TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Realzeit-Computersysteme

Eine realzeitfähige Architektur zur Integration kognitiver Funktionen

Matthias Goebl

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr.-Ing. J. Eberspächer
Prüfer der Dissertation:	1. Univ.-Prof. Dr.-Ing. G. Färber, em.
	2. Univ.-Prof. Dr.-Ing. R. Dillmann, Universität Karlsruhe (TH)

Die Dissertation wurde am 04.06.2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 21.07.2009 angenommen.

Danksagung

Die vorliegende Dissertation entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Realzeit-Computersysteme der Technischen Universität München. Teile dieser Arbeit wurden von der *Deutschen Forschungsgemeinschaft* (DFG) als Teil des Projekts „Architektur und Schnittstellen für kognitive Funktionen in Fahrzeugen“ (ASKOF) unter dem Förderkennzeichen Fa 109/17-1 und anschließend im Rahmen des Sonderforschungsbereichs/Transregio 28 „Kognitive Automobile“ als Teilprojekt C3 mit dem Titel „Hardware und Software-Architektur“ gefördert.

Mein größter Dank gilt meinem Doktorvater, Herrn Prof. Dr.-Ing. Georg Färber, für das in mich gesetzte Vertrauen und die gewährten Freiheiten. Er ermöglichte diese Arbeit und förderte die Selbständigkeit, stand jederzeit mit fachlichen aber auch allgemeinen Ratschlägen zur Seite und trug damit wesentlich zu ihrem Gelingen bei.

Herrn Prof. Dr.-Ing. Rüdiger Dillmann danke ich für sein großes Interesse an dieser Arbeit, seine Bereitschaft zur Übernahme des Zweitberichts und für seine hilfreichen Anregungen.

Ich möchte mich bei allen Mitarbeitern des Lehrstuhls für Realzeit-Computersysteme, insbesondere bei dem neuen Institutsleiter Herrn Prof. Dr. sc. Samarjit Chakraborty, für die Unterstützung, das gute Arbeitsklima und die zahlreichen interessanten Diskussionen bedanken. Besonderer Dank geht an meinen Kollegen Florian Rattei, der auch für den SFB/TR 28 am Lehrstuhl arbeitet und mit dem ich zusammen mit weiteren Projektkollegen manche Nacht bei der Arbeit in unserem Versuchsträger verbracht habe.

Ebenfalls bedanke ich mich für die gute und produktive Zusammenarbeit bei allen Projektkollegen an beiden Standorten der „Kognitiven Automobile“, in Karlsruhe und München sowie für den regen Austausch, auch an den Vorabenden der zahlreichen gemeinsamen Treffen. Mein Dank gilt außerdem meinem Team „AnnieWAY“ auf der DARPA Urban Challenge für eine spannende und erfolgreiche Zeit in Kalifornien. Darüber hinaus bedanke ich mich bei allen Freunden und Kollegen, die mich bei meiner Arbeit unterstützt haben und so ebenfalls zum Gelingen beigetragen haben.

Nicht zuletzt danke ich meiner Familie und meiner Freundin Tina, die mir in jeder Phase Beistand geleistet, mich angespornt und uneingeschränkt unterstützt haben.

Für Tina.

Inhaltsverzeichnis

Verzeichnis der verwendeten Symbole viii

1 Einleitung 1
1.1 Motivation . . . 1
1.2 Ziele dieser Arbeit . . . 2
1.3 Gliederung . . . 3

2 Stand der Technik 5
2.1 Architekturen von (autonomen) Forschungsfahrzeugen . . . 5
2.1.1 VaMoRs und VaMP (UniBW-M/ISF) . . . 6
2.1.2 VITA und UTA (Daimler) . . . 9
2.2 Autonome Fahrzeuge im DARPA Grand Challenge . . . 13
2.2.1 DARPA Urban Challenge . . . 13
2.2.2 Tartan Racing (Carnegie Mellon University) . . . 15
2.2.3 Junior (Stanford Racing) . . . 18
2.2.4 Odin (Team VictorTango) . . . 19
2.2.5 Rocky (Team Urbanator) . . . 21
2.3 Robotikarchitekturen . . . 22
2.4 Nicht-Blockierende Datenaustauschprotokolle . . . 24
2.4.1 Sperrenfreie Protokolle . . . 25
2.4.2 Wartefreie Protokolle . . . 26
2.4.3 Schleifenfreie Protokolle . . . 27

3 Grundlagen 29
3.1 Automatisierung von Prozessen . . . 29
3.2 Realzeitsysteme . . . 30
3.3 Realzeiteigenschaften von PC-Hardware und Peripherie . . . 31
3.4 Realzeiteigenschaften von Betriebssystemen . . . 32
3.4.1 Standardbetriebsystem Linux . . . 34
3.4.2 Echtzeiterweiterungen . . . 35
3.5 Echtzeiteigenschaften der Speicherverwaltung . . . 37
3.5.1 Dynamische Speichervergabe . . . 39
3.5.2 Echtzeitfähigkeit von Speicherverwaltungsalgorithmen . . . 39
3.5.3 Speicherzuteilung im Betriebssystem Linux . . . 40
3.6 Kognitive Automobile . . . 41
3.6.1 Sonderforschungsbereich/Transregio 28 . . . 41

