hat. Ein möglicher Parameter für die Funktion wäre eine Zeichenkette mit dem Namen derjenigen Methode, in welche der Kontrollfluss fortgeführt werden soll.

5.4.4. Verwandte Arbeiten

Veröffentlichungen mit dem Schwerpunkt Ausnahmebehandlungen in Netzwerken und der Problemstellung entsprechend sind schwer zu finden. Der Entwurf dieses Verhaltensmusters orientiert sich aber an den folgenden verwandten Arbeiten.

- Exception handling: issues and a proposed notation [22]
- A Knowledge-based Approach to Handling Exceptions in Workflow Systems [35]
- Exception Handling and Software Fault Tolerance [8]

5.4.5. Evaluierung

Um das neue Verhaltensmuster Resume on Exception zu evaluieren, wurden eine Laufzeitumgebung der Programmiersprache Java adaptiert. Dazu wurde das *OpenJDK 7 (Build b147)*[6] verwendet, deren Quelldateien für die Laufzeitumgebung im Ordner *hotspot* zu finden sind. Die Verwendung einer spezifischen Plattform war nicht gefordert; die neue Laufzeitumgebung wurde auf einem mobilen Gerät mit einem Linux Derivat entwickelt.

Um die Laufzeitumgebung zu adaptieren, wurde zuerst der Kompilierer *javac* um das Schlüsselwort `globalthrow` erweitert und die Semantik des Schlüsselwortes der Laufzeit hinzugefügt. Abbildung 5.15 zeigt eine beispielhafte Anwendung des in die Laufzeit integrierten Verhaltensmusters. Eine Implementierung muss zunächst den entsprechenden Namensraum `java.lang.ext` einbinden. Innerhalb dieses Namensraum werden die neuen Methoden und Klassen veröffentlicht, die das Entwurfsmuster abbilden. Über zum Beispiel die Methode `GlobalThrowManager.registerGlobalThrowHandler()` wird ein neuer Handler angemeldet, der auf globalem Level auf entsprechende Ausnahmen abarbeiten kann. Die einzige Methode dieser Schnittstelle ist `public GlobalThrowContinuePolicy update(Exception ex)`. Wird nun zur Laufzeit eine Ausnahme durch das Schlüsselwort `globalthrow` geworfen, wird für alle angemeldeten Handler diese Methode aufgerufen. Die Implementierung muss außerdem einen Rückgabewert der Enumeration *GlobalThrowContinuePolicy* zurückgeben. Dieser Rückgabewert bestimmt entsprechend der Definition der Methode `continue`, wie der Manager die Ausnahme nach der Aktualisierung der Handler weiter verarbeiten soll. Wurde kein Handler an den Manager angemeldet, entspricht das Laufzeitverhalten von `globalthrow` der ursprünglichen Version `throw`.

Das Verhaltensmuster Resume on Exception wurde vollständig in die Java Laufzeitumgebung integriert. Es wurde gezeigt, dass die Laufzeitumgebung abwärtskompatibel war und Applikationen, entwickelt für die ursprüngliche Laufzeitumgebung, ohne ein verändertes funktionales oder zeitliches Laufzeitverhalten arbeiten.

5.4.6. Fazit

Das Verhaltensmuster Resume on Exception definiert eine neue Ausnahmebehandlung für objektorientierte Programmiersprachen. Die entsprechenden Laufzeitumgebungen oder die generierte Maschinensprache müssen dabei adaptiert werden. Allerdings bleiben die Softwareplattformen dabei abwärtskompatibel und kompatibel Software, die das neue Entwurfsmuster nicht verwenden, zeigen keine Veränderung ihrer Funktion oder

[6] http://www.java.net/download/openjdk

```
import java.lang.ext;

class Evaluation {

  class GlobalException extends Exception {
    public GlobalException(boolean isFatal) {
  this.isFatal = isFatal; }
    public GlobalException(boolean isFatal, String s) {
      super(s);
      this.isFatal = isFatal;
    }
    public boolean isFatal;
  }

  public GlobalThrowExceptionHandler handler = new
  GlobalThrowExceptionHandler {
    @Override
    GlobalThrowContinuePolicy HandleException(Exception ex) {
      GlobalException e = (GlobalException)ex;
      if(e.isFatal) {
  return GlobalThrowContinuePolicy.RETHROW; }
      return GlobalThrowContinuePolicy.CONTINUE;
    }
  };

  public static void Main(String[] args) {
    GlobalThrowManager.
  registerGlobalThrowHandler(GlobalException.class, handler);

    globalthrow new GlobalException(false);

    try { globalthrow new GlobalException(true); }
    catch(GlobalException e) {
      System.out.println("Caught fatal GlobalException"); }
  }
}
```

Abbildung 5.15.: Quelltextbeispiel zur Resume on Exception Evaluierung

des Laufzeitverhaltens. Die Adaptierung der Softwareplattform hat noch einen weiteren Nachteil, dass bei neuen Versionen der Plattform diese wieder um das neue Verhaltensmuster angepasst werden müssen. Dies kann vor allem in kurzlebigen Zyklen, wie zum Beispiel für die *Android* Plattform, einen erheblichen zusätzlichen Aufwand darstellen. Die Vorteile von Resume on Exception liegen einerseits im Potential mit Hilfe dieses Entwurfsmusters zuverlässigeren Quelltext zu implementieren und andererseits in der Möglichkeit, effizienter übersichtlicheren Quelltext zu erzeugen. Das ist wichtig für die Wartung und Weiterentwicklung des Programms; aber auch die Quelltextqualität an sich kann verbessert werden, die insbesondere im Falle von Software für eingebettete Systeme wichtig ist. Betrachtet man die Möglichkeit, über das Entwurfsmuster an beliebigen Stellen einer gegebenenfalls komplexen Software Ausnahmebehandlungen in die Laufzeitkontext zu integrieren, wird die Wahrscheinlichkeit einer inkonsistenten Ausnahmebehandlung erhöht. Allerdings existiert diese Problemstellung auch in einer rein sequenziellen Umsetzung einer Ausnahmebehandlung, sobald die Software mindestens quasi-parallele Ablaufplanungen unterstützt. In beiden Fällen obliegt es Entwurf und Implementierung einer konkreten Software, das Verhaltensmuster genauso korrekt einzusetzen wie die ursprüngliche Ausnahmebehandlung. Das ist dem Entwickler komfortabel möglich, da sich das Verhaltensmuster Resume on Exception an der Definition des Beobachtermusters orientiert.

5.5. Virtual Real-time

In diesem Abschnitt wird das Verhaltensmuster *Virtual Real-time* vorgestellt. Dieses Verhaltensmuster definiert eine Laufzeitumgebung, in der Prozesse mit virtueller Echtzeit ausgeführt werden können. Virtual Real-time wurde in Rahmen eines Kooperationsprojektes mit der Avionikindustrie und dort mit Unterstützung von Tim Ewald und Lukas Armborst entwickelt und evaluiert.

Aufgrund der sicherheitskritischen Szenarien müssen avionische Systeme autark, robust und über lange Zeiträume hinweg arbeiten. Während der Qualitätssicherung werden verschiedene Testmethoden eingesetzt, um diese Systemeigenschaften sicherzustellen. Aber erst in einer der letzten Phasen findet Hardware-in-the-Loop statt, bei der die neu entwickelte Software auf der konkreten Zielhardware ausgeführt wird. Diese Phase ist zeit- sowie kostenaufwendig und findet sehr spät im Entwicklungsprozess statt, obwohl erst hier zeitliche Anforderungen an das System getestet werden können.

Ziel bei der Entwicklung der hier vorgestellten Laufzeitumgebung war die technische Möglichkeit, ein echtzeitfähiges Netzwerk gegenüber einzelner Prozessen zu simulieren. Dabei wird ein deterministisches Laufzeitverhalten der Prozesse und des umgebenen

Netzwerkes den zu testenden Programmen vorgetäuscht. Im Rahmen dieser virtuellen Echtzeit werden dann Ergebnisse wie Speicherbedarf und Zeitbedarf der Prozesse in einer Simulation ermittelt, wobei die dafür benötigte Schnittstelle ebenfalls in diesem Verhaltensmuster definiert ist. Die durch die quasi-echtzeitfähige Ausführung der Prozesse gelieferten Ergebnisse können dann mit geeigneten Metriken so operationalisiert werden, dass Anforderungen an eine reale Hardware abgeleitet werden können. Es handelt sich prinzipiell um einen Software-in-the-Loop Ansatz, der zusätzlich zeitliche Eigenschaften der simulierten Software beurteilbar macht.

Abschnitt 5.5.1 beschreibt das Konzept des neuen Verhaltensmusters und einige Details, die bei einer konkreten Implementierung des Entwurfsmusters berücksichtigt werden müssen. Anschließend wird in Abschnitt 5.5.2 eine Evaluierung vorgestellt. Diese Evaluierung basiert auf der Entwicklung, wie sie innerhalb des Kooperationsprojektes mit der Avionikindustrie durchgeführt wurde. Ein kurzes Fazit in Abschnitt 5.5.3 fasst das neue Verhaltensmuster sowie Vor- und Nachteile zusammen.

5.5.1. Konzept

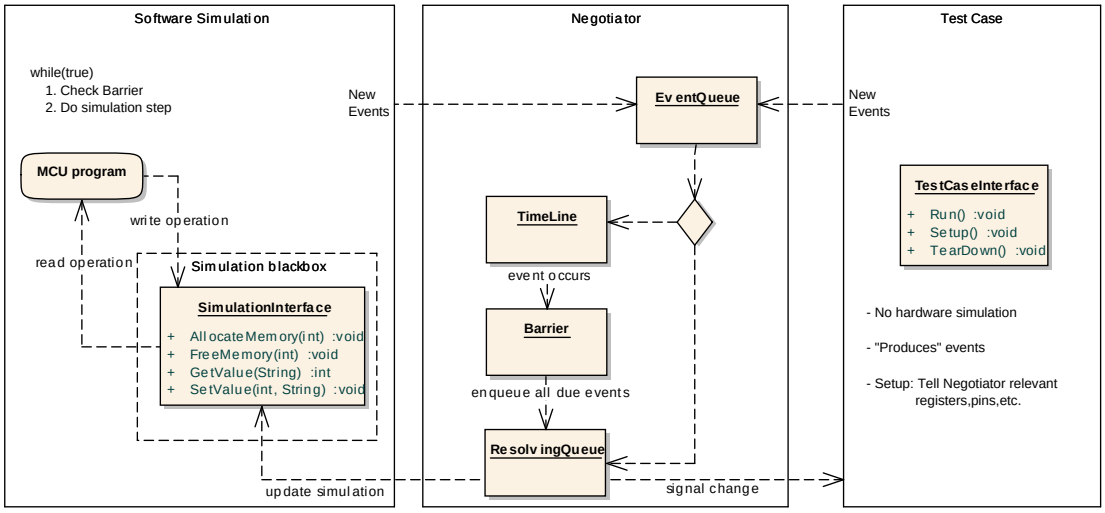


Abbildung 5.16.: Virtuelle Echtzeit im Software-in-the-Loop Kontext

Das Verhaltensmuster Virtual Real-time definiert eine *Sandbox*, in der Prozesse von der zu Grunde liegenden Softwareplattform zeitlich gekapselt werden. Diese Sandbox wird durch eine weitere Softwareschicht realisiert, die einen Teil der Funktionen der Softwa-

replattform dekoriert oder erweitert. Dadurch wird eine neue Programmierschnittstelle veröffentlicht. Verwenden Programme diese Schnittstelle, delegiert die Sandbox die entsprechenden Methodenaufrufe entweder an die Softwareplattform oder verarbeitet diesen Aufruf immanent. Da basierend auf einem handelsüblichen Rechnersystem in beiden Fällen nicht-deterministisches Zeitverhalten auftreten kann, müssen innerhalb der Sandbox alle Ereignisse auf einen virtuellen Zeitstrang abgebildet und bei Bedarf auf einzelne Zeitpunkte dieses Zeitstranges synchronisiert werden.

Definition 10: Virtueller Zeitstrang Eine Warteschlange, in der Ereignisse eingefügt werden können, und ein hinreichend genauer Zähler. Über das Inkrementieren des Zählers wird eine virtuelle Zeit, die insbesondere unabhängig von den Systemzeiten der Softwareplattform ist, realisiert. Die Konfiguration der Granularität des Zählvorgangs sowie ein Anhalten und Fortsetzen dieser virtuellen Zeit ist möglich. Ein durch dieses Inkrement gesteuerter Iterator verarbeitet die Ereignisse aus der Warteschlange. Dabei werden die Ereignisse über eine Ordnungszahl referenziert, die einem Zeitpunkt in der virtuellen Zeit entsprechen. Erreicht der Zählerwert den nächsten in der Warteschlange eingefügten Zeitpunkt, wird dieses Ereignis verarbeitet. Die Signaturen der veröffentlichten Funktionen zum Einfügen neuer Ereignisse in den virtuellen Zeitstrang erzwingen als zusätzlichen Parameter einen Zeitpunkt, der durch das Inkrementieren des Zählers im Moment des Einfügens noch nicht erreicht wurde. ◇

Der virtuelle Zeitstrang ist der Kern des Verhaltensmusters Virtual Real-time, dessen Einsatzmöglichkeiten sich aus dem Funktionsumfang der Programmierschnittstelle ableiten. Dabei ist dieser Funktionsumfang nicht durch das Entwurfsmuster vorgegeben, sondern wird durch konkrete Einsatzszenarien bestimmt. Abbildung 5.16 zeigt deswegen das neue Verhaltensmuster im Rahmen eines Software-in-the-Loop Szenarios, wie es für das Kooperationsprojekt mit der Avionikindustrie spezifiziert wurde. Die Virtual Real-time Softwareschicht besteht aus den drei Komponenten *SoftwareSimulation*, welche die Funktionalität der eigentlichen Simulation und damit auch der hardwarenahen Software kapselt, den *Negotiator* und die *TestCase* Komponenten, welche die Funktionalität eines Testszenarios beinhaltet.

Die zentrale Komponente der neuen Softwareschicht ist der Negotiator. Diese Komponente veröffentlicht die Methoden der Zeitstrang-Implementierung, den Zugriff auf die Ereigniswarteschlange und die Schnittstelle zum Konfigurieren von Zähler und Zeitsynchronisation. Außerdem wird die Kommunikation zwischen der Softwaresimulation und den Testszenarien durch den Negotiator entkoppelt. Beide Komponenten erhalten über den virtuellen Zeitstrang asynchrone Ereignisse von der jeweils anderen Komponente, wobei diese Kommunikation vollständig *Thread*-sicher implementiert werden muss. Bei

der Realisierung des Negotiators sind zwei wesentliche Funktionen umzusetzen: Alle Ereignisse die im Kontext eines virtuellen Echtzeitverhaltens gleichzeitig auftreten, müssen re-synchronisiert abgearbeitet werden. Nicht-gleichzeitig auftretende Ereignisse müssen quasi-synchron geordnet innerhalb der Zeitlinie abgearbeitet werden. Das Verhaltensmuster Virtual Real-time definiert deswegen innerhalb der Negotiatorkomponente eine Zeitbarriere und eine *Resolving Queue*. An der Zeitbarriere werden die asynchron in den Kontext gebrachten Ereignisse synchronisiert, aus dem eigentlichen Zeitstrang entnommen und in der Resolving Queue zwischengespeichert. Während die Ereignisse aus dieser Warteschlange verarbeitet werden, wird der Zähler des virtuellen Zeitstrangs angehalten. Es ist zu beachten, dass durch die Verarbeitung von Ereignissen neue Ereignisse erzeugt werden können, die zum gleichen Zeitpunkt wie andere auslösende aber auch weitere ausgelöste Ereignisse im Rahmen der virtuellen Zeit zum selben Zeitpunkt eintreten sollen. All diese Ereignisse werden in die Resolving Queue eingetragen. Eine Implementierung des Verhaltensmusters muss einen geeigneten Auflösungsmechanismus bereitstellen, um auf diese quasi-gleichzeitigen Ereignisse zu reagieren. Sobald alle Ereignisse innerhalb der neuen Warteschlange abgearbeitet sind, wird der Zähler des virtuellen Zeitstranges wieder geschaltet und die virtuelle Zeit fortgesetzt.