3.6.2 Projektüberblick 42
3.6.3 Funktionale Architektur 43

4 Hardwareplattform für kognitive Fahrzeuge 46
4.1 Heutige Bordnetzarchitekturen 46
4.2 Hardwarearchitektur für ein kognitives Automobil 47
4.3 Fahrzeugrechnersystem 48
4.3.1 Ausgewählte AMD Opteron Architektur 48
4.3.2 Datenfluss bei der Aufzeichnung 50

5 Realzeitfähige Softwarearchitektur 52
5.1 Anforderungen 52
5.2 Integrationsframework 53
5.2.1 Datengetriebener Ansatz 53
5.2.2 Datenfluss 54
5.2.3 Datenbanken zur Informationsdistribution 54
5.3 Die Realzeitdatenbasis für kognitive Automobile (KogMo-RTDB) 56
5.3.1 RTDB-Objekte 57
5.3.2 Zugriffsmethoden 59
5.3.3 Zeitverwaltung 60
5.3.4 Historienkonzept 63
5.3.5 Blockierungsfreiheit und Datenkonsistenz 65
5.3.6 Benachrichtigungsmethode 70
5.3.7 Speicherverwaltung 71
5.3.8 Applikationsschnittstellen 74
5.4 Harte Echtzeitfähigkeit 77
5.4.1 Systembetrachtung 77
5.4.2 Reaktionszeit des Rechnersystems 79
5.4.3 Komponenten der echtzeitfähigen Softwarearchitektur 79
5.4.4 Betriebssystem 80
5.4.5 Interprozesskommunikation durch die KogMo-RTDB 83
5.4.6 Hardwareschnittstellen 84
5.4.7 Schnittstellentreiber 85
5.4.8 Echtzeitrechenprozesse 86
5.5 Kooperation verschieden harter Echtzeitprozesse 90
5.5.1 Harte Echtzeit und Kooperation im kognitiven Automobil 91
5.5.2 KogMo-RTDB als Bindeglied zwischen Echtzeitebenen 92
5.5.3 Anforderungen an echtzeitfähigen Datenaustausch 93
5.5.4 Erlaubte deterministische Blockierungen 94
5.5.5 Unzulässige blockierende Funktionalitäten 95
5.5.6 Synchronisierungsmethoden 97
5.5.7 Synchronisation von KogMo-RTDB Transaktionen 98
5.5.8 Prioritätserhalt bei konkurrierenden Transaktionen 100
5.5.9 Echtzeitnachweis 106

5.5.10 Echtzeitfähige Einfüge- und Löschoperationen . . . 110
5.6 Nicht-echtzeitfähiges Funktionsentwicklungssystem . . . 111
5.7 Datenaufzeichnung und Simulation . . . 112
5.7.1 Aufzeichnungsmethode . . . 113
5.7.2 Aufzeichnungsformat . . . 114
5.7.3 Einspielen einer Aufzeichnung . . . 116
5.7.4 Einsatz von Aufzeichnungen zur Simulation . . . 118

6 Messergebnisse und Anwendungen 120
6.1 Ausführungszeiten von KogMo-RTDB-Operationen . . . 120
6.1.1 Latenzzeit der Interprozesskommunikation . . . 122
6.1.2 Abhängigkeit von der Objektgröße . . . 122
6.1.3 Latenzzeit des Rechnersystems einschließlich Ein/Ausgabe . . . 123
6.2 Datenaufzeichnung . . . 125
6.2.1 Aufzeichnungsbandbreite . . . 125
6.3 Automatisierte Prozessanalyse . . . 126
6.3.1 Ressourcenbedarfsermittlung . . . 127
6.3.2 Laufzeitbestimmung . . . 127
6.3.3 Echtzeitfähige Prozessüberwachung . . . 128
6.4 Autonome Versuchsfahrten . . . 131
6.4.1 Videobasiertes Spurhalten mit bewegter Kameraplattform . . . 132
6.4.2 Echtzeitfähige automatische Notbremsung . . . 135
6.5 Skalierbarkeit auf eingebettete Systeme . . . 138
6.6 Weitere Anwendungsfelder und Projekte . . . 139

7 Zusammenfassung und Bewertung 140
7.1 Bewertung und Vergleich mit dem Stand der Technik . . . 140
7.1.1 Hardwarearchitektur . . . 141
7.1.2 Effizienz des Datenflusses . . . 141
7.1.3 Zeitverhalten . . . 142
7.1.4 Skalierbarkeit . . . 143
7.1.5 Netzwerklösungen . . . 143
7.1.6 Aufzeichnungsmethoden . . . 144
7.1.7 Generik der Architektur . . . 145
7.1.8 Flexibilität der Architektur . . . 146
7.1.9 Blockierungsfreier Datenaustausch . . . 146
7.2 Fazit . . . 147
7.3 Ausblick . . . 148

A Protokoll zum blockierungsfreien Datenaustausch 149
A.1 Ablauf des Schreibalgorithmus . . . 149
A.2 Ablauf des Lesealgorithmus . . . 151

Literaturverzeichnis 153

Verzeichnis der verwendeten Symbole

API	Application Programming Interface (Programmierschnittstelle)
CAN	Controller Area Network (Serielles Bussystem im Fahrzeug)
DARPA	Defense Advanced Research Projects Agency (Forschungsbehörde des Verteidigungsministeriums der Vereinigten Staaten von Amerika)
GB	Gigabyte (2^{30} Byte) ($\widehat{=}$ GiB in [46])
GPOS	General Purpose Operating System
GPS	Global Positioning System (Satellitengestützte Positionsbestimmung)
IA-32	Intel Architecture 32-Bit ($\widehat{=}$ x86)
IEEE	Institute of Electrical and Electronics Engineers (Berufsverband)
I/O	Input/Output (Ein-/Ausgabe, E/A)
IPC	Inter-Process Communication (Interprozesskommunikation)
KogMo-RTDB	Realzeitdatenbasis für kognitive Automobile
LIDAR	Light Detection and Ranging (Optische Entfernungsmessung)
MB	Megabyte (2^{10} Byte) ($\widehat{=}$ MiB in [46])
NRT	Non-Real-Time (nicht-echtzeitfähig)
NUMA	Non-Unified Memory Architecture
RADAR	Radio Detection and Ranging (Entfernungs- und Geschwindigkeitsmessung durch elektromagnetische Wellen)
RAM	Random Access Memory
RT	Real-Time (echtzeitfähig)
RTOS	Real-Time Operating System
SFB/TR	Sonderforschungsbereich/Transregio
SMP	Symmetric Multiprocessing
TSC	Time-Stamp Counter von x86-kompatiblen Prozessoren
WCET	Worst-Case Execution Time
x86	Befehlssatz einer verbreiteten Mikroprozessor-Architektur von Intel

Formelzeichen:

c	Rechenzeitbedarf
n_{bytes}	Größe eines Datenblocks
OID	Objektidentifikationsnummer
PID	Prozessidentifikation
t_{cycle}	Zykluszeit der Aktualisierung
$T_{history}$	Zeitspanne der Datenhistorie
TID	Objekttypenkennung

Zusammenfassung

Eingebettete Systeme finden sich inzwischen in verschiedensten Geräten des Alltags, von der Unterhaltungselektronik bis zu Automobilen. Deren Einsatzgebiete umfassen, abgesehen von rechenaufwendigen Tätigkeiten, zunehmend auch kognitive Aufgabenstellungen, wie sie z. B. in intelligenten Fahrzeugen auftreten. Dabei müssen komplexe Entscheidungen getroffen, das Umfeld verstanden und mit Beteiligten interagiert werden. Obwohl kognitive Systeme datenintensiv sind und meist aus vielen Teilfunktionen bestehen, müssen sie dennoch schritthaltend reagieren. Daher stellt der echtzeitfähige Austausch großer Datenmengen zwischen den einzelnen Funktionen eine große Herausforderung dar.

In dieser Arbeit wird dafür eine abgestimmte Hard- und Softwarearchitektur vorgestellt. Basierend auf einer Untersuchung der Echtzeiteigenschaften von PC-Hardware wird eine leistungsfähige Rechnerplattform mit mehreren Prozessorkernen und -knoten erarbeitet, deren Bussysteme die anfallenden Datenströme effizient transportieren. Um darauf Tasks mit verschiedenen Zeitanforderungen auszuführen, wird ein flexibles Prozessmodell entworfen. Es basiert auf einer Auswahl von geeigneten Echtzeitbausteinen auf Betriebssystemebene, u. a. zur Ereignispriorisierung, Treibermodellierung und Speicherverwaltung.

Eine Kernkomponente dieser Architektur bildet die entwickelte *Realzeitdatenbasis für kognitive Automobile* (*KogMo-RTDB*). Sie fungiert als zentrale Schnittstelle zum reibungslosen Datenaustausch zwischen allen Softwaremodulen eines Systems. Dafür wurden neue Methoden zur echtzeitfähigen Kommunikation und Zusammenarbeit von Echtzeit- mit Nicht-Echtzeit-Modulen geschaffen. Ein neuer Algorithmus für ein blockierungsfreies Schreib-/Leseprotokoll sorgt stets für Datenkonsistenz. Sein Historienkonzept ermöglicht die zeitliche Entkopplung verschiedener Wahrnehmungs- und Handlungsebenen eines kognitiven Systems. Darüber hinaus wird eine rückwirkungsfreie Methode zur Datenaufzeichnung präsentiert und ihr Einsatz in der Simulation erläutert.

Um die Echtzeitfähigkeiten der konzipierten Architektur zu beweisen, wird ein Echtzeitnachweis durchgeführt. Die Effizienz ihrer Implementierung wird durch zahlreiche Messungen bestätigt, in die alle relevanten Hard- und Softwarekomponenten miteinbezogen werden.