Die Komponente Softwaresimulation veröffentlicht abstrakte Klassen und Methoden, um eine konkrete Simulation im Software-in-the-Loop Kontext zu realisieren. Dadurch wird die Ereignisverwaltung gegenüber dem Negotiator gekapselt. Die Komponente abstrahiert von der Hardware, auf der die zu testende Software eingesetzt werden soll. Das beinhaltet vor allem die Allokation und Deallokation von Speicherbereichen sowie das Manipulieren der Register. Dadurch ist es für den späteren Einsatz möglich Bedingungen über sowohl Speicher als auch die Register innerhalb der Simulation zu spezifizieren. Diese werden zur Laufzeit der Sandbox nach gehalten und im Falle einer Verletzung von Bedingungen entsprechende Informationen aufgezeichnet. Das Verhaltensmuster Virtual Real-time definiert ein `MCUProgram` und eine Schnittstelle, über die eine Hardwaresimulation in den Kontext integriert werden kann. Echtzeitsoftware wird oftmals in der Programmiersprache C/C++ oder zumindest in einer anderen Programmiersprache entwickelt als die Sandbox. Eine Implementierung von `MCUProgram` muss deswegen einen passenden Interpreter für die zu testende Software oder das Kompilat selber realisieren oder einbinden.

Die Komponente `TestCase` veröffentlicht abstrakte Klassen und Methoden, um ein konkretes Testszenario im Software-in-the-Loop Kontext zu realisieren. Im Wesentlichen wird wieder die Ereignisverarbeitung vorgegeben und eine Schnittstelle bereitgestellt, um durch die Implementierung der Methoden `setup`, `run` und `teardown` einen konkreten Testfall umzusetzen. Über eine zyklische Implementierung und der Verwendung der Ereignisverarbeitung, zum Beispiel durch die Implementierung von periodischen Er-

eignissen, lassen sich auch Regelkreise wie in Kapitel 3 beschrieben als Testszenario realisieren.

Abbildung 5.17 zeigt einen Überblick über das Laufzeitverhalten der Sandbox. In einem typischen Software-in-the-Loop Kontext wird die zu testende Software auf einer Simulation der Hardware ausgeführt, auf der diese Software später laufen soll. Die Softwaresimulation entkoppelt diese beiden Programme voneinander, so dass diese für das jeweils zu testende Szenario ausgetauscht werden können. Insbesondere soll es möglich sein, verschiedene Hardware für dasselbe Programm zu konfigurieren und das jeweilige Laufzeitverhalten zu vergleichen. Der Überblick in Abbildung 5.17 beschreibt die folgenden vier Abläufe stellvertretend für das Laufzeitverhalten:

- Während der Initialisierung eines Testszenarios werden durch die Implementierung der `TestCase` Komponente Werte in die Register der simulierten Hardware geschrieben. Dieser Teil der Initialisierung erfolgt nach dem Synchronisieren der Systemzeiten und erfolgt zum Beispiel durch den Einsatz der Ereignisverwaltung des Negotiators.

- Sobald die Hardwaresimulation initialisiert ist, wird das konkrete Programm auf dieser Hardware schrittweise simuliert. Nach jedem Simulationsschritt wird die Sandbox erneut prüfen, ob auf dem virtuellen Zeitstrang ein Ereignis verarbeitet werden muss.

- Die Speicherverwaltung wird durch die Sandbox delegiert. Das Setzen und Lesen von Registern durch das Programm sowie die Allokation und Freigabe von Speichern auf der simulierten Hardware erfolgt über die Schnittstelle der Softwaresimulation. Dadurch können je nach Konfiguration des Testszenarios auch Ereignisse ausgelöst werden.

5.5.2. Evaluierung

Das Verhaltensmuster Virtual Real-time wurde im Rahmen des Kooperationsprojektes mit der Avionikindustrie prototypisch evaluiert. Dabei wurde die Sandbox inklusive aller Schnittstellen wie in Abschnitt 5.5.1 beschrieben umgesetzt. Im Rahmen des Kooperationsprojektes bestand die Testspezifikationen aus einer konkreten Implementierung der abstrakten Klasse `TestCase` und einer Spezifikation, mit deren Hilfe die Laufzeitergebnisse der Sandbox auf Basis dieses Testfalls operationalisiert werden können. Für die Evaluierung wurde ein kleines C Programm entwickelt, das auf einer stark abstrahierten Hardwaresimulation lauffähig ist. Während des Projektes waren die Ressourcen allerdings zu knapp, um für dieses C Programm einen Interpreter umzusetzen. Stattdessen

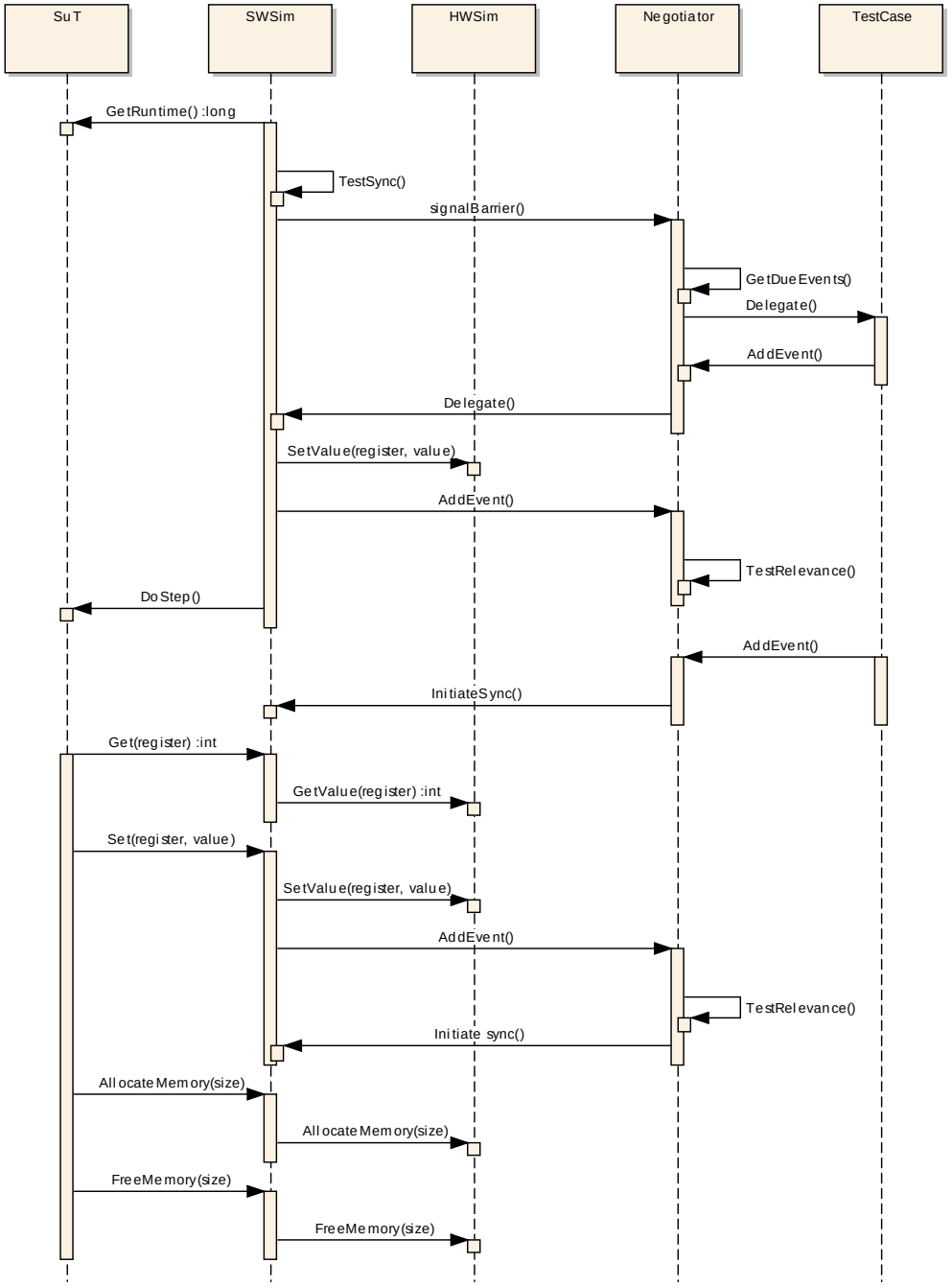


Abbildung 5.17.: Laufzeitverhalten der Sandbox

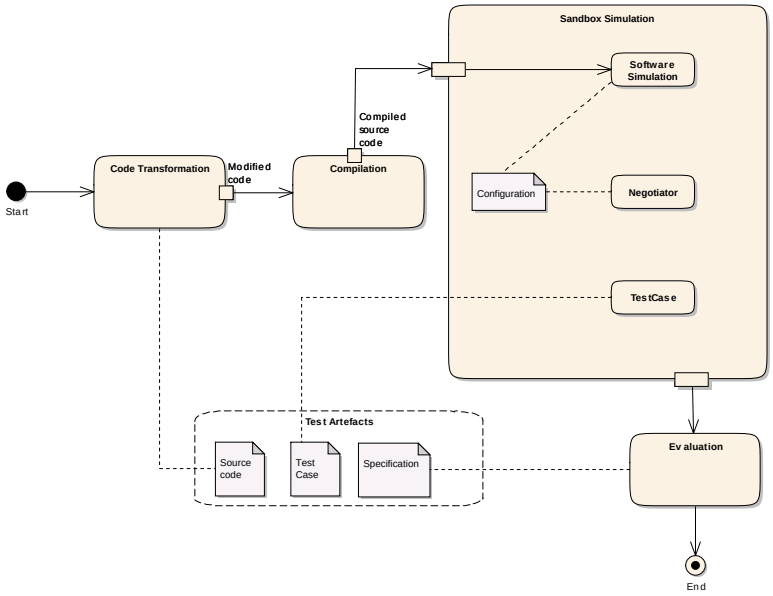


Abbildung 5.18.: Testablauf im Prototypen

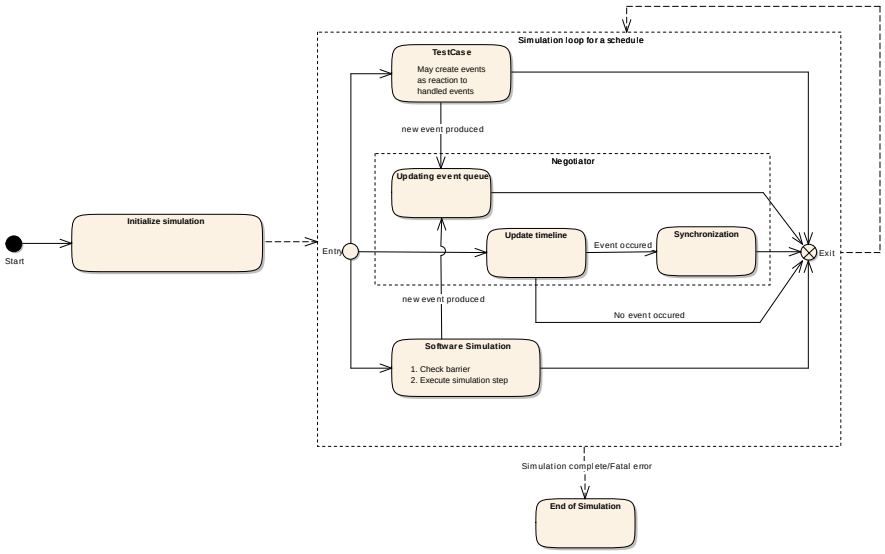


Abbildung 5.19.: Simulationsablauf bezüglich der Zeitlinie

wurde durch Einsatz vorhandener Bibliotheken beziehungsweise einer Werkzeugkette das C Programm...

1. Quelltexttransformiert in ein C# Programm,
2. Quelltext injiziert, um das Programm mit der Schnittstelle zur Softwaresimulation kompatibel zu machen,
3. und das kompatible C# Programm in eine Biblitohek kompiliert, die von der Sandbox ausgeführt wird.

Abbildung 5.18 skizziert den semi-automatischen Ablauf des Prototypen, um einen Testfall abzuarbeiten. Zuerst wird das zu testende Programm transformiert, dann alle benötigten Komponenten neu kompiliert und schließlich ausgeführt. Zur Laufzeit werden alle relevanten Dateninformationen aufgezeichnet und im Anschluss dann manuell ausgewertet.

Zur Laufzeit des Prototypen realisieren Instanzen der Klasse `ASimulation` die durch das Entwurfsmuster geforderte Hardwareabstraktion durch die Bereitstellung der Methoden:

- `Getvalue()` und
- `SetValue()` zum Modifizieren von Registern
- `AllocateMemory()` und
- `FreeMemory()` zur Verwaltung von flüchtigem Speicher

Diese Klasse hält außerdem die Größe des belegten Speichers vor. Die Simulation der zu testenden Software wird durch die Klasse `SoftwareSimulation` mit Hilfe einer Reihe von Callback-Methoden realisiert:

- `SuTMain()`, Initialisierung der Simulation
- `SuTStep()`, Durchführung eines Simulationsschrittes
- `SuTGetRunning()`, asynchrones Handle
- `SuTGetTime()`, liefert die Systemzeit

Zusätzlich wurde in Abbildung 5.19 der Ablauf der Simulation skizziert, diesmal mit dem Fokus auf die Zeitlinie.