Die Praxistauglichkeit der entwickelten Architektur und ihrer Methoden wird durch den erfolgreichen Einsatz in mehreren kognitiven Automobilen belegt. Deren Übertragbarkeit wird durch die Verwendung außerhalb des beabsichtigen Einsatzgebiets der Fahrerassistenzsysteme demonstriert. Im Vergleich zu bisherigen Konzepten für intelligente Fahrzeuge zeichnet sich diese Architektur insbesondere durch ihr Echtzeitverhalten, ihre Effizienz und ihre Flexibilität aus.

Abstract

Today, embedded systems can be seen in a variety of everyday devices from consumer electronics to vehicles. In addition to handling compute-intensive tasks, there is an increasing use of embedded systems for cognition-oriented tasks as well, e.g. those arising in intelligent vehicles. Such tasks require making complex decisions, understanding the environment and interacting with other parties. Apart from being data-intensive and being constituted by multiple sub-tasks, cognitive systems have stringent timing constraints. Therefore, the exchange of large data sets in real-time between multiple cognitive functions poses a considerable challenge.

This work addresses the above real-time constraint by proposing a complementary hard- and software-architecture. Based on the timing characteristics of off-the-shelf computer hardware, an efficient multi-core, multi-node computation platform has been selected. Its bus systems are capable of transporting all data streams. In order to execute tasks with different timing requirements, a flexible process model has been created. It is based on an appropriate OS-level selection of real-time components for different tasks, such as event prioritization, driver modeling and memory management.

A key component of this architecture is the proposed *real-time database for cognitive automobiles* (*KogMo-RTDB*). It serves as the central interface for unobstructed data exchange between all software modules within the system. Towards this, new methods have been created for real-time communication and co-operation between both real-time and non-real-time modules. A new algorithm for a lock-free read/write protocol always ensures data consistency. A concept of history in this algorithm enables temporal decoupling of different perception and action levels within the cognitive system at hand. In addition, a new interference-free data-logging method has been used, which has been shown to be useful for simulating cognitive systems.

A static-analysis based proof has been used to demonstrate the real-time capabilities of the designed architecture. Further, the efficiency of the overall system has also been confirmed by several measurements, which incorporated all the relevant hard- and software-components.

The practical utility of the developed architecture and methods has been proved by their successful deployment in several cognitive vehicles. Further, their portability has been demonstrated by their use in areas outside those for which they were originally developed, e.g. advanced driver assistance systems. In summary, compared to known hardware/software architectures for intelligent vehicles, the proposed architecture has demonstrable benefits in terms of real-time properties, efficiency and flexibility.

1 Einleitung

1.1 Motivation

Eine besondere Faszination erzeugen Computer, wenn sie Aufgaben übernehmen, die über die reine Datenverarbeitung herausgehen und ein Ergebnis nicht nur auf einem Bildschirm angezeigt wird, sondern sie Vorgänge steuern, die eine unmittelbare Auswirkung auf die Realität haben. In den Anfängen waren dies beispielsweise Prozessleitrechner in der Automatisierungstechnik. Mit der zunehmenden Miniaturisierung der Rechnertechnik finden sich heute eingebettete Rechnersysteme an vielen Stellen im Alltag.

Wurden in der klassischen Datenverarbeitung die Eingangsdaten noch vom Operator für den Computer aufbereitet, besteht heute die Herausforderung darin, dass eingebettete Systeme in der Welt ohne menschliche Hilfe ihre Aufgabe erledigen können. Dies trägt dazu bei, dass Computer den Menschen das Leben erleichtern, statt es, wie oft bemängelt wird, zu erschweren.

Ein sehr interessantes Einsatzgebiet eingebetteter Systeme ist das Automobil, in dem Fahrerassistenzsysteme dazu beitragen, den Fahrer von immer wiederkehrenden Aufgaben zu entlasten. War für die Realisierung einer Tempomatfunktion noch eine analoge Regelstrecke ausreichend, erfordert ein robuster Abstandsregeltempomat einen Steuerrechner, der eine regelbasierte Auswahl z. B. des Zielfahrzeugs treffen muss. Schon für die scheinbar minimale Erweiterung um ein automatisches Wiederanfahren aus dem Stillstand ist meist noch eine Sicherheitsfreigabe durch den Fahrer nötig. Denn die Fähigkeit, solche komplexen Entscheidungen sicher zu treffen, setzt ein umfassendes Umweltverständnis voraus, das derzeit noch kein Rechnersystem so gut wie der Mensch beherrscht.

Kognitive Systeme, die das Ziel haben, ihre Umwelt zu verstehen und mit ihr zu interagieren, könnten diese Lücke in Zukunft füllen und eigenständig komplexere Entscheidungen treffen. Es wird zwar noch einige Zeit dauern, bis die dafür notwendige Robustheit erreicht ist, dennoch konnten z. B. auf den DARPA Grand Challenges erste Erfolge beobachtet werden. Dort wurden mit Rechnern ausgestattete Fahrzeuge „sich selbst überlassen“, haben ohne menschlichen Fahrer eigenständig vorgegebene Missionen erfüllt und dabei mit anderen Fahrzeugen interagiert. Der Einsatz in autonomen Fahrzeugen ist hier ein sehr anschauliches Anwendungsgebiet kognitiver Systeme, deren Ergebnisse zwischenzeitlich in zukünftige Fahrerassistenzsysteme einfließen könnten.

Die Integration aller benötigten kognitiven Funktionen stellt jedoch neue Anforderungen an die Systemarchitektur. War es früher ausreichend, einzelne analoge Mess- oder Stell-

größen von Regelmodulen in Hardware elektrisch miteinander zu verbinden, müssen nun umfangreiche Zusammenhänge und Erkenntnisse kommuniziert werden, die die Übertragungskapazitäten heutiger Fahrzeugbusse sprengen. Aufgrund zunehmender Integrationsdichte von Rechnersystemen und der Verbreitung preiswerter Multicore- und Multinode-Plattformen eröffnet sich die Möglichkeit zur Hardwarekonsolidierung, weg von einem vernetzten verteilten Steuergeräteverbund hin zu einer zentralen Recheneinheit. Ein solches „kleines Rechnercluster“ bietet ausreichend Kommunikationsbandbreite, sodass nun passende Architekturen gefragt sind, um diese Leistungsfähigkeit zu nutzen.

1.2 Ziele dieser Arbeit

Ziel dieser Arbeit ist eine aufeinander abgestimmte Hard- und Softwarearchitektur zur Integration kognitiver Funktionen. Diese findet ihre primäre Anwendung in intelligenten Fahrzeugen, soll aber auch darüber hinaus nutzbar sein, um verschiedene kognitive Funktionen auf einem Rechnersystem zu integrieren. Diese Integration soll auf Softwareebene derart geschehen, dass einzelne Funktionen gleichzeitig arbeiten und miteinander kommunizieren können, ohne sich gegenzeitig zu stören. Die Architektur soll einheitliche Kommunikationsschnittstellen bereitstellen, und gleichzeitig weitgehende Freiheiten bei der Definition der zu übermittelnden Informationen lassen.

Verfügbare Forschungsarchitekturen für Fahrerassistenzsysteme haben meist bestimmte Schwerpunkte wie Videoverarbeitung und damit einhergehende Designfestlegungen wie einen festen Arbeitstakt. Architekturen aus dem Nachbargebiet der Robotik sind oft auf bestimmte Hardwarekonfigurationen beschränkt wie omnidirektionale Plattformen oder Laserscanner. Die Architektur in dieser Arbeit soll keine bestimmte Sensorik oder Aktorik voraussetzen. Darüber hinaus soll die Aufnahme neuer Softwarefunktionen oder Hardwarekomponenten jederzeit möglich sein. Zudem soll eine neue Funktion verfügbare Daten leicht mitnutzen können, ohne dass neue Schnittstellen geschaffen werden müssen. Dabei muss auch sichergestellt werden, dass die bestehenden Funktionalitäten nicht durch eine neue Komponente gestört werden.

Ein besonderer Schwerpunkt soll auf der Berücksichtigung von Echtzeitaspekten liegen, um mit dieser Architektur eine Grundlage für harte Echtzeitanwendungen zu schaffen. War früher Echtzeit in diesem Bereich nur auf dedizierten Teilsystemen verbreitet, soll dies nun auf Anwendungen auf Standardrechnern ausgedehnt werden. Da die Hard- und Softwarekomponenten von Standardrechnern für eine gute durchschnittliche Verarbeitungsleistung optimiert werden, bleiben sporadisch auftretende längere Laufzeiten oft unzureichend berücksichtigt. Als Plattform für diese Architektur muss in dieser Arbeit daher das vollständige System mit allen ausgewählten Hard- und Softwarekomponenten untersucht werden, um darauf aufbauend die harte Echtzeitfähigkeit zu gewährleisten.

Da jedoch nicht vorausgesetzt werden darf, dass alle zu erprobenden Funktionen schon von Beginn an hart echtzeitfähig entwickelt werden, müssen zudem Methoden gefunden werden, diese parallel so zu betreiben, dass sie wichtige zeit- und sicherheitskritische

Funktionen nicht blockieren können. Unter Verwendung dieser Methoden soll dann eine robuste Überwachungs- und Sicherheitsebene geschaffen werden. Um bestehende Module hart echtzeitfähig zu machen, müssen geeignete Regeln für deren Entwicklung aufgestellt werden. Beim Übergang in den Echtzeitbetrieb darf auf Architekturseite jedoch kein Umstellungsaufwand anfallen.

Ein weiteres wichtiges Ziel dieser Arbeit wurde durch den interdisziplinären Projektrahmen vorgegeben. Dieser überträgt der Architektur eine gewisse Integrationsmitverantwortung. So musste eine Lösung gefunden werden, die von allen beteiligten Wissenschaftlern der verschiedenen Fachrichtungen akzeptiert werden konnte. Da hinter jedem Softwaremodul ein Entwickler steht, bekommt die tragende Integrationsarchitektur eine Vermittlerfunktion. Geeignete Vorgaben zur Schnittstellendefinition können hier den Konsens auf technischer Ebene fördern.