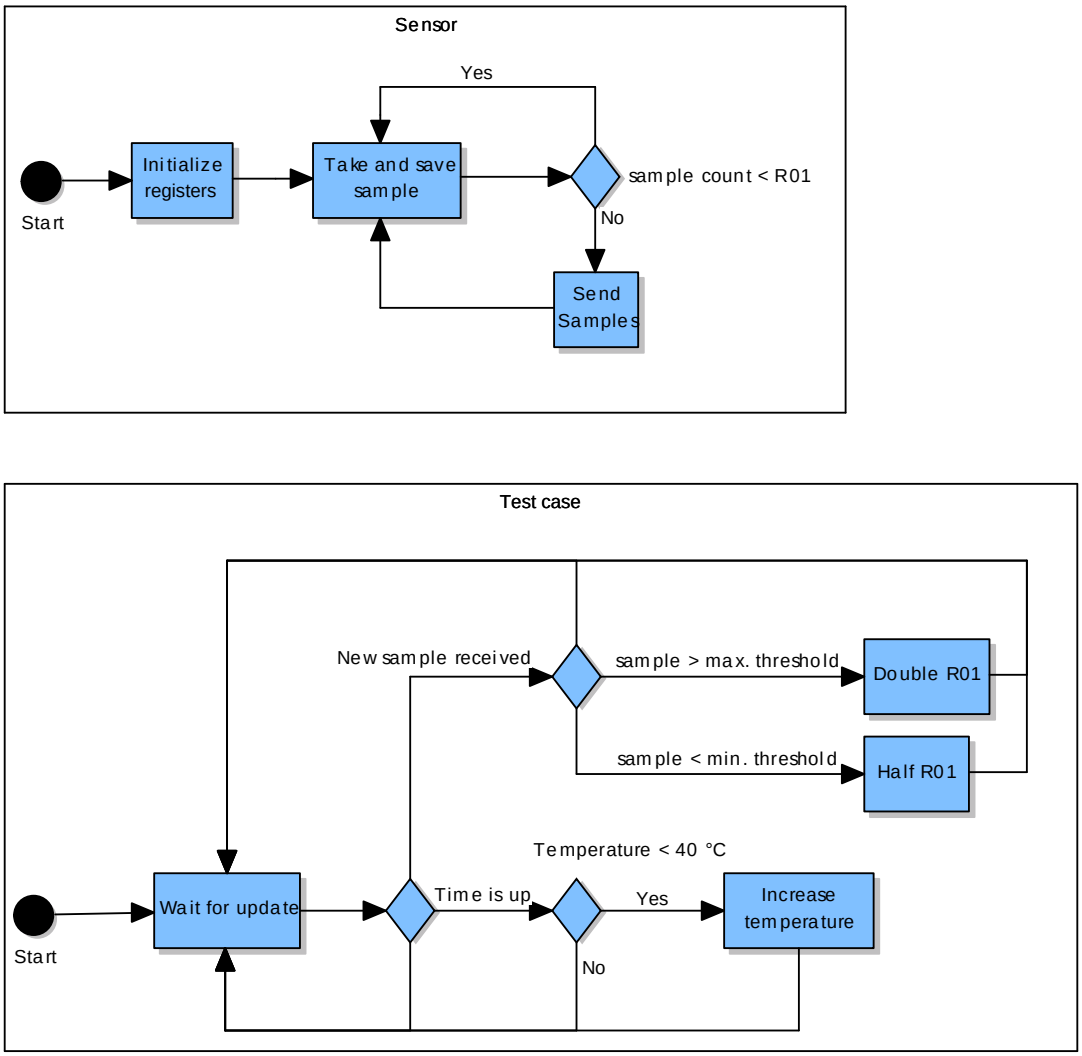


Abbildung 5.20.: Beispiel Software – Sensoren Controller

Der Prototyp wurde mit Hilfe einer schmalen im Projekt entwickelten (virtuellen) Hardware und Software einem einfachen Test unterzogen. Ein Temperatursensor wird durch eine zu testende Software ausgelesen; bei steigenden Werten erhöht diese Software die Frequenz, mit welcher der Sensor ausgelesen wird. Dadurch kann aber das virtuelle Echtzeitsystem die Daten nicht mehr rechtzeitig an einen Monitor, eine Implementierung der `TestCase` Klasse, abgeben. Der Prototyp verarbeitet diesen programmierten Fehler wie erwartet und quittiert den Test als erfolgreich / nicht bestanden. Die Diagramme aus Abbildung 5.20 skizzieren den Testfall sowie das Programm, das getestet wurde. Aus den Testdaten, die in eine Datei geschrieben werden, kann dieser Fehler bezüglich Speicherverbrauch und zeitlichem Verhalten rekonstruiert werden.

Die Abstraktion erlaubt auch eine direkte Kontrolle über zum Beispiel den tatsächlich verbrauchten Speicherbedarf und die Register. Implementierungen der abstrakten Simulation müssen selbstständig für das Laden der entsprechenden Klassen zur Simulation sorgen. Diese Programme haben den Vorteil für die meisten Plattformen nativ und ohne großen Overhead kompiliert werden zu können. Der Prototyp umgeht diese Herausforderung durch eine Konvertierung des Testprogramms in ein C# konformes Model, das dann weiterverarbeitet wird. Dadurch reduziert sich der Aufwand der Simulation auf ein dem Projektumfang angemessenes Maß.

5.5.3. Fazit

Der Prototyp hat gezeigt, dass eine Simulation von Echtzeit möglich ist und prinzipiell dazu eingesetzt werden kann, um zeitkritische Eigenschaften einer Software zu analysieren. Innerhalb des Software-in-the-Loop Kontextes muss sichergestellt werden, dass die Laufzeitergebnisse und die Testspezifikation vergleichbar sind. Dadurch kann je nach Eigenschaft ein großer Aufwand entstehen. Muss eine hohe Genauigkeit simuliert werden, verbraucht die Synchronisation der einzelnen Prozesse proportional mehr Rechenkapazität. Dadurch kann der Ansatz unpraktikabel werden, wenn zum Beispiel 72 Stunden Tests einen Monat dauern. Es liegt allerdings im Konzept der Sandbox, dass innerhalb ihres Rahmens die Zeit langsamer vergeht, als in der realen Umgebung.

Für eine Umsetzung des Entwurfsmusters ist dies eine Herausforderung, weil eine verlässliche Ausführung eines nativen C Programms innerhalb einer simulierten Umgebung schwierig ist. Dieser Faktor muss bei Implementierungen für eine Feldversion der Sandbox eliminiert werden, in dem ein Simulator entwickelt wird, der direkt die zur Anwendung kommende Programmiersprache verarbeitet, hier C.

6. Schluss

Die vorliegende Arbeit zeigt die Problemstellung zeitheterogener Netzwerke: zeitkritische Anforderungen werden propagiert, wenn echtzeitfähige, eingebettete Systeme mit nicht-echtzeitfähigen Systemen vernetzt sind und kommunizieren müssen. Dabei wird das Potential zeit-heterogener Netzwerke am Beispiel der Kooperationsprojekte mit der Automobil, Avionikindustrie sowie der Licht- und Signaltechnik gezeigt. Eine Lösung der so motivierten Problemstellung wird prozessorientiert gegeben, einerseits werden Evaluierungsmethoden zur Analyse der Softwarequalität, andererseits Entwurfsmuster, um auf Basis der Analyse die Software zu adaptieren, vorgestellt. Zum Abschluss der vorliegenden Arbeit werden in diesem Kapitel noch einmal die Evaluierungsmethoden und Entwurfsmuster zusammengefasst, welche eine strukturierte Adaptierung von Software genau solcher nicht-echtzeitfähiger Systeme ermöglichen, um sicherzustellen, dass die spezifische Hauptfunktion der Echtzeitsysteme im Netzwerk weiterhin korrekt ausgeführt wird. Der Abschnitt 6.1 fasst dabei die Ergebnisse der Arbeit zusammen während in Abschnitt 6.2 offene Punkte bezüglich der Problemstellung aufgezeigt werden.

6.1. Zusammenfassung

In Kapitel 4 wird eine auf die Problemstellung zugeschnittene Vorgehensweise zur Evaluierung von Software und Softwareplattformen vorgestellt. Abbildung 4.1 stellt dazu die Vorgehensweise im Überblick dar. Aus dieser Abbildung heraus wird auch die iterative Anwendbarkeit deutlich. Während das Testen, in den Schritten Testauswahl und Testablauf, und im Anschluss die Beurteilung der Testergebnisse an sich generisch verwendet werden kann, sind Operationalisierung und Konformitätsmessung, hier von physikalischen Signalen, auf eingebettete Systeme zugeschnitten und in der Abbildung als Durchführung zusammengefasst. Diese Durchführung ist so entworfen, dass diese Durchführung im Rahmen des Testablaufs verwendet werden kann. In Kapitel 4 wird ebenfalls ein Rahmenwerk zur Testfallgenerierung anhand zeitkritischer Anforderungen vorgestellt, welches unterstützt wird durch im Anhang gegebene Listen von Anforderungen und Anwendungsfällen; ergänzt durch eine Übersicht von Metriken zur Beurteilung von Laufzeitverhalten der zur evaluierenden Software oder Softwareplattformen.

Im Kapitel 5 werden fünf auf die beschriebene Problemstellung zugeschnittene Entwurfsmuster [19] zur Adaptierung von Software beziehungsweise Softwareplattformen vorgestellt. Das Entwurfsmuster *Decoupling Datastructure* entkoppelt Schichten einer Softwarearchitektur voneinander und erlaubt unidirektional echtzeitfähigen Datenzugriff. Das Entwurfsmuster *Active Cross-Layer Balancing* verlagert den Datenzugriff in die Schichten einer Softwarearchitektur und balanciert aktiv den Datendurchsatz zwischen diesen Schichten. Der Datenzugriff wird dabei asynchron verzögert. Das Entwurfsmuster *Adaptive Controlflow Transitions* ermöglicht einen echtzeitfähigen Wechsel des Netzwerkbetriebs auf allen Netzwerkknoten. Insbesondere erfolgt die Freigabe von Ressourcen unmittelbar. Das Entwurfsmuster *Resume on Exception* behandelt Ausnahmen in Netzwerkknoten unter Berücksichtigung des netzwerkweiten Betriebszustands. Die Ausnahmeroutine muss dabei die lokale Ausführung nicht mehr grundsätzlich abbrechen. Das Entwurfsmuster *Virtual Real-time* virtualisiert zeitliches Verhalten. Innerhalb der virtuellen Plattform können Programme echtzeitfähig ausgeführt werden.

Drei Fallstudien wurden durchgeführt und bewerten die vorgestellten Methoden zur Evaluierung von Software und Softwareplattformen. Dabei wurden ein Linux Betriebssystem, eine Unix-basierte Mittelschicht und eine Erweiterung für Microsoft Windows ™ mit Hilfe dieser Methoden auf Echtzeitfähigkeit hin untersucht. Während der Durchführung des Kooperationsprojektes mit einem Unternehmen der Automobilindustrie wurde ein Werkzeug zur Konformitätsüberprüfung physikalischer Signale entwickelt. Die Entwurfsmuster Decoupling Datestructure und Active Cross-Layer Balancing entstanden für eine Softwaresimulation für die Licht- und Signaltechnik. Das Entwurfsmuster Virtual Real-time wurden in einen Software-in-the-Loop Prototypen verbaut. Weiterhin wurde das Betriebssystem *EvalOS* für Mikrocontroller entwickelt, das ein Testrahmenwerk beinhaltet mit dem Benutzerprozesse ausgewertet werden können.

6.2. Ausblick

Der vorgestellte Ansatz, um die Konformität der betroffenen Komponenten schrittweise zu erhöhen, lässt sich in einen konventionellen wie auch agilen Entwicklungsprozess integrieren. Dazu wurde bewusst ein iterativer Ansatz gewählt. Motivation ist vor allem einer steigenden Komplexität der Systeme und damit einhergehenden Komplexität der Entwicklung sowie deren Kosten entgegen wirken zu können. Gerade bei eingebetteten Systeme kann die Effektivität des Entwicklungsprozesses einen zentralen Wettbewerbsfaktor ausmachen. Eine entsprechende Fallstudie über die Anwendbarkeit des Ansatzes ist aber nicht mehr Teil der vorgelegten Arbeit. Die Bewertung der Evaluierungsmethoden und Entwurfsmuster zeigt im Rahmen der Forschungsprojekte und Fallstudien,

dass die Problemstellung gelöst werden kann. Damit ist aber noch nicht nachgewiesen, dass der Konformitätsgrad der resultierenden Software ausreichend für zum Beispiel eine Anwendung in sicherheitskritischen Domänen ist. Diese Aspekte der praktischen Anwendung sollten in einem realen industriellen Entwicklungsszenario mit Hilfe weiterer Forschung untersucht werden.

Die in der vorliegenden Arbeit vorgestellten Entwurfsmuster wurden jeweils für spezifische Problemstellungen im Rahmen der Kooperationsprojekte mit der Industrie entwickelt. Dabei sind diese fünf Entwurfsmuster als Verhaltensmuster [19] definiert worden, bedingt durch die Problemstellung, Laufzeitverhalten gemäß zeitkritischer Anforderungen zu adaptieren. Dieser Katalog kann deswegen noch erweitert werden: Einerseits, um allgemein neue Entwurfsmuster für weitere spezifische Probleme zu definieren. Andererseits, um Entwurfsmuster zu ergänzen, die Strukturen modellieren. Parallel zur dieser Definition der Vorgehensweisen und Entwurfsmuster kann dann ein regressiver Weg definiert und implementiert werden, Echtzeitfähigkeit bezüglich der Entwurfsmuster nachzuweisen und bezüglich des Entwicklungsprozesses wiederverwendbar zu machen. Diese Implementierung sollte nicht mehr unabhängig im Rahmen einzelner Projekte stattfinden, sondern als Paket funktionieren.

Literaturverzeichnis

[1] Avizienis, A. : Fault-tolerance: The survival attribute of digital systems. In: *Proceedings of the IEEE* 66 (1978), Nr. 10, S. 1109–1125. – ISSN 0018–9219

[2] Beer, G. : *Konformitätsbewertung - Begriffe und allgemeine Grundlagen.* Beuth Verlag, 2005 (Redaktionelle Beiträge)

[3] Berger, A. S.: *Ebedded Systems Design: An Introduction to Processes, Tools and Techniques.* CMP Books, 2001. – ISBN 1–57820–073–3

[4] Bültmann, M. : *Tauglichkeitsstudie von Betriebssystemen für den Einsatz in zeitkritischen Prozessen.* Germany, RWTH Aachen Universität, Bachelorarbeit, 2009

[5] Born, M. ; Favaro, J. ; Kath, O. : Application of ISO DIS 26262 in practice. In: *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety.* New York, NY, USA : ACM, 2010 (CARS '10). – ISBN 978–1–60558–915–2, S. 3–6

[6] Broekman, B. ; Notenboom, E. : *Testing Embedded Software.* Addison-Wesley Professional, 2002. – ISBN 0321159861

[7] Cormen, T. H. ; Leiserson, C. E. ; Rivest, R. L. ; Stein, C. : *Introduction to Algorithms, Second Edition.* The MIT Press, 2001. – 273–301 S. – ISBN 0262032937

[8] Cristian, F. : Exception Handling and Software Fault Tolerance. In: *Computers, IEEE Transactions on* C-31 (1982), Nr. 6, S. 531–540. `http://dx.doi.org/10.1109/TC.1982.1676035`. – DOI 10.1109/TC.1982.1676035. – ISSN 0018–9340

[9] Daintith, J. (Hrsg.) ; Wright, E. (Hrsg.): *Oxford Dictionary of Computing.* Oxford University Press, 2008. – ISBN 978–0–19–923400–4

[10] Dechev, D. ; Pirkelbauer, P. ; Stroustrup, B. : Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs. In: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, 2010. – ISSN 1555–0885, S. 185–192

[11] Douglass, B. P.: *Doing Hard Time.* Addison-Wesley Professional, 1999. – ISBN 0–201–49837–5

[12] Douglass, B. P.: *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002

[13] Draper, N. R. ; Smith, H. : *Applied Regression Analysis (Wiley Series in Probability and Statistics).* Wiley-Interscience, 1998. – ISBN 0471170828

[14] Dustin, E. ; Rashka, J. ; Paul, J. : *Software automatisch testen: Verfahren, Handhabung und Leistung (Xpert.press) (German Edition).* Springer, 2000. – ISBN 3540676392

[15] Franke, D. : *Verifikation der Java-Echtzeitfähigkeit für den Einsatz in zeitkritischen Systemen.* Germany, RWTH Aachen Universität, Diplomarbeit, 2009

[16] Franke, D. ; Kowalewski, S. ; Weise, C. : A Mobile Software Quality Model. In: *12th International Conference on Quality Software (QSIC)*, IEEE Computer Society, 2012. – ISBN 978–1–4673–2857–9, S. 154 – 157

[17] Franke, D. ; Schommer, J. ; Kowalewski, S. : Softwarequalität in der Mobilen Welt: Testen von Lebenszyklen Mobiler Applikationen. In: *Embedded Software Engineering Kongress (ESE)*, ELEKTRONIKPRAXIS, 2012. – ISBN 978–3–8343–2407–8, S. 478 – 482

[18] Friske, M. ; Schlingloff, H. : Von Use Cases zu Test Cases: Eine systematische Vorgehensweise. In: Klein, T. (Hrsg.) ; Rumpe, B. (Hrsg.) ; Schätz, B. (Hrsg.) ; TU Braunschweig (Veranst.): *Tagungsband Dagstuhl-Workshop MBEES: Model Based Engineering of Embedded Systems* TU Braunschweig, 2005 (Informatik-Bericht 2005-01), S. 1–10

[19] Gamma, E. : *Entwurfsmuster . Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, 1996. – ISBN 3827318629

[20] Gerlitz, T. ; Kalkov, I. ; Schommer, J. ; Franke, D. ; Kowalewski, S. : Non-Blocking Garbage Collection for Real-Time Android. In: *11th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES).* ACM (ACM Digital Library), 108-117

[21] Glass, M. ; Lukasiewycz, M. ; Haubelt, C. ; Teich, J. : Incorporating graceful degradation into embedded system design. In: *DATE*, IEEE, 2009, S. 320–323

[22] Goodenough, J. B.: Exception Handling: Issues and a Proposed Notation. In: *Commun. ACM* 18 (1975), Dez., Nr. 12, S. 683–696. – ISSN 0001–0782

[23] Gosling, J. ; Bollella, G. ; Dibble, P. ; Furr, S. ; Turnbull, M. : *The Real-Time Specification for Java*. Addison Wesley Longman, 2000. – ISBN 0201703238

[24] Gottlieb, A. ; Lubachevsky, B. D. ; Rudolph, L. : Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. In: *ACM Trans. Program. Lang. Syst.* 5 (1983), Apr., Nr. 2, S. 164–189. – ISSN 0164–0925

[25] Hänsch, P. ; Schommer, J. F. ; Kowalewski, S. : Self-balancing Controllable Robots in Education: A Practical Course for Bachelor Students. In: *Intelligent Robotics and Applications (ICIRA 2011)*, Springer, 2011. – ISBN 978–3–642–25488–8, S. 297–306

[26] Haritsa, J. R. ; Carey, M. J. ; Livny, M. : On being optimistic about real-time constraints. In: *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM (PODS '90). – ISBN 0–89791–352–3, 331–343

[27] Haritsa, J. R. ; Carey, M. J. ; Livny, M. : On being optimistic about real-time constraints. In: *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* ACM, 1990, S. 331–343

[28] Haubelt, C. ; Koch, D. ; Reimann, F. ; Streichert, T. ; Teich, J. : ReCoNets Design Methodology for Embedded Systems Consisting of Small Networks of Reconfigurable Nodes and Connections. In: Platzner, M. (Hrsg.) ; Teich, J. (Hrsg.) ; Wehn, N. (Hrsg.): *Dynamically Reconfigurable Systems*. Springer Netherlands, 2010, S. 223–243

[29] Heinsohn, J. M. ; Kostrzewa, T. ; Hieronymus, M. : Einführung in die ISO 26262 "Functional Safety - Road Vehicles". Version: 2011. `http://hdl.handle.net/10419/67089`. Nordakademie (2011-07). – Arbeitspapiere der Nordakademie. – Working Paper

[30] Henzinger, T. ; Sifakis, J. : The Embedded Systems Design Challenge. Version: 2006. `http://dx.doi.org/10.1007/11813040_1`. In: Misra, J. (Hrsg.) ; Nipkow, T. (Hrsg.) ; Sekerinski, E. (Hrsg.): *FM 2006: Formal Methods* Bd. 4085. Springer Berlin Heidelberg. – ISBN 978–3–540–37215–8, 1-15

[31] Kalkov, I. : *Design and Integration of Real-Time into the Android Platform*, RWTH Aachen Universität, Masterarbeit, 2011

[32] Kalkov, I. ; Franke, D. ; Schommer, J. F. ; Kowalewski, S. : A Real-time Extension to the Android Platform. In: *10th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, ACM, 2012. – ISBN 978–1–4503–1688–0, S. 105 – 114

[33] Kamsties, E. ; Pohl, K. ; Reis, S. ; Reuys, A. : Testing Variabilities in Use Case Models. In: Linden, F. J. (Hrsg.): *Software Product-Family Engineering* Bd. 3014. Springer Berlin Heidelberg, 2004, S. 6–18

[34] Köhler, B.-U. : *Konzepte der statistischen Signalverarbeitung (German Edition)*. Springer, 2005. – ISBN 3540234918

[35] Klein, M. ; Dellarocas, C. : A Knowledge-based Approach to Handling Exceptions in Workflow Systems. In: *Computer Supported Cooperative Work (CSCW)* 9 (2000), Nr. 3-4, S. 399–412. – ISSN 0925–9724

[36] Kopetz, H. : *Real-time systems: design principles for distributed embedded applications*. Springer, 2011

[37] Kowalewski, S. ; Rumpe, B. ; Stollenwerk, A. : Cyber-Physical Systems - eine Herausforderung an die Automatisierungstechnik? In: Wissensforum, V. (Hrsg.): *Automation 2012*, VDI-Verlag, 2012 (VDI Berichte). – ISBN 978–3–18–092171–6, S. 113–116

[38] Kristian, K. ; Rigoll, G. ; Schuller, B. W.: *Statistische Informationstechnik: Signal - und Mustererkennung, Parameter- und Signalschätzung (German Edition)*. Springer, 2011. – ISBN 3642159532

[39] Lange, T. F.: *Entwicklung eines Implementierungsentwurfs für den Prototypen einer Softwaresimulation*. Germany, RWTH Aachen Universität, Bachelorarbeit, 2011

[40] Lee, E. : Cyber Physical Systems: Design Challenges. In: *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, 2008, S. 363–369

[41] Lewis, D. w.: *Fundamentals of Embedded Software: Where C and Assembly Meet*. Prentice Hall, 2001. – ISBN 0–13–061589–7

[42] Linux: *Kernel Referenz*. `http://linux.die.net/man/2/getrusage`, 2014 (zuletzt aufgerufen am 28.April 2016)

[43] Love, R. M.: *Linux Kernel Handbuch*. Addison Wesley Verlag, 2005. – ISBN 3827322049

[44] Lu, C. ; Blum, B. M. ; Abdelzaher, T. F. ; Stankovic, J. A. ; He, T. : RAP: A Real-Time Communication Architecture for Large-Scale Wireless Sensor Networks. In: *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 02)*, 2002, S. 55 – 66

[45] Lunze, J. : *Regelungstechnik 1: Systemtheoretische Grundlagen, Analyse und Entwurf einschleifiger Regelungen (Springer-Lehrbuch) (German Edition)*. Springer, 2004. – ISBN 3540207422

[46] Mellor-Crummey, J. : *Concurrent Queues: Practical Fetch-and- Algorithms*. Department of Computer Science, University of Rochester (Technical report). `http://books.google.de/books?id=tOkEtwAACAAJ`

[47] Michael, M. M. ; Scott, M. L.: Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA : ACM, 1996 (PODC '96). – ISBN 0–89791–800–2, S. 267–275

[48] Michael, M. M. ; Scott, M. L.: Nonblocking Algorithms and Preemption-safe Locking on Multiprogrammed Shared Memory Multiprocessors. In: *J. Parallel Distrib. Comput.* 51 (1998), Mai, Nr. 1, S. 1–26. – ISSN 0743–7315

[49] Michelson, B. M.: *Event-Driven Architecture Overview*. Boston, MA, USA : Patricia Seybold Group, 2006

[50] OSADL: *Realtime Linux Preemption Patch*. `https://www.osadl.org`, 2014 (zuletzt aufgerufen am 28.April 2016)

[51] Palczynski, J. ; Weise, C. ; Kowalewski, S. : Testing Continuous Systems Conformance Using Cross Correlation. In: *Proceedings of the 22nd IFIP International Conference on Testing Software and Systems: Short Papers*. Montréal : CRIM, 2010. – ISBN 978–2–89522–136–4, S. 31–36

[52] Palczynski, J. ; Weise, C. ; Kowalewski, S. ; Ulmer, D. : Estimation of Clock Drift in HiL Testing by Property-Based Conformance Check. In: *Fourth International Conference on Software Testing, Verification and Validation Workshops*

(ICSTW, TAIC PART) 2011, IEEE Computer Society, 2011. – ISBN 978–1–4577–0019–4, S. 590 – 595

[53] Palczynski, J. ; Weise, C. ; Moj, S. ; Kowalewski, S. : Comparing Continuous Behaviour in Model-based Development of Embedded Software. In: *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII (MBEES 2011)*, fortiss GmbH, 2011, S. 61 – 70

[54] Pichler, R. : *Scrum - Agiles Projektmanagement erfolgreich einsetzen.* Dpunkt.Verlag GmbH, 2007. – ISBN 3898644782

[55] Salewski, F. : *Empirical Evaluations of Safety-Critical Embedded Systems*, Embedded Software Laboratory - RWTH Aachen University, Diss., 2008. `http://aib.informatik.rwth-aachen.de`. – AIB-2008-18

[56] Sasnauskas, R. ; Dustmann, O. S. ; Kaminski, B. L. ; Wehrle, K. ; Weise, C. ; Kowalewski, S. : Scalable Symbolic Execution of Distributed Systems. In: *Proceedings of the 2011 31st International Conference on Distributed Computing Systems.* Washington, DC, USA : IEEE Computer Society, 2011 (ICDCS '11). – ISBN 978–0–7695–4364–2, S. 333–342

[57] Sasnauskas, R. ; Landsiedel, O. ; Alizai, M. H. ; Weise, C. ; Kowalewski, S. ; Wehrle, K. : KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment. In: *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2010)*, 2010, S. 186 – 196

[58] Schäuffele, J. ; Zurawka, T. : *Automotive Software Engineering.* Vieweg Verlag, 2003 (ATZ-MTZ-Fachbuch). – ISBN 3–528–01040–1

[59] Schmithausen, D. : *Evaluierung des Ressourcen-Schedulings in zeit-heterogenen Systemen.* Germany, RWTH Aachen Universität, Bachelorarbeit, 2011

[60] Schnellbach, A. : Betriebsbewährtheit nach der ISO 26262-8, Kapitel 14. 2. In: *Osnabrücker Forum Funktionale Sicherheit, Osnabrück* Bd. 11, 2011, S. 299—-308

[61] Scholz, P. : *Softwareentwicklung Eingebetteter Systeme.* Physica-Verlag, 2005. – ISBN 9783540275220

[62] Schommer, J. ; Gerlitz, T. ; Kowalewski, S. : Load and Quality Cooperation for Distributed Embedded Systems Using Different Modes of Operation. In: *7th Junior Researcher Workshop on Real-Time Computing*, 45-48

[63] Schommer, J. F. ; Franke, D. ; Kowalewski, S. ; Weise, C. : Evaluation of the Real-Time Java Runtime Environment for Deployment in Time-Critical Systems. In: *7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, ACM, 2009. – ISBN 978–1–60558–732–5, S. 51–60

[64] Schommer, J. F. ; Franke, D. ; Lange, T. ; Kowalewski, S. : Load Balancing for Cross Layer Communication. In: *36th Computer Software and Applications Conference Workshops (COMPSACW)*, IEEE Computer Society, 2012. – ISBN 978–1–4673–2714–5, S. 476–481

[65] Schulz, K. P.: *Stichwörter zur Europäischen Normung.* Beuth, 2002. – ISBN 3410152695

[66] Seidl, R. : *Der Systemtest.* Hanser Fachbuchverlag, 2006. – ISBN 3446407936

[67] Sha, L. ; Gopalakrishnan, S. ; Liu, X. ; Wang, Q. : Cyber-Physical Systems: A New Frontier. Version: 2009. `http://dx.doi.org/10.1007/978-0-387-88735-7_1`. In: *Machine Learning in Cyber Trust.* Springer US. – ISBN 978–0–387–88734–0, 3-13

[68] Shakkottai, S. ; Karlsson, P. C.: Cross-Layer Design for Wireless Networks. In: *IEEE Communications Magazine* 41 (2003), S. 74–80

[69] Shin, K. ; Meissner, C. : Adaptation and graceful degradation of control system performance by task reallocation and period adjustment. In: *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, 1999. – ISSN 1068–3070, S. 29–36

[70] Smith, S. : *Digital Signal Processing: A Practical Guide for Engineers and Scientists.* Newnes, 2002. – ISBN 075067444X

[71] Srivastava, V. ; Motani, M. : Cross-layer design: a survey and the road ahead. In: *Communications Magazine, IEEE* 43 (2005), Nr. 12, S. 112–119. – ISSN 0163–6804

[72] Stankovic, J. A.: Misconceptions about real-time computing: A serious problem for next-generation systems. In: *Computer* 21 (1988), Nr. 10, S. 10–19

[73] Steinbach, T. ; Korf, F. ; Schmidt, T. C.: Simulationsbasierte Evaluierung von Metriken in Echtzeit-Ethernet basierten Fahrzeugnetzen. In: Wolfinger, B. E. (Hrsg.) ; Heidtmann, K.-D. (Hrsg.): *6ter GI/ITG-Workshop Leistungs-, Zuverlaessigkeits- und Verlaesslichkeitsbewertung von Kommunikationsnetzen und*

verteilten Systeme (MMBnet 2011). Hamburg : Universität Hamburg, 2011, S. 9–20

[74] Stewart, D. B.: Measuring Execution Time and Real-Time Performance. In: *In: Proceedings of the Embedded Systems Conference (ESC SF*, 2002, S. 1–15

[75] Stonebraker, M. ; Çetintemel, U. ; Zdonik, S. : The 8 requirements of real-time stream processing. In: *ACM SIGMOD Record* 34 (2005), Nr. 4, S. 42–47

[76] Thönnessen, D. : *Steuerung einer Fertigungsanlage mit RTAndroid.* Germany, RWTH Aachen Universität, Bachelorarbeit, 2012

[77] Tsigas, P. ; Zhang, Y. : A Simple, Fast and Scalable Non-blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. In: *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures.* New York, NY, USA : ACM, 2001 (SPAA '01). – ISBN 1–58113–409–6, S. 134–143

[78] Valois, J. D.: Implementing Lock-Free Queues. In: *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV*, 1994, S. 64–69

[79] Valois, J. D.: Lock-Free Linked Lists Using Compare-and-Swap. In: *In Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, 1995, S. 214–222

[80] Wolf, W.-G. B. H.: *Agile Softwareentwicklung.* Dpunkt.Verlag GmbH, 2010. – ISBN 3898647013

[81] Wong, C. S. ; Tan, I. K. T. ; Kumari, R. D. ; Lam, J. ; Fun, W. : Fairness and interactive performance of O(1) and CFS Linux kernel schedulers. In: *Information Technology, 2008. ITSim 2008. International Symposium on* Bd. 4, 2008, S. 1–8

[82] Wörn, H. ; Brinkschulte, U. : *Echtzeitsysteme.* Springer-Verlag, 2005 (eXamen.press). – ISBN 3–540–20588–8

[83] Zurawski, R. (Hrsg.): *Networked Embedded Systems (Industrial Information Technology.* CRC Press, 2009. – ISBN 978–1–4398–0761–3

A. Anforderungskatalog

In diesem Kapitel werden die für die Problemstellung relevanten zeitkritischen Anforderungen katalogisiert. Der Katalog ordnet jeder Anforderung mindestens eine Kategorie aus den folgenden sechs Kategorien zu: Kommunikation zwischen einzelnen System- oder Softwarekomponenten, Datenverarbeitung über die Eingabe bis zur Ausgabe der Daten und die Kategorie Betriebssystem, welche wiederum aus den Kategorien Scheduling, Speicherverwaltung und Zeitgeber besteht. Dabei muss ein Echtzeitsystem nicht zwingend ein vollständiges Betriebssystem einsetzen oder realisieren. Die Angabe der Kategorie Betriebssystem bedeutet in diesem Kontext dann, dass die katalogisierte Anforderung in allen drei Unterkategorien dieser Kategorie berücksichtigt werden muss. Zusätzlich werden die Abhängigkeiten der Anforderungen untereinander katalogisiert. Einerseits werden die Eigenschaften der Software oder Softwareplattform genannt, die Voraussetzungen der Erfüllbarkeit der katalogisierten Anforderung sind. Andererseits werden Anforderungen genannt, für deren Erfüllbarkeit die effektive Umsetzung der katalogisierten Anforderung benötigt wird. Dadurch entsteht ein Abhängigkeitsgraph, der in Abbildung 4.2 aus Seite 40 abgebildet ist.