1.3 Gliederung

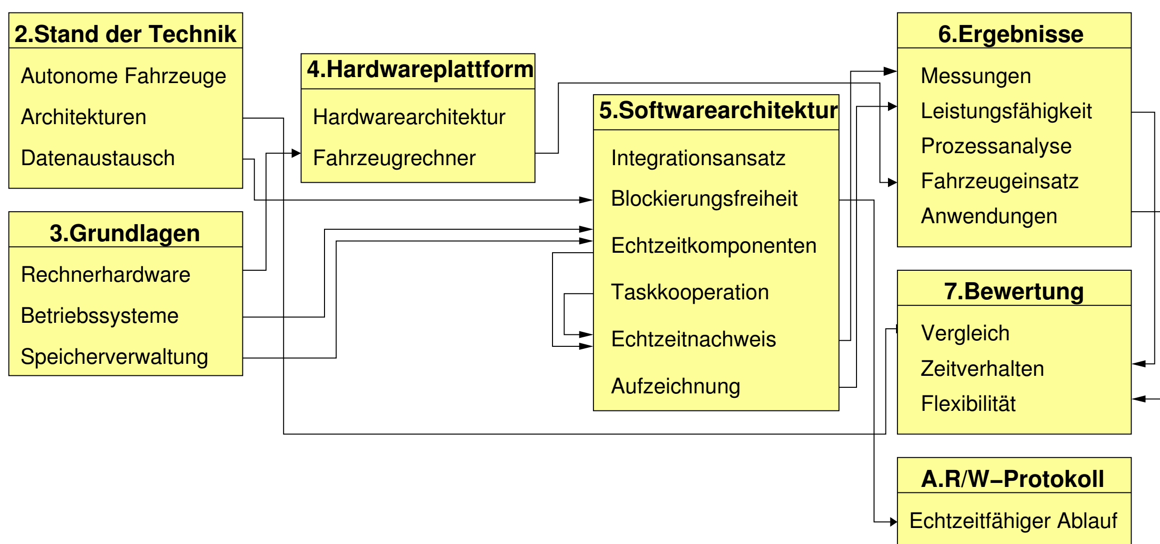


Abbildung 1.1: Kapitelzusammenhang der behandelten Thematiken

Kapitel 2 gibt einen Überblick über den aktuellen Stand der Forschung zu wichtigen von dieser Arbeit berührten Themengebieten. Dies sind zum einen Hardware- und Softwarearchitekturen von intelligenten Automobilen, darunter auch in Abschnitt 2.2 die Fahrzeuge eines Wettbewerbs, auf den im Ergebniskapitel 6.3.3 erneut Bezug genommen wird, da Ergebnisse dieser Arbeit dort zum Einsatz kamen. Zum anderen werden wissenschaftliche Arbeiten zum Thema der nicht-blockierenden Datenaustauschprotokolle vorgestellt, zu dem im späteren Kapitel 5.3 ein Beitrag geleistet wird.

Die in Kapitel 3 gelieferten Grundlagen dienen dem Verständnis darauffolgender Kapitel. So basiert die Entscheidung für das Fahrzeugrechnersystem aus Kapitel 4.3 auf den

Eigenschaften von PC-Hardware von Abschnitt 3.3. Die Grundlagen zu Realzeitbetriebssystemen aus Kapitel 3.4 werden für die Komponentenauswahl in Kapitel 5.4 vorausgesetzt. Auf Abschnitt 3.5 schließlich stützt sich die echtzeitfähige Speicherverwaltung in Kapitel 5.3.7.

In Kapitel 4 wird eine Hardwareplattform für kognitive Fahrzeuge vorgestellt, die in den Versuchsträgern aus Kapitel 6.4 zum Einsatz kommt. Der Schwerpunkt liegt dabei auf der Architektur des Fahrzeugrechnersystems.

Den Hauptteil dieser Arbeit bildet Kapitel 5 mit der echtzeitfähigen Softwarearchitektur. Die in Abschnitt 5.3 entwickelten Datenaustauschmethoden und die im Unterkapitel 5.4 entworfene Softwarearchitektur machen die echtzeitfähige Kooperation von Rechenprozessen in Kapitel 5.5 erst möglich. Dies erlaubt schließlich den Echtzeitnachweis, der durch Messungen in Kapitel 6.1 bestätigt wird. Die Aufzeichnungsmethoden aus Kapitel 5.7, deren Leistungsfähigkeit Abschnitt 6.2 belegt, haben sich zu einer gefragten Eigenschaft dieser Architektur entwickelt.

Kapitel 6 liefert zum einen Messergebnisse, die mit der entwickelten Hard- und Softwarearchitektur erzielt wurden, und belegt damit deren Echtzeitfähigkeit. Zum anderen werden Anwendungen präsentiert, die die Praxistauglichkeit und den weiten Einsatz der Ergebnisse dieser Arbeit anschaulich demonstrieren. Die automatisierte Prozessanalyse in Abschnitt 6.3 hilft mit Messdaten bei fehlenden Eingangsdaten für den Echtzeitnachweis in Kapitel 5.5.9.

In Kapitel 7 werden die Ergebnisse dieser Arbeit zusammengefasst und mit dem Stand der Technik verglichen. Wichtige Punkte dabei sind insbesondere das Zeitverhalten und die Flexibilität der Architektur sowie der enthaltene Beitrag zum blockierungsfreien Datenaustausch.