A.1. Echtzeit

Kategorien	Betriebssystem, Datenverarbeitung, Kommunikation
Voraussetzung	Zeitliche Vorhersagbarkeit, Determinismus, Zuverlässigkeit
Ermöglicht	*Echtzeitfähigkeit des Systems beziehungsweise der Software*
Referenzen	[3, 4, 11, 61, 82]

Echtzeit oder Echtzeitfähigkeit ist eine nicht-funktionale Anforderung. Ein System beziehungsweise eine Software ist dann echtzeitfähig, wenn in einer kalkulierbaren Frist eine Eingabe verarbeitet und ein Ergebnis ausgeben wird. Das System oder die Software ist zu jedem Moment der Laufzeit betriebsbereit. Die entsprechende First zur Berechnung der Ausgabe ist entweder zufällig verteilt, läuft periodisch oder zu vorherbestimmten Zeitpunkten ab. Je nach Entwurf des Systems oder der Software wird das Echtzeitverhalten durch unterschiedliche Methoden realisiert.

A.2. Determinismus

Kategorien	Betriebssystem
Voraussetzung	Effizienz, Speicherallokation, Determinismus bezüglich einer Ausgabe
Ermöglicht	Echtzeit
Referenzen	[4, 15, 82]

Alle Berechnungen eines Echtzeitsystems und seiner Software müssen in vorhersagbarer Zeit beziehungsweise innerhalb eines vorhersagbaren Zeitintervalls ausgeführt werden. Determinismus kann probabilistisch definiert sein, wenn Heuristiken bezüglich der Antwortzeiten im Echtzeitsystem verwendet werden. Determinismus resultiert aus der Reaktivität auf externe Ereignisse und der Verfügbarkeit der benötigten Kapazitäten.

A.3. Zuverlässigkeit

Kategorien	Betriebssystem
Voraussetzung	Watchdog, Persistenz-freie Datenverarbeitung, Fehlerhafte Eingabedaten, Datenspeicherung, Datenintegrität, Zustandsüberwachung, Zeitige Fehlererkennung, Kommunikation, Wiederverwendbarkeit, Unabhängigkeit
Ermöglicht	Echtzeit
Referenzen	[4, 72]

Ein Echtzeitsystem muss unter Berücksichtigung zeitkritischer Anforderungen zuverlässig arbeiten. Zuverlässigkeit ist eine sicherheitskritischen Anforderung.

A.4. Zeitliche Vorhersagbarkeit

Kategorien	Betriebssystem
Voraussetzung	Rechtzeitigkeit, Verfügbarkeit, Reaktivität
Ermöglicht	Echtzeit
Referenzen	[4, 82]

Zeitliche Vorhersagbarkeit ist eine Konsequenz aus Rechtzeitigkeit und Verfügbarkeit. Die Verarbeitungszeit innerhalb eines Echtzeitsystems ist genau dann vorhersagbar, wenn einerseits die benötigten Ressourcen zur Verfügung stehen und auf diesen Ressourcen eine rechtzeitige Berechnung durchführbar ist.

A.5. Gleichzeitigkeit

Kategorien	Betriebssystem
Voraussetzung	Koordination und Ablauf
Ermöglicht	Reaktivität
Referenzen	[4, 15, 82]

Gleichzeitige Eingaben müssen gleichzeitig verarbeitet werden. Für jede resultierende Aufgabe des Echtzeitsystems müssen alle zeitkritischen Anforderungen erfüllt sein.

A.6. Rechtzeitigkeit

Kategorien	Betriebssystem
Voraussetzung	Anpassung (Zeitgeber), Deadlock, Watchdog, Speicherschutz Interrupt (Scheduler), Echtzeitwarteschlange
Ermöglicht	Effizienz, Zeitliche Vorhersagbarkeit
Referenzen	[4, 82]

Die Ausgaben eines Echtzeitsystems, basierend auf entsprechenden Eingaben in das System, müssen innerhalb einer vorgegebenen Antwortzeit beziehungsweise Frist erfolgen.

A.7. Reaktivität

Kategorien	Betriebssystem
Voraussetzung	Gleichzeitigkeit, Interrupt (Zeitgeber)
Ermöglicht	Effizienz, Zeitliche Vorhersagbarkeit
Referenzen	[4, 20, 82]

Auf nicht-kalkulierte Ereignisse muss ein Echtzeitsystem innerhalb einer vorgegebenen Antwortzeit reagieren. Dabei sind geplante sowie verschachtelte Ereignisse zu beachten.

A.8. Koordination und Ablauf

Kategorien	Scheduling
Voraussetzung	Synchronisation
Ermöglicht	Gleichzeitigkeit
Referenzen	[4, 20, 82]

Der Scheduler teilt die verfügbare Prozessorzeit derart zu, dass alle Aufgaben in einer vorgegebenen Zeit und innerhalb eines vorgegebenen Zeitintervalls ablaufen können.

A.9. Verwaltung der Ressourcen

Kategorien	Scheduling
Voraussetzung	-
Ermöglicht	Synchronisation, Deadlock
Referenzen	[4, 20, 82]

Verfügbare Ressourcen müssen den einzelnen Aufgaben derart zugeteilt werden, dass alle Aufgaben in einer vorgegebenen Zeit und innerhalb eines vorgegebenen Zeitintervalls ablaufen können. Insbesondere müssen Zugriffe auf Ressourcen in der richtigen Reihenfolge und separat erteilt werden, wenn mehrere Prozesse auf eine Ressource zugreifen.

A.10. Deadlock

Kategorien	Scheduling
Voraussetzung	Verwaltung der Ressourcen
Ermöglicht	Synchronisation, Rechtzeitigkeit, Verfügbarkeit, Echtzeitwarteschlange
Referenzen	[4, 15, 82]

Der Scheduler muss dafür sorgen, dass Prozesse sich durch die Verwendung von entsprechenden Ressourcen nicht gegenseitig blockieren oder aushungern. Prozesse blockieren sich, wenn mindestens zwei Prozesse jeweils auf die Freigabe einer Ressource durch den jeweils anderen Prozess warten. Prozesse hungern aus, wenn Ressourcen durch hoch priorisierte Prozesse quasi-permanent blockiert werden.

A.11. Interrupt (Scheduler)

Kategorien	Scheduling
Voraussetzung	-
Ermöglicht	Rechtzeitigkeit
Referenzen	[4, 82]

Das Echtzeitsystem muss externe und interne ungeplante Ereignisse effektiv verarbeiten können. Dabei unterbricht das ungeplante Ereignis die Ausführung einer geplanten Aufgabe, die nach der Verarbeitung dieses Interrupts fortgesetzt werden muss.

A.12. Verfügbarkeit

Kategorien	Betriebssystem
Voraussetzung	Kommunikation, Watchdog, Deadlock, Speicherschutz, Speicherallokation
Ermöglicht	Zeitliche Vorhersagbarkeit, Effizienz
Referenzen	[4, 15, 82]

Alle für eine Eingabeverarbeitung und anschließender Ergebnisausgabe benötigten Ressourcen stehen während der Laufzeit des Systems zu jedem Zeitpunkt zur Verfügung.

A.13. Speicherallokation

Kategorien	Speicherverwaltung
Voraussetzung	Verschiebbarkeit
Ermöglicht	Determinismus, Verfügbarkeit
Referenzen	[4, 20, 82]

Die Speicherverwaltung eines Echtzeitsystems muss den von Prozessen angeforderten Speicher innerhalb vorhersagbarer Zeitintervallen zuteilen.

A.14. Gemeinsamer Speicher

Kategorien	Speicherverwaltung
Voraussetzung	-
Ermöglicht	Synchronisation
Referenzen	[4, 15, 82]

Unter Berücksichtigung zeitkritischer Anforderungen einzelner Prozesse, können diese über gemeinsam genutzten Speicher eine Kommunikation realisieren.

A.15. Interrupt (Zeitgeber)

Kategorien	Zeitgeber
Voraussetzung	Anpassung (Zeitgeber)
Ermöglicht	Reaktivität
Referenzen	[4, 82]

Um entsprechend zeitkritischer Anforderungen Ereignisse zu verarbeiten, muss die zeitliche Granularität der Verarbeitung präzise und ausreichend effizient sein.

A.16. Speicherbereinigung

Kategorien	Speicherverwaltung
Voraussetzung	-
Ermöglicht	Verschiebbarkeit
Referenzen	[4, 15, 20, 82]

Das Echtzeitsystem muss eine Speicherbereinigung unter der Berücksichtigung zeitkritischer Anforderungen durchführen, wenn durch dynamische Vorgänge im System insgesamt nicht ausreichend Speicher für den Ablauf aller Prozesse zur Verfügung steht. Diese kann auch inkrementell erfolgen, um zeitkritische Eigenschaften des Systems nicht zu gefährden.

A.17. Datenintegrität

Kategorien	Datenverarbeitung
Voraussetzung	-
Ermöglicht	Zuverlässigkeit
Referenzen	[75]

Speicher ein Echtzeitsystem Daten, muss zu jedem Moment der Laufzeit den Datenintegrität gewährleistet werden. Bei persistenter Datenspeicherung muss die Integrität auch über die Laufzeiten hinweg gewährleistet sein. Ein Systemausfall darf nicht in Dateninkonsistenz resultieren.

A.18. Effizienz

Kategorien	Betriebssystem
Voraussetzung	Verfügbarkeit, Reaktivität, Rechtzeitigkeit, Bedienbarkeit
Ermöglicht	Determinismus
Referenzen	[4, 20, 82]

Ein Echtzeitsystem muss Kapazitätsanforderungen einhalten, dazu muss das Verbrauchsverhalten des Systems berechenbar sein. Ein Echtzeitsystem muss Zeitanforderungen einhalten. Dazu muss vor der Laufzeit des Systems berechenbar sein, wie schnell das System auf externe Ereignisse reagieren muss, welche Antwortzeiten eingehalten werden müssen und wie häufig und wie verteilt diese Ereignisse auftreten.

A.19. Verschiebbarkeit

Kategorien	Speicherverwaltung
Voraussetzung	Speicherbereinigung
Ermöglicht	Speicherallokation
Referenzen	[4, 15, 20, 82]

Dedizierter Speicher einem einzelnen Prozess zugeordnet muss im Speicher des Echtzeitsystems verschoben werden können. Diesen Vorgang durch entsprechende Konfiguration unterdrückbar zu implementieren ist Teil der Anforderung.

A.20. Zeitgeber

Kategorien	Zeitgeber
Voraussetzung	-
Ermöglicht	Watchdog, Anpassung (Zeitgeber)
Referenzen	[4, 82]

Das Echtzeitsystem benötigt einen präzisen Zeitgeber. Dieser Zeitgeber bestimmt die zeitliche Granularität, mit der Anwendungen auf Ereignisse reagieren können.

A.21. Anpassung (Zeitgeber)

Kategorien	Zeitgeber
Voraussetzung	Zeitgeber
Ermöglicht	Rechtzeitigkeit, Interrupt (Zeitgeber)
Referenzen	[4, 82]

Der Zeitgeber muss konfigurierbar sein.

A.22. Watchdog

Kategorien	Zeitgeber
Voraussetzung	Zeitgeber
Ermöglicht	Rechtzeitigkeit, Verfügbarkeit, Zuverlässigkeit
Referenzen	[4, 82]

Das Echtzeitsystem überwacht die ablaufenden Prozesse bezüglich ihrer Antwortzeiten und Reaktivität.

A.23. Echtzeitwarteschlange

Kategorien	Speicherverwaltung
Voraussetzung	Deadlock
Ermöglicht	Rechtzeitigkeit
Referenzen	[4, 15, 82]

Das Echtzeitsystem stellt für die Kommunikation einzelner Prozesse untereinander eine Echtzeitwarteschlange bereit. Dadurch wird eine gepufferte, mindestens unidirektionale Kommunikation mit nicht-echtzeitfähigen Prozessen möglich.

A.24. Wiederverwendbarkeit

Kategorien	Betriebssystem
Voraussetzung	-
Ermöglicht	Zuverlässigkeit
Referenzen	[72]

Komponenten eines Echtzeitsystems müssen modular entworfen sein, um wiederverwendet werden zu können.

A.25. Interaktion

Kategorien	Betriebssystem
Voraussetzung	-
Ermöglicht	Bedienbarkeit
Referenzen	[72]

Das Echtzeitsystem muss zur Interaktion mit dem technischen Kontext oder der Umwelt entsprechende Sensoren und Aktoren bereitstellen.

A.26. Priorisiertes Blocken

Kategorien	Scheduling
Voraussetzung	-
Ermöglicht	Verfügbarkeit
Referenzen	[27]

Die Ablaufplanung des Echtzeitsystems unterstützt Prioritäten.

A.27. Bedienbarkeit

Kategorien	Betriebssystem
Voraussetzung	Interaktion
Ermöglicht	Effizienz
Referenzen	[4, 82]

Das Echtzeitsystem muss für etwaige Benutzer bedienbar sein. Diese Bedienung muss erlernbar sein und umfasst die feingranulare Kontrolle des Benutzers über die Abläufe innerhalb des Echtzeitsystems und gegebenenfalls der Kommunikation.

A.28. Synchronisation

Kategorien	Scheduling
Voraussetzung	Verwaltung der Ressourcen, Deadlock, Gemeinsamer Speicher
Ermöglicht	Koordination und Ablauf
Referenzen	[4, 82]

Durch die Verteilung der Ressourcen und die Vermeidung von Deadlocks kann synchronisiert werden. Das Echtzeitsystem muss den Zugriff mehrerer Prozesse auf einzelne Ressourcen koordinieren.

A.29. Persistenz-freie Datenverarbeitung

Kategorien	Datenverarbeitung
Voraussetzung	-
Ermöglicht	Zuverlässigkeit
Referenzen	[20, 75]

Das Echtzeitsystem muss große Datenmengen in kurzer Zeit verarbeiten ohne die Daten auf persistente Speichermedien zu verschieben.

A.30. Robustheit Gegen Fehlerhafte Eingabedaten

Kategorien	Datenverarbeitung
Voraussetzung	-
Ermöglicht	Zuverlässigkeit
Referenzen	[75]

Das Echtzeitsystem muss unvollständige, fehlerhafte Eingabedaten robust tolerieren.

A.31. Speicherschutz

Kategorien	Speicherverwaltung
Voraussetzung	-
Ermöglicht	Rechtzeitigkeit, Verfügbarkeit
Referenzen	[4, 15, 82]

Das Echtzeitsystem stellt sicher, dass nur dedizierte Speicherbereiche einzelnen Prozessen zur Verfügung stehen. Insbesondere der Speicher anderer Prozesse oder eines Betriebssystems müssen vor Zugriffen geschützt werden.