Anhang A erläutert abschließend in einer ausführlichen Fassung den genauen Ablauf des in Abschnitt 5.3.5 vorgestellten Schreib-/Leseprotokolls für den echtzeitfähigen Datenaustausch.

2 Stand der Technik

Dieses Kapitel gibt einen Überblick über den aktuellen Stand der Technik zu den von dieser Arbeit berührten Themengebieten. Da der Schwerpunkt der zu integrierenden kognitiven Funktionen durch das bearbeitete Forschungsprojekt auf Automobilen liegt, werden vor allem relevante Arbeiten aus diesem Bereich herangezogen. Ergänzend werden zudem Arbeiten aus dem weiteren Feld der Robotik hinzugenommen. Der Schwerpunkt der Beschreibungen liegt stets auf der Hardware-/Softwarearchitektur, auf dem Datenfluss und dem Zeitverhalten, da diese die für spätere Kapitel relevanten Aspekte darstellen. Schließlich werden wissenschaftliche Arbeiten zum sehr spezifischen Thema der nicht blockierenden Datenaustauschprotokolle angeführt.

2.1 Architekturen von (autonomen) Forschungsfahrzeugen

Die ersten Kraftfahrzeuge, die mit kognitiven Teilfunktionen versehen waren, dienten vorrangig der Demonstration einzelner neuartiger Fähigkeiten. Diese Fähigkeiten waren oft fest verdrahtet und nutzen dieselben Ressourcen, sodass sie nur getrennt vorgeführt werden konnten. Um jedoch mehrere Funktionen gleichzeitig im selben System einsetzen zu können, begann die Entwicklung geeigneter Architekturen.

Erfahrungsgemäß bestehen in der Praxis Anforderungen und treten Probleme auf, die in der Theorie gerne ausgeklammert werden, aber letztendlich bei einer Realisierung entscheidungsrelevant sind. Daher werden nur solche Ansätze miteinbezogen, die ihre Praxistauglichkeit in realen Versuchsfahrzeugen bewiesen haben. Reine Simulationen können schwer alle Randbedingungen mit abbilden und bieten somit nur begrenzte Aussagekraft.

Die folgenden Untersuchungen erheben keinen Anspruch auf Vollständigkeit und sollen keine umfassende Übersicht geben. Einen historischen Überblick über die Entwicklung „sehender“ Fahrzeuge gibt z. B. [39]. Vielmehr werden gezielt einzelne Lösungen und Architekturen herausgegriffen und in der Tiefe näher untersucht. Dabei werden einige gut publizierte und repräsentative Lösungen ausgewählt.

In der folgenden Auswahl werden mit EMS-Vision und ANTS zwei benachbarte Ansätze herausgegriffen und näher betrachtet, die mit der neu entwickelten Architektur das Konzept einer gemeinsamen Datenbank bzw. Datenbasis teilen. Dabei werden besonders die Details herausgearbeitet, die Unterschiede zu der später präsentierten Architektur aufweisen.

Eine besondere Rolle spielen die auf der DARPA Urban Challenge verwendeten Architekturen. Zum einen ist durch die Publikationen der Finalistenteams im *Journal of Field Robotics* eine umfassende Darstellung der aktuell verwendeten Ansätze verfügbar. Zum anderen wurde die in dieser Arbeit vorgestellte Architektur in einem der 11 Fahrzeuge eingesetzt, die das Finale erreicht haben.

Einen guten Überblick über die aktuelle Entwicklung von intelligenten Fahrzeugen und deren Industrie-nähere Ergebnisse in Form von Fahrerassistenzsystemen geben u. a. die Konferenzen und Workshops *IEEE Intelligent Vehicles* (Tochterkonferenz der *Intelligent Transportation Systems*), *Autonome Mobile Systeme* und *Workshop Fahrerassistenzsysteme*, auf denen jeweils diese Architektur in Publikationen bereits vorgestellt wurde.

2.1.1 VaMoRs und VaMP (UniBW-M/ISF)

(a) Außenansicht (b) Innenansicht

Abbildung 2.1: Versuchsträger VaMP (Versuchsfahrzeug für autonome Mobilität und Rechnersehen im PKW)

An der Universität der Bundeswehr München wurden am Institut für Systemdynamik zwei Versuchsträger betrieben. Das ältere Fahrzeug war VaMoRs (Versuchsfahrzeug für autonome Mobilität und Rechnersehen), ein Mercedes-Benz Kastenwagen vom Typ 508D (Bj. 1983) mit Automatikgetriebe. Er verfügte über eine zweiachsige multifokale Kameraplattform [169] und war mit einem zusätzlichen 10kW-Dieselgenerator ausgestattet [42, 183].

Das jüngere Fahrzeug, VaMP (VaMoRs im PKW), war ein Mercedes-Benz 500SEL (Bj. 1992) mit Automatikgetriebe. Er verfügte über zwei einachsige bifokale Kameraplattformen nach vorne und hinten, sowie einen nach vorne gerichteten 77 GHz RADAR-Sensor und eine zusätzliche 24V Lichtmaschine. Beide Fahrzeuge waren mit elektronisch ansteuerbarer Lenkung, Gas und Bremse ausgestattet, die bis auf E-Gas beim 500SEL alle nachgerüstet wurden. Zudem verfügten sie über Inertialsensoren und GPS-Empfänger [137, 138].