A.32. Determinismus bezüglich der Ausgabe

Kategorien	Datenverarbeitung
Voraussetzung	-
Ermöglicht	Determinismus
Referenzen	[75]

Das Echtzeitsystem muss für eine identische Eingabe immer die gleiche Ausgabe liefern, solange sich das System ebenfalls in einem identischen Zustand befindet.

A.33. Datenspeicherung

Kategorien	Datenverarbeitung
Voraussetzung	-
Ermöglicht	Zuverlässigkeit
Referenzen	[75]

Das Echtzeitsystem unterstützt die persistente Speicherung von Daten.

A.34. Zustandsüberwachung

Kategorien	Kommunikation
Voraussetzung	-
Ermöglicht	Zuverlässigkeit
Referenzen	[36]

Alle Komponenten eines Echtzeitsystems oder vernetzte Echtzeitsysteme informieren sich gegenseitig über die Systemzustände. Dabei wird ein temporäres Zustandsabbild oder Teil eines Abbildes der Komponenten oder Systeme überwacht.

A.35. Zeitige Fehlererkennung

Kategorien	Kommunikation
Voraussetzung	-
Ermöglicht	Zuverlässigkeit
Referenzen	[36]

Potentielle Systemfehler müssen mit einer hohen Wahrscheinlichkeit durch das Echtzeitsystem erkannt werden. Fehlererkennung und gegebenenfalls eine Korrektur müssen den zeitkritischen Anforderungen an das System genügen.

A.36. Kommunikation

Kategorien	Kommunikation
Voraussetzung	-
Ermöglicht	Zuverlässigkeit, Verfügbarkeit
Referenzen	[72]

Wenn Echtzeitsysteme kommunizieren, muss diese Kommunikation zeitkritischen Anforderungen genügen. Es muss die Datenintegrität garantiert werden.

A.37. Unabhängigkeit

Kategorien	Kommunikation
Voraussetzung	-
Ermöglicht	Zuverlässigkeit
Referenzen	[11]

Einzelne Komponenten des Echtzeitsystems arbeiten weitestgehend unabhängig voneinander. Dadurch soll die Propagation funktionaler Fehler aber auch zeitkritischer Probleme auf das gesamte System reduziert oder ganz vermieden werden.

B. Anwendungsszenarien

In diesem Kapitel werden die für die Problemstellung relevanten zeitkritischen Anwendungsszenarien katalogisiert. Diese Szenarien werden den Unterkategorien von Betriebssystem, Scheduling, Speicherverwaltung und Zeitgeber, zugeordnet, wie diese in Kapitel A beschriebenen sind. Andere Kategorien wie Kommunikation oder Datenverarbeitung wurden im Rahmen dieser Arbeit nicht weiter vertieft.

B.1. Überblick

Dieser Abschnitt ist eine tabellarische Zusammenfassung der folgenden Anwendungsszenarien, sortiert nach Kategorie.

B.1.1. Scheduling

Koordination/Ablauf

- Zuteilung von Rechenzeit
- Zuteilung von Rechenzeit und Betriebsmitteln
- Versuch der Zuteilung von Rechenzeit (Nicht-Einhaltung-Deadline)

Synchronisation - Verwaltung von Betriebsmitteln → Deadlock - Livelock

- Synchronisierter Zugriff auf Betriebsmittel
  - Zuteilung Rechenzeit/Betriebsmittel (Synchronisation-Nicht-Einhaltung der Deadline)
- Synchronisation - Einhaltung der Deadline und Zugriffsüberwachung (Deadlock)
- Synchronisation - Deadlockbehandlung

Interruptbehandlung

- Interrupt tritt auf
- Interrupt tritt auf und Nicht-Einhaltung der Deadline

B.1.2. Speicherverwaltung

Zuteilung von Speicher an Tasks

- erfolgreiche Zuteilung von Speicher unter Einhaltung der Deadline
- erfolglose Zuteilung von Speicher unter Verletzung der Deadline

Koordination gemeinsamer Speicher

- Zuteilung gemeinsamer Speicher
- Koordination (Synchronisation)
- Rechtzeitigkeit der Koordination gemeinsamer Speicher

Kommunikation

- Kommunikation von Echtzeittasks und normalen Tasks

Schutz des Speichers (Abgrenzung der Tasks gegeneinander)

- Zugriffsschutz

Verschiebbarkeit der Tasks im Hauptspeicher

Speicherbereinigung

- Speicherbereinigung ohne Reorganisationspause
- Inkrementelle Speicherbereinigung

B.1.3. Zeitgeber

Zeitgeber

- Zeitgeber gibt Taktrate für Anwenderprozesse vor
- Zeitgeber gibt Taktrate für Scheduling vor
- Zeitgeber gibt Taktrate für Interrupts vor

Anpassen Frequenz

- Benutzer passt Taktfrequenz des Zeitgebers an

Watchdog

- Watchdog überwacht zeitkritische Anwendung (Timeout)

Interruptbehandlung

B.2. Anwendungsszenarien Scheduling

Dieser Abschnitt beschreibt die Anwendungsszenarien, welche der Kategorie Scheduling zugeordnet sind.

B.2.1. Koordination/Ablauf

Name	**Koordination/Ablauf → Zuteilung Rechenzeit**
Ziel	Scheduling der Tasks unter Beachtung der Echtzeit-Bedingungen, das heißt, Tasks werden rechtzeitig gestartet und vor ihrer Deadline beendet.
(Level)	Scheduling
Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt • (Bekanntheit der aktuellen Prozessorauslastung)
Nachbedingung	Verteilung der Rechenzeit auf die Prozesse unter Einhaltung der Deadlines
Nachbedingung beim Fehlerauftritt	*Hart:* Verteilung der Rechenzeit ohne Einhaltung der Deadlines und Benachrichtigung an Benutzer *Soft:* Verteilung der Rechenzeit ohne Einhaltung der Deadlines und weiteres Scheduling
Aktoren	Tasks, Scheduler, Timer
Auslöser (Trigger)	Tasks sind im System vorhanden und benötigen Rechenzeit
Standard-Sequenz	• Timer gibt Taktrate vor für den Scheduler • Prozesse, welche Rechenzeit benötigen, sind im System und kommen bei Scheduler an • Prozesse laufen in zugeteilter Reihenfolge – Prozesse halten Deadlines ein
	Fortsetzung auf der nächsten Seite

Alternativen / Ausnahmen	• – Prozesse halten Deadlines nicht ein. Scheduler gibt Meldung an den Benutzer und führt zum Prozess passende Fehlerbehandlung durch

Name	**Koordination/Ablauf - Zuteilung Rechenzeit und Betriebsmittel**
Ziel	Scheduling der Tasks unter Beachtung der Echtzeit-Bedingungen, das heißt, Tasks werden rechtzeitig gestartet und vor ihrer Deadline beendet. Die Tasks erhalten die Betriebsmittel, die sie benötigen, innerhalb der zeitlichen Grenzen.
(Level)	Scheduling
Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt • (Bekanntheit der aktuellen Prozessorauslastung) • verschiedene Betriebsmittel, die von den Prozessen genutzt werden können
Nachbedingung	Verteilung der Rechenzeit und der Betriebsmittel auf die Prozesse unter Einhaltung der Deadlines.
Nachbedingung beim Fehlerauftritt	*Hart:* Verteilung der Rechenzeit und Betriebsmittel auf die Prozesse ohne Einhaltung der Deadlines und Benachrichtigung an Benutzer *Soft:* Verteilung der Rechenzeit und Betriebsmittel ohne Einhaltung der Deadlines und weiteres Scheduling
Aktoren	Tasks, Scheduler, Timer, Betriebsmittel
Auslöser (Trigger)	Tasks sind im System vorhanden und benötigen Rechenzeit und Betriebsmittel
	Fortsetzung auf der nächsten Seite

Standard-Sequenz	• Timer gibt Taktrate für den Scheduler vor • Prozesse, welche Rechenzeit und Betriebsmittel benötigen, sind im System und kommen bei Scheduler an • Prozesse laufen in zugeteilter Reihenfolge • Betriebsmittel werden den Prozessen nach Anforderung zugeteilt – Prozesse halten Deadlines ein
Alternativen / Ausnahmen	• – Prozesse halten Deadlines nicht ein. Scheduler gibt Meldung an den Benutzer und führt zum Prozess passende Fehlerbehandlung durch

B.2.2. Synchronisation

Name	**Synchronisation - Verwaltung (Synchronisation) von Betriebsmittel**
Ziel	Scheduling der Tasks unter Beachtung der Echtzeit-Bedingungen, das heißt, Tasks werden rechtzeitig gestartet und vor ihrer Deadline beendet. Die Tasks erhalten die Betriebsmittel, die sie benötigen, innerhalb der zeitlichen Grenzen, und der Zugriff auf die Betriebsmittel wird synchronisiert.
(Level)	Scheduling
	Fortsetzung auf der nächsten Seite

Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und-Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt • (Bekanntheit der aktuellen Prozessorauslastung) • verschiedene Betriebsmittel, die von den Prozessen genutzt werden können
Nachbedingung	Verteilung der Rechenzeit und der Betriebsmittel auf die Prozesse unter Einhaltung der Deadlines. Die Betriebsmittel konnten dabei so auf die Prozesse verteilt werden, dass alle Echtzeitbedingungen eingehalten wurden.
Nachbedingung beim Fehlerauftritt	*Hart:* Verteilung der Rechenzeit und Betriebsmittel auf die Prozesse ohne Einhaltung der Deadlines und Benachrichtigung an Benutzer *Soft:* Verteilung der Rechenzeit und Betriebsmittel ohne Einhaltung der Deadlines und weiteres Scheduling
Aktoren	Tasks, Scheduler, Timer, Betriebsmittel
Auslöser (Trigger)	Tasks sind im System vorhanden und benötigen Rechenzeit und Betriebsmittel
Standard-Sequenz	• Timer gibt Taktrate vor für den Scheduler • Prozesse, welche Rechenzeit und Betriebsmittel benötigen, sind im System und kommen bei Scheduler an • Prozesse laufen in zugeteilter Reihenfolge • Betriebsmittel werden den Prozessen nach Anforderung zugeteilt – Prozesse halten Deadlines ein
	Fortsetzung auf der nächsten Seite

Alternativen / Ausnahmen	• – Prozesse halten Deadlines nicht ein. Scheduler gibt Meldung an den Benutzer und führt zum Prozess passende Fehlerbehandlung durch

Name	**Synchronisation - Deadlock**
Ziel	Mehrere Prozesse können ein und dasselbe Betriebsmittel benötigen. Der Zugriff auf die Betriebsmittel (z. B. Daten, Geräte, Treiber) erfolgt derart, dass nur eine Anwendung gleichzeitig auf das Betriebsmittel zugreift und währenddessen jedoch die Echtzeiteigenschaften (Rechtzeitigkeit, zeitliche Vorhersagbarkeit, Determinismus) eingehalten werden
(Level)	Scheduling
Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt • (Bekanntheit der aktuellen Prozessorauslastung) • Betriebsmittel bzw. geteilter Speicher, worauf mindestens zwei Prozesse gleichzeitig zugreifen wollen
Nachbedingung	Beide Prozesse konnten unter Synchronisation auf das Betriebsmittel zugreifen und ihre Aufgabe beenden. Dies geschieht unter Einhaltung der Rechtzeitigkeit.
	Fortsetzung auf der nächsten Seite

Nachbedingung beim Fehlerauftritt	1. Beide Prozesse konnten unter Synchronisation auf das Betriebsmittel zugreifen und ihre Aufgaben beenden. Jedoch konnten sie dabei nicht die Rechtzeitigkeit einhalten. 2. Beide Prozesse konnten auf das Betriebsmittel zugreifen, jedoch unterlief bei der Synchronisation ein Fehler, so dass Inkonsistenzen entstehen.
Aktoren	Prozesse, Betriebsmittel, Scheduler, Timer
Auslöser (Trigger)	Mehrere Prozesse wollen auf ein Betriebsmittel zugreifen
Standard-Sequenz	• Timer gibt Taktrate vor für den Scheduler • Prozesse, welche Rechenzeit und Betriebsmittel benötigen, sind im System und kommen bei Scheduler an • Scheduler vergibt Rechenzeit und verteilt Betriebsmittel • Prozesse arbeiten in vorgegebener Reihenfolge • Scheduler synchronisiert den Zugriff auf die Betriebsmittel • Prozesse arbeiten mit der zugeteilten Rechenzeit und den zugeteilten Betriebsmitteln
	Fortsetzung auf der nächsten Seite

Alternativen / Ausnahmen	1. Scheduler vergibt Rechenzeit und verteilt Betriebsmittel: • Prozess A erhält Rechenzeit und Betriebsmittel 1 • Prozess B erhält Rechenzeit und wartet auf BM 1, da Prozess A höhere Priorität • Deadline von Prozess B wird nicht eingehalten, Verletzung der Rechtzeitigkeit 2. Scheduler vergibt Rechenzeit und verteilt Betriebsmittel • Prozess A erhält Rechenzeit und Betriebsmittel 1 • Prozess A behält Zugriff auf Betriebsmittel 1 • Prozess B erhält Rechenzeit und Zugriff auf Betriebsmittel 1 (Veränderungen von Prozess A wurden noch nicht gespeichert) • Prozess A erhält wieder Rechenzeit und beendet Zugriff auf BM1 und speichert seine Ergebnis in BM 1. • Prozess B erhält Rechenzeit und beendet seinen Zugriff auf BM1 und speichert ebenfalls seine Ergebnisse. – Inkonsistenz der Daten, da ohne Berücksichtigung von Veränderungen von Prozess A berechnet wurde.

Name	**Deadlockbehandlung**
Ziel	Mehrere Prozesse können ein und dasselbe Betriebsmittel benötigen. Der Zugriff auf die Betriebsmittel (z. B. Daten, Geräte, Treiber) erfolgt derart, dass nur eine Anwendung gleichzeitig auf das Betriebsmittel zugreift und währenddessen jedoch die Echtzeiteigenschaften (Rechtzeitigkeit, zeitliche Vorhersagbarkeit, Determinismus) eingehalten werden. Außerdem darf es nicht dazu kommen, dass zwei Prozesse gleichzeitig aufeinander warten (Deadlock).
	Fortsetzung auf der nächsten Seite

(Level)	Scheduling
Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor der Rechenzeit zur Verfügung stellt • (Bekanntheit der aktuellen Prozessorauslastung) • Betriebsmittel bzw. geteilter Speicher, worauf mindestens zwei Prozesse gleichzeitig zugreifen wollen
Nachbedingung	Beide Prozesse konnten unter Synchronisation auf das Betriebsmittel zugreifen und ihre Aufgabe beenden. Dies geschieht unter Einhaltung der Rechtzeitigkeit. Dabei wird vermieden, dass Prozesse gegenseitig aufeinander warten müssen.
Nachbedingung beim Fehlerauftritt	Zwei Prozesse besitzen jeweils das vom anderen benötigte Betriebsmittel und warten gegenseitig auf die Freigabe des jeweils anderen Betriebsmittels (Deadlock). Dadurch entsteht eine Verklemmung, welche die Einhaltung der Rechtzeitigkeit und zeitlichen Vorhersagbarkeit verhindert.
Aktoren	Prozesse, Betriebsmittel, Scheduler, Timer
Auslöser (Trigger)	Mindestens zwei Prozesse wollen gegenseitig auf das jeweils andere Betriebsmittel zugreifen.
	Fortsetzung auf der nächsten Seite

Standard-Sequenz	• Timer gibt Taktrate vor für den Scheduler • Prozesse, welche Rechenzeit und Betriebsmittel benötigen, sind im System und kommen bei Scheduler an • Scheduler vergibt Rechenzeit und verteilt Betriebsmittel • Prozesse arbeiten in vorgegebener Reihenfolge • Scheduler synchronisiert den Zugriff auf die Betriebsmittel • Prozesse arbeiten mit der zugeteilten Rechenzeit und den zugeteilten Betriebsmitteln (Reihenfolge der Zuteilung der Rechenzeit und der Tasks erfolgt unter Vermeidung von Deadlocks)
Alternativen / Ausnahmen	• Timer gibt Taktrate vor für den Scheduler • Prozesse, welche Rechenzeit und Betriebsmittel benötigen, sind im System und kommen bei Scheduler an, Scheduler vergibt Rechenzeit und verteilt Betriebsmittel – Prozess A benötigt Rechenzeit und Betriebsmittel 1 und 2 – Prozess B benötigt Rechenzeit und Betriebsmittel 1 und 2 – Prozess A erhält Rechenzeit und Betriebsmittel 1 – Danach erhält Prozess B Rechenzeit und Betriebsmittel 2 – Prozessor A erhält wieder Rechenzeit, hat noch BM1 und benötigt nun BM2 – Prozessor B erhält nun Rechenzeit, hat noch BM2 und benötigt nun BM1 1. Deadlock

Name	**Livelockbehandlung**
Ziel	Mehrere Prozesse können ein und dasselbe Betriebsmittel benötigen. Der Zugriff auf die Betriebsmittel (z. B. Daten, Geräte, Treiber) erfolgt derart, dass nur eine Anwendung gleichzeitig auf das Betriebsmittel zugreift und währenddessen jedoch die Echtzeiteigenschaften (Rechtzeitigkeit, zeitliche Vorhersagbarkeit, Determinismus) eingehalten werden. Außerdem muss gewährleistet sein, dass Prozesse nicht von Betriebsmittel ausgeschlossen werden (Livelockvermeidung).
(Level)	Scheduling
Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor der Rechenzeit zur Verfügung stellt • (Bekanntheit der aktuellen Prozessorauslastung) • Betriebsmittel bzw. geteilter Speicher, worauf mindestens drei Prozesse gleichzeitig zugreifen wollen
Nachbedingung	Die Prozesse konnten unter Synchronisation auf Betriebsmittel zugreifen und ihre Aufgaben beenden. Dies geschieht unter Einhaltung der Rechtzeitigkeit. Dabei wird vermieden, dass Prozesse einen anderen Prozess von der Benutzung eines Betriebsmittels ausschließen.
Nachbedingung beim Fehlerauftritt	Mehrere Prozesse haben immer wieder Zugriff auf ein Betriebsmittel, wohingegen ein (niedrig-priorisierter) Prozess den Zugriff nicht erlangen kann und somit nicht rechtzeitig fertig wird
Aktoren	Prozesse, Betriebsmittel, Scheduler, Timer
Auslöser (Trigger)	Zugriff auf ein Betriebsmittel von mindestens zwei Prozessen, die höher priorisiert sind als ein anderer Task.
	Fortsetzung auf der nächsten Seite

Standard-Sequenz	• Timer gibt Taktrate vor für den Scheduler • Prozesse, welche Rechenzeit und Betriebsmittel benötigen, sind im System und kommen bei Scheduler an • Scheduler vergibt Rechenzeit und verteilt Betriebsmittel • Prozesse arbeiten in vorgegebener Reihenfolge • Scheduler synchronisiert den Zugriff auf die Betriebsmittel • Prozesse arbeiten mit der zugeteilten Rechenzeit und den zugeteilten Betriebsmitteln (unter Vermeidung des Auftretens von Livelocks)
Alternativen / Ausnahmen	• Timer gibt Taktrate vor für den Scheduler • Prozesse, welche Rechenzeit und Betriebsmittel benötigen, sind im System und kommen bei Scheduler an – Prozess A erhält Rechenzeit und BMA – Prozess B und C fordern Rechenzeit und BMA – A rechnet fertig und gibt BMA frei – Prozessor B erhält Rechenzeit und BMA – A fordert Rechenzeit und BMA an – B rechnet fertig und gibt BMA frei – A erhält Rechenzeit und BMA → C wird ausgehungert

Interruptbehandlung

Name	**Interruptbehandlung - Auftreten eines externen Ereignisses**
Ziel	Durch Durchführung von Interrupts wird es ermöglicht, dass zeitnah und unter effizienter Ausnutzung des Prozessors auf externe Ereignisse reagiert werden kann.
(Level)	Scheduling
Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt. • (Bekanntheit der aktuellen Prozessorauslastung) • externes Ereignis (Interrupt)
Nachbedingung	Der Interrupt wurde durchgeführt, wobei bei der Unterbrechung der Tasks die Echtzeitbedingungen eingehalten wurden, d.h., die Geschwindigkeit der Reaktion auf externe Ereignisse ist deterministisch und die Reaktionsfreudigkeit des Systems ist in dergestalt, dass die Rechtzeitigkeit gewährleistet ist.
Nachbedingung beim Fehlerauftritt	*Hart:* Der Interrupt hat bei der Verteilung der Rechenzeit das Einhalten der Deadlines verhindert. Benachrichtigung an Benutzer *Soft:* Nicht-Einhaltung der Rechtzeitigkeit, weiteres Scheduling
Aktoren	Tasks, Scheduler, Timer, Interrupt (externes Ereignis)
Auslöser (Trigger)	Ereignis von außen (Interrupt)
	Fortsetzung auf der nächsten Seite

Standard-Sequenz	• Timer gibt Taktrate vor für den Scheduler • Prozesse, welche Rechenzeit und Betriebsmittel benötigen, sind im System und kommen bei Scheduler an • Prozesse laufen in zugeteilter Reihenfolge • Interrupt tritt auf und benötigt auch Rechenzeit • Unterbrechen und Wegspeichern des aktuellen Tasks durch die ISR und • Durchführen des Interrupts • Zurückspringen vom Interrupt und Wiederaufnehmen des Tasks • Fortsetzung des Schedulings, so dass alle Zeitbedingungen eingehalten werden.
Alternativen / Ausnahmen	• Prozesse halten Deadlines nicht ein. Scheduler gibt Meldung an den Benutzer und führt zum Prozess passende Fehlerbehandlung durch

B.3. Anwendungsszenarien Speicherverwaltung

Dieser Abschnitt beschreibt die Anwendungsszenarien, welche der Kategorie Speicherverwaltung zugeordnet sind.

Zuteilung von Speicher an Tasks

Name	**Zuteilung von Speicher**
Ziel	Tasks benötigen für ihre Ausführung Hauptspeicher, der durch die Speicherverwaltung dem Task zugeteilt wird. Die Zuteilung des Speichers muss rechtzeitig erfolgen, damit der Task in Echtzeit seine Aufgabe abarbeiten kann.
(Level)	Speicherverwaltung
Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt • Prozesse, die Speicher benötigen • Freier Speicher (und belegter Speicher), der verwaltet wird
Nachbedingung	Task kann unter Einhaltung seiner Deadline seine Aufgabe rechnen und hat den benötigten Speicher zugeteilt bekommen.
Nachbedingung beim Fehlerauftritt	Task hat den Speicher nicht rechtzeitig zugeteilt bekommen.
Aktoren	Task, Speicherverwaltung, Prozessor
Auslöser (Trigger)	Task der Speicher anfordert
Standard-Sequenz	• Task fordert Speicher an • Speicher wird Task zugeteilt • Task benutzt Speicher
	Fortsetzung auf der nächsten Seite

Alternativen / Ausnahmen	1. Nicht genügend Speicher verfügbar • Fehlermeldung an den Task 2. Speicherzuteilung dauert zu lange, und der Task kann seine Deadline nicht einhalten. • Fehlermeldung an den Task

Verwaltung gemeinsamer Speicher

Name	**Verwaltung der Zuteilung von gemeinsam genutztem Speicher**
Ziel	Prozesse, die voneinander abhängen bzw. miteinander kommunizieren müssen, tun dies über gemeinsam genutzten Speicher. Der Zugriff auf diesen Speicher muss für alle beteiligten Prozesse derart möglich sein, dass die Daten konsistent bleiben und der Zugriff in endlicher Zeit (zeitliche Vorhersagbarkeit) und innerhalb der zeitlich gesetzten Fristen (Deadlines) rechtzeitig geschieht.
(Level)	Speicherverwaltung
Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt • Prozesse, die Speicher benötigen • Freier Speicher (und belegter Speicher), der verwaltet wird
Nachbedingung	Task hat den benötigten gemeinsamen Speicher rechtzeitig zugeteilt bekommen.
Nachbedingung beim Fehlerauftritt	Task hat den gemeinsam zu nutzenden Speicher nicht rechtzeitig zugeteilt bekommen.
Aktoren	Task, Speicherverwaltung, Prozessor
Auslöser (Trigger)	Tasks, die gemeinsamen Speicher anfordern
Standard-Sequenz	• Tasks mit Interdependenzen fordern gemeinsam zu nutzenden Speicher an • Gemeinsamer Speicher wird den Tasks durch die Speicherverwaltung zugeteilt • Tasks können gemeinsamen Speicher benutzen
	Fortsetzung auf der nächsten Seite

Alternativen / Ausnahmen	1. Nicht genügend Speicher verfügbar • Fehlermeldung an den Task 2. Speicherzuteilung dauert zu lange, und der Task kann seine Deadline nicht einhalten. • Fehlermeldung an den Task

Name	**Koordination des Zugriffs auf gemeinsam genutzten Speicher**
Ziel	Prozesse, die voneinander abhängen bzw. miteinander kommunizieren müssen, tun dies über gemeinsam genutzten Speicher. Der Zugriff auf diesen Speicher muss für alle beteiligten Prozesse derart möglich sein, dass die Daten konsistent bleiben und der Zugriff in endlicher Zeit (zeitliche Vorhersagbarkeit) und innerhalb der zeitlich gesetzten Fristen (Deadlines) rechtzeitig geschieht.
(Level)	Speicherverwaltung
Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt • Prozesse, die Speicher benötigen • Freier Speicher (und belegter Speicher), der verwaltet wird
Nachbedingung	Task kann unter Einhaltung seiner Deadline seine Aufgabe rechnen und hat den benötigten Speicher zugeteilt bekommen. Die Tasks, die den gemeinsamen Speicher nutzen, haben außerdem den benötigten gemeinsamen Speicher erhalten und konnten darauf rechtzeitig abwechselnd zugreifen.
	Fortsetzung auf der nächsten Seite

Nachbedingung beim Fehlerauftritt	1. Bei der Koordination ist ein Fehler aufgetreten, und die verwendeten Daten zur Kommunikation waren ab einem Punkt nicht mehr konsistent und die Ergebnisse der Berechnung der Tasks stimmen nicht. 2. Bei der Koordination konnte durch den synchronisierten Zugriff auf den Speicher die Deadline nicht eingehalten werden (Rechtzeitigkeit)
Aktoren	Task, Speicherverwaltung, Prozessor
Auslöser (Trigger)	Tasks, die gemeinsamen Speicher anfordern
Standard-Sequenz	• Tasks mit Interdependenzen fordern gemeinsam zu nutzenden Speicher an • Gemeinsamer Speicher wird Task den Tasks durch die Speicherverwaltung zugeteilt • Tasks benutzen den gemeinsamen Speicher, so dass keine zwei Tasks gleichzeitig auf den Speicher zugreifen.
	Fortsetzung auf der nächsten Seite

Alternativen / Ausnahmen	1. Inkonsistente Daten: Siehe Synchronisation beim Scheduling 2. Rechtzeitigkeit • Prozesse A und B fordern gemeinsam genutzten Speicher an • Prozesse A und B bekommen gemeinsamen Speicher zugeteilt • A fordert Zugriff auf den gemeinsamen Speicher • Speicherverwaltung koordiniert den Zugriff • B fordert Zugriff auf den gemeinsam genutzten Speicher an und muss warten, da A noch auf den Speicher zugreift – Deadline von B läuft ab (Rechtzeitigkeit)

B.3.1. Schutz Speicher (Abgrenzung der Tasks gegeneinander)

Name	**Abgrenzung Speicherbereiche einzelner Tasks gegeneinander**
Ziel	Prozesse, die voneinander abhängen bzw. miteinander kommunizieren müssen, tun dies über gemeinsam genutzten Speicher. Der Zugriff auf diesen Speicher muss für alle beteiligten Prozesse derart möglich sein, dass die Daten konsistent bleiben und der Zugriff in endlicher Zeit (zeitliche Vorhersagbarkeit) und innerhalb der zeitlich gesetzten Fristen (Deadlines) rechtzeitig geschieht. Zudem ist der Zugriff auf den Speicher der Tasks durch andere Tasks zu schützen. Die Speicherverwaltung stellt sicher, dass ein Task nur in seinen eigenen Adressraum schreiben kann.
(Level)	Speicherverwaltung
	Fortsetzung auf der nächsten Seite

Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt • Prozesse, die Speicher benötigen • Freier Speicher (und belegter Speicher), der verwaltet wird • Speicher, der eindeutig einem Task zugeordnet wurde
Nachbedingung	Prozesse bekommen eigenen Speicher zugeteilt und können auf diesen zugreifen. Auf Speicher von anderen Prozessen kann nicht zugegriffen werden.
Nachbedingung beim Fehlerauftritt	Prozesse können auf Speicher zugreifen, der nicht in ihrem eigenen Adressraum steht.
Aktoren	Prozesse, Speicherverwaltung, Prozessor, Speicher
Auslöser (Trigger)	Prozesse, die auf eigenen Speicher zugreifen, aber auch auf anderen Speicher zugreifen wollen.
Standard-Sequenz	• Prozess benötigt Speicher und fordert diesen an • Prozess erhält eigenen Speicher zugeteilt • Prozess greift auf eigenen Speicher zu • Speicherverwaltung prüft Legitimation/Berechtigung des Tasks und gewährt Zugriff • Prozess kann Daten in seinen Speicher schreiben • Prozess versucht auf nicht-eigenen Speicher zuzugreifen • Speicherverwaltung überprüft Berechtigung und verweigert Zugriff
	Fortsetzung auf der nächsten Seite

Alternativen / Ausnahmen	• Prozess benötigt Speicher und fordert diesen an • Prozess erhält eigenen Speicher zugeteilt • Prozess versucht auf nicht-eigenen Speicher zuzugreifen • Speicherverwaltung überprüft Berechtigung nicht • Prozess greift auf nicht-eigenen Speicher zu und schreibt/liest dort Daten

B.3.2. Verschiebbarkeit der Tasks im Hauptspeicher, Speicherbereinigung

Name	**Verschiebbarkeit der Tasks im Hauptspeicher**
Ziel	Die Speicherverwaltung organisiert den hierarchisch angeordneten Speicher im Computer. Wenn der (schnelle) Hauptspeicher belegt ist, dann werden Prozesse ausgelagert und verschoben. Dieses Verschieben muss derart geschehen, dass, wenn der Prozess wieder Rechenzeit erlangt, dieser schnell genug wieder zurückgeschoben werden kann (Rechtzeitigkeit), so dass der Prozess die gegebene Startzeit und Deadline einhalten kann (zeitliche Vorhersagbarkeit).
(Level)	Speicherverwaltung
Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt • Prozesse, die Speicher benötigen • Freier Speicher (und belegter Speicher), der verwaltet wird
Nachbedingung	Prozesse konnten im Speicher verschoben werden (ausgelagert, wieder neu eingelagert) unter Einhaltung der Echtzeitbedingungen
Nachbedingung beim Fehlerauftritt	Das Verschieben von einem Prozess hat zu einer zu großen Verzögerung geführt, wodurch dieser seine Deadline nicht einhalten konnte
Aktoren	Prozess, Speicher, Speicherverwaltung
Auslöser (Trigger)	Fast vollständig belegter Hauptspeicher, so dass Tasks verschoben werden müssen.
	Fortsetzung auf der nächsten Seite

Standard-Sequenz	• Zeitkritischer Prozess (A) fordert Speicher an • Speicherverwaltung muss anderen (nicht so zeitkritischen) Prozess (B) verschieben • A rechnet fertig, • es laufen andere Prozesse • B fordert Rechenzeit an, und sein Speicher wird wieder zurückgeschoben • B kann fertigrechnen unter Einhaltung seiner Deadlines
Alternativen / Ausnahmen	• Zeitkritischer Prozess (A) fordert Speicher an • Speicherverwaltung muss anderen Prozess (B) (ebenfalls sehr zeitkritisch) auslagern • Prozess A läuft • Prozess B wird im Anschluss direkt wieder eingelagert, kann jedoch nicht innerhalb seiner Deadline fertig rechnen.

Name	**Verschiebbarkeit der Tasks im Hauptspeicher - Task Lock**
Ziel	Die Speicherverwaltung organisiert den hierarchisch angeordneten Speicher im Computer. Wenn der (schnelle) Hauptspeicher belegt ist, dann werden Prozesse ausgelagert und verschoben. Dieses Verschieben muss derart geschehen, dass, wenn der Prozess wieder Rechenzeit erlangt, dieser schnell genug wieder zurückgeschoben werden kann (Rechtzeitigkeit), so dass der Prozess die gegebene Startzeit und Deadline einhalten kann (zeitliche Vorhersagbarkeit). Gegebenenfalls sollte das Verschieben von besonders zeitkritischen Prozessen mittels Task Lock vermieden werden.
(Level)	Speicherverwaltung
	Fortsetzung auf der nächsten Seite

Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt. • Prozesse, die Speicher benötigen • Freier Speicher (und belegter Speicher), der verwaltet wird
Nachbedingung	Prozesse konnten im Speicher verschoben werden (ausgelagert, wieder neu eingelagert) unter Einhaltung der Echtzeitbedingungen, bzw. das Verschieben konnte mittels Task Lock vermieden werden
Nachbedingung beim Fehlerauftritt	Das Verschieben von einem Prozess hat zu einer zu großen Verzögerung geführt, wodurch dieser seine Deadline nicht einhalten konnte, bzw. Task Lock war nicht möglich
Aktoren	Prozess, Speicher, Speicherverwaltung
Auslöser (Trigger)	Fast vollständig belegter Hauptspeicher, so dass Tasks verschoben werden müssen.
Standard-Sequenz	• Zeitkritischer Prozess (A) fordert Speicher an und wird im Speicher verriegelt • Speicherverwaltung muss anderen (nicht so zeitkritischen) Prozess (B) verschieben • A rechnet fertig • es laufen andere Prozesse • B fordert Rechenzeit an, und sein Speicher wird wieder zurückgeschoben • B kann fertigrechnen unter Einhaltung seiner Deadlines
Alternativen / Ausnahmen	-

B.3.3. Speicherbereinigung

Name	**Speicherbereinigung**
Ziel	Um immer genügend Speicher im Hauptspeicher für alle Prozesse zur Verfügung zu stellen, muss der Speicher regelmäßig bereinigt werden. Dies muss jedoch so erfolgen, dass dabei die Echtzeit-Prozesse nicht unterbrochen werden, bzw. die Rechtzeitigkeit und zeitliche Vorhersagbarkeit eingehalten werden.
(Level)	Speicherverwaltung
Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt. • Prozesse, die Speicher benötigen • Speicher, der verwaltet wird und reorganisiert werden muss
Nachbedingung	Der Speicher wurde aufgeräumt, ohne dabei die Prozesse zu behindern
Nachbedingung beim Fehlerauftritt	1. Der Speicher wurde nicht aufgeräumt 2. Der Speicher wurde aufgeräumt, hat dabei jedoch die Prozesse in ihrer Ausführung behindert, so dass die Rechtzeitigkeit und zeitliche Vorhersagbarkeit nicht eingehalten werden konnte
Aktoren	Speicher, Prozesse, Speicherverwaltung
Auslöser (Trigger)	Stark fragmentierter Speicher
Standard-Sequenz	• Speicherbereinigung ohne Reorganisierungspause • Inkrementelle Speicherbereinigung
	Fortsetzung auf der nächsten Seite

Alternativen / Ausnahmen	Speicherbereinigung unterbricht Prozesse in der Ausführung (nicht echtzeittauglich)

B.3.4. Kommunikation

Name	**Kommunikation von Echtzeittasks und normalen Tasks**
Ziel	Zwischen Echtzeit- und Nicht-Echtzeittasks muss eine Kommunikation möglich sein. Kommunikation kann stattfinden, wenn z.B. Daten in Echtzeit gelesen werden um darauf „normal“ bearbeitet zu werden. Diese Kommunikation findet in der Regel über Echtzeit-Warteschlangen bzw. -Puffer statt. Es muss dabei möglich sein, dass Echtzeittasks rechtzeitig lesen, was die normalen Tasks schreiben, aber auch umgekehrt Echtzeittasks schreiben und normale Tasks das Ergebnis auslesen können. Diese Vorgänge sollten möglichst gleichzeitig geschehen. Besonders wichtig ist dabei die Verfügbarkeit der Daten, deshalb muss das was in den Puffer geschrieben wurde, ausreichend schnell wieder aus dem Puffer gelesen werden können, ohne dass Daten überschrieben werden, also verloren gehen.
(Level)	Speicherverwaltung - Kommunikation
Vorbedingung	Echtzeittasks und Nicht-Echtzeittasks, die mitteinander kommunizieren wollen
Nachbedingung	Der Echtzeittask konnte rechtzeitig lesen, was der normale Task geschrieben hat bzw. der Echtzeittask konnten Daten schreiben, die vom normelen Task ausgelesen wurden, bevor der Kommunikationspuffer vom Echtzeittask wieder überschrieben wurde.
Nachbedingung beim Fehlerauftritt	• Nicht-Einhaltung der Deadline, da zu langsam gelesen/geschrieben wurde • keine Konsistenz der Daten, da der Kopf im Ringpuffer vom Echtzeittask überschrieben wurde, bevor der Nicht-Echtzeittask die Daten auslesen konnte
Aktoren	Normaler Task, Echtzeittask, Echtzeit-Warteschlange
Auslöser (Trigger)	Echtzeit- und Nicht-Echtzeittask, die miteinander kommunizieren müssen.
	Fortsetzung auf der nächsten Seite

Standard-Sequenz	• Echtzeittask nimmt Messungen in Echtzeit vor • Daten werden in Echtzeit in den Puffer geschrieben • Daten werden von den Nicht-Echtzeittasks so schnell wie möglich gelesen und weiterverarbeitet
Alternativen / Ausnahmen	• Echtzeit wird nicht eigehalten, da zu langsam aus dem Puffer gelesen wird • Echtzeit wird nicht eingehalten, da die geschriebenen Daten überschrieben werden und somit nicht mehr verfügbar sind.

B.4. Anwendungsszenarien Zeitgeber

Dieser Abschnitt beschreibt die Anwendungsszenarien, welche der Kategorie Zeitgeber zugeordnet sind.

B.4.1. Zeitgeber

Name	**Benutzen des Timers als Zeitgeber**
Ziel	Die Anwendungen in Echtzeitsystemen müssen sich an die Echtzeitbedingungen halten. Um zeitnah, rechtzeitig und deterministisch arbeiten zu können, wird dafür ein genauer Zeitgeber benötigt. Dieser gibt über die Taktrate vor, mit welcher Genauigkeit Anwendungen auf Ereignisse reagieren.
(Level)	Timer
	Fortsetzung auf der nächsten Seite

Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt • Timer, der die Taktrate vorgibt
Nachbedingung	Durch den Zeitgeber konnten alle Anwendungen rechtzeitig ihre Aufgabe erfüllen
Nachbedingung beim Fehlerauftritt	-
Aktoren	Anwendung (Prozesse), Timer, Scheduler, Interrupt
Auslöser (Trigger)	Das System läuft
Standard-Sequenz	Der Timer gibt eine Taktrate vor • Anwendungen nutzen die Taktrate zur Zeitmessung • Scheduler benutzt die Taktrate, um Prozesse zu starten, zu beenden und zu wechseln (etc.) • Der Interrupthandler benutzt die Taktrate, um die ISR durchzuführen
Alternativen / Ausnahmen	

B.4.2. Anpassen der Taktfrequenz

Name	**Anpassen der Taktfrequenz durch den Benutzer/Anwender**
Ziel	Um Echtzeit zu gewährleisten und auf alle Ereignisse rechtzeitig reagieren zu können, muss der Timer als Zeitgeber so genau eingestellt sein, dass das System damit seine Aufgaben (Prozessverwaltung, Speicherverwaltung, Kommunikation) im geforderten Rahmen erfüllt. Die Genauigkeit des Timers sollte dabei vom Anwender einstellbar sein.
	Fortsetzung auf der nächsten Seite

(Level)	Timer
Vorbedingung	Timer als Zeitgeber mit Taktfrequenz
Nachbedingung	Timer als Zeitgeber mit geänderter Taktfrequenz
Nachbedingung beim Fehlerauftritt	Timer als Zeitgeber mit unzureichend genauer Taktfrequenz
Aktoren	Timer, Benutzer
Auslöser (Trigger)	Nicht passende Taktrate
Standard-Sequenz	• Timer gibt Taktrate vor • Benutzer ändert Taktrate • Timer gibt nun angepasste Taktrate vor
Alternativen / Ausnahmen	• Timer gibt Taktrate vor • Benutzer ändert Taktrate • Timer gibt angepasste Taktrate vor • Zeitbedingungen können nicht eingehalten werden.

B.4.3. Watchdog

Name	**Benutzung des Watchdogs des Timers zur Überwachung von zeitkritischen Prozessen**
Ziel	Echtzeitprozesse müssen rechtzeitig innerhalb einer vorgegebenen Deadline ihre Ergebnisse liefern. Um zu vermeiden, dass sie blockiert werden, überwacht der Watchdog diese Prozesse und startet, wenn er keine Rückmeldung (in Form eines Lebenszeichens) vom Prozess erhält, diesen neu.
(Level)	Timer
	Fortsetzung auf der nächsten Seite

Vorbedingung	• Menge von Prozessen mit zugeordneten Startzeiten und Deadlines • Mindestens ein Prozessor, der Rechenzeit zur Verfügung stellt. • Watchdog, der zeitkritische Prozesse überwachen kann.
Nachbedingung	Durch den Zeitgeber konnten alle Anwendungen rechtzeitig ihre Aufgabe erfüllen
Nachbedingung beim Fehlerauftritt	-
Aktoren	Anwendungen (Prozesse), Timer, Scheduler, Interrupt
Auslöser (Trigger)	Zu überwachender zeitkritischer Prozess
Standard-Sequenz	• Zeitkritischer Prozess wird gestartet • Watchdog überwacht diesen Prozess • Der Prozess wird unterbrochen • Etwas anderes wird im System durchgeführt, so dass der Prozess sich in einem Deadlock befindet • Watchdog erhält kein Lebenszeichen von seinem Prozess (Timeout) • Watchdog startet den zeitkritischen Prozess neu
Alternativen / Ausnahmen	-

B.4.4. Interruptbehandlung

Name	**Interruptbehandlung**
Ziel	Um auf externe Ereignisse schnell und unter Einhaltung von Deadlines reagieren zu können, muss in einem Echtzeitbetriebssystem ein Timer als Zeitgeber verwendet werden, der genau genug für alle zeitkritischen Anwendungen arbeitet. Bei der Interruptbehandlung muss auf die Interruptanfrage in einer gewissen Geschwindigkeit reagiert werden (Determinismus), und die Reaktion darauf (ISR) muss ausreichend schnell erfolgen (Reaktionsfreudigkeit). Insgesamt muss die Antwortzeit auf externe Ereignisse in einem Rahmen liegen, um die Rechtzeitigkeit und somit Echtzeitfähigkeit des Systems zu gewährleisten.
(Level)	Timer
Vorbedingung	• Laufendes System • Eingeschaltete Interrupts / Möglichkeit der Interrupts
Nachbedingung	Der Interrupt wurde im Rahmen der Einhaltung der Rechtzeitigkeit durchgeführt.
Nachbedingung beim Fehlerauftritt	Der Interrupt wurde durchgeführt, jedoch ist die Rechtzeitigkeit für einige Prozesse nicht mehr gewährleistet, da der Interrupt zu lange den Scheduler unterbrochen hatte.
Aktoren	Timer, Interrupt, Scheduler
Auslöser (Trigger)	Externes Ereignis schickt Interruptsignal an das System
Standard-Sequenz	• Interruptanforderung an CPU • Ausführen der ISR, d.h. Unterbrechen, Wegspeichern, Zurückspringen, Durchführung des Interrupts • Fortsetzen des normalen Schedulings
	Fortsetzung auf der nächsten Seite

Alternativen / Ausnahmen	• Interruptanforderung an CPU • Durchführen des Interrupts (ISR) • Nichteinhalten der Rechtzeitigkeit von mindestens einem der unterbrochenen Tasks

Curriculum Vitae

Name	John F. Schommer
Geburtstag	29. Juli 1977
Geburtstort	Mönchengladbach

2008 – 2013	Wissenschaftlicher Assistent am Lehrstuhl Informatik 11 an der RWTH Aachen Universität
2000 – 2008	Studium der Informatik an der RWTH Aachen Universität
1998 – 1999	Grundwehrdienst
1988 – 1998	Gymnasium Korschenbroich
1984 – 1988	Städtische Gemeinschaftsgrundschule Andreas-Schule in Korschenbroich

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2017-01 *	Fachgruppe Informatik: Annual Report 2017
2017-02	Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity
2017-04	Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE
2017-05	Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems
2017-06	Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy
2017-07	Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++
2017-08	Thomas Gerlitz: Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts
2017-09	Muhammad Hamad Alizai, Jan Beutel, Jó Ágila Bitsch, Olaf Landsiedel, Luca Mottola, Przemyslaw Pawelczak, Klaus Wehrle, and Kasim Sinan Yildirim: Proc. IDEA League Doctoral School on Transiently Powered Computing
2018-01 *	Fachgruppe Informatik: Annual Report 2018
2018-02	Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2018-03	Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices
2018-04	Andreas Ganser: Operation-Based Model Recommenders

2018-05	Matthias Terber: Real-World Deployment and Evaluation of Synchronous Programming in Reactive Embedded Systems
2018-06	Christian Hensel: The Probabilistic Model Checker Storm - Symbolic Methods for Probabilistic Model Checking
2019-01 *	Fachgruppe Informatik: Annual Report 2019
2019-02	Tim Felix Lange: IC3 Software Model Checking
2019-03	Sebastian Patrick Grobosch: Formale Methoden für die Entwicklung von eingebetteter Software in kleinen und mittleren Unternehmen
2019-05	Florian Göbe: Runtime Supervision of PLC Programs Using Discrete-Event Systems
2020-01 *	Fachgruppe Informatik: Annual Report 2020
2020-02	Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla: Towards an Isabelle Theory for distributed, interactive systems - the untimed case

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.