Beide Versuchsträger wurden über die Jahre mit verschiedenen Rechnersystemen ausgestattet. VaMoRs wurde anfangs mit BVV 2 [79] betrieben, ein selbst entwickeltes System aus 15 Intel 8088 Prozessoren zur Bildverarbeitung, zusammen mit einem Intel 80286 Rechner zur Fahrzeugführung. Damit war bereits eine autonome Längs- und Querführung möglich [44, 43].

Später wurde VaMoRs mit einem Transputercluster aus 60 Knoten mit 16 und 32 Bit umgerüstet, davon 45 Knoten zur Bildverarbeitung und der Rest zur Fahrzeugführung und Kameraplattformkontrolle. Mit dieser Architektur wurde auch VaMP initial aufgebaut und demonstrierte z. B. beim EUREKA-Verbundprojekt PROMETHEUS autonomes Fahren mit Spurwechsel auf einer dreispurigen Autobahn [42]. Danach wurde VaMP mit Motorola PowerPC 601 Prozessoren ausgebaut [137].

EMS-Vision System

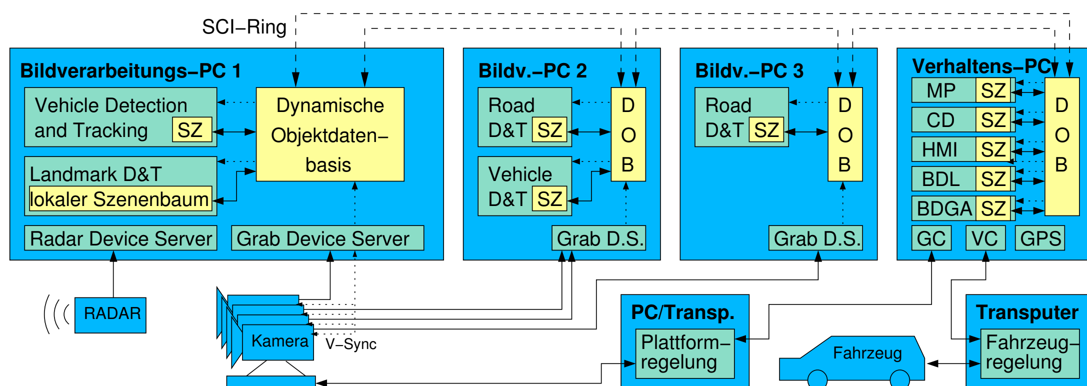


Abbildung 2.2: Architektur des *EMS-Vision Systems* (erstellt in Anlehnung an [165])

Die letzte Architekturgeneration war ein System mit dem Namen EMS-Vision [80, 165, 40], über dessen Architektur Abb. 2.2 einen Überblick gibt. Es bestand aus 2-4 Standard-PCs mit je 1-2 Intel Pentium II/III Prozessoren, die inzwischen leistungsfähiger als die Transputerknoten waren. Vernetzt wurden diese mit 100 MBit/s Ethernet zur Kontrolle und einem SCI-Ring (Scalable Coherent Interface) zum Datenaustausch mit einer nominalen Bandbreite von 90 MB/s.

Als Betriebssystem wurde Windows-NT ausgewählt, dessen gute Entwicklungsumgebung, Treiberunterstützung und Verbreitung, sowie das günstigste Preis-Leistungsverhältnis den Ausschlag gaben. Dabei entschied man sich trotz der damals unklaren Echtzeiteigenschaften von Windows-NT gegen die ausgewiesenen Echtzeitbetriebssysteme LynxOS und VxWorks [165]. Da beim Einsatz des 4D-Ansatzes damit keine schlechten Erfahrungen gemacht wurden, wurde diese Konfiguration beibehalten. Auf einem kleinen Transputercluster zwischen den PCs und dem Fahrzeug wurden alle echtzeitkritischen Fahrzeugregelkreise mit ihren unterschiedlichen Zykluszeiten geschlossen.

Alle Wahrnehmungsmodule, wie z. B. ein Spurerkenner (Road D&T), wurden als eigene Prozesse implementiert, die von einzelnen Bearbeitern mit den jeweiligen Spezialkenntnissen entwickelt wurde. Durch die Organisation als nebenläufige Prozesse konnten diese Einzelfähigkeiten kombiniert werden.

Dynamische Objektdatenbasis

Auf dem vorhergehenden Transputersystem existierte zwar schon eine gemeinsame Datenbasis, der Großteil der Kommunikation wurde mit individuellen Protokollen direkt zwischen Prozessen abgewickelt. Weil dort schon bei der Implementierung die Verteilung der Prozesse auf Prozessoren festgelegt wurde, konnten nicht alle Fähigkeiten gleichzeitig genutzt werden [165].

Im EMS-Vision System hingegen erfolgt jede Kommunikation zwischen zwei Modulen über die eingeführte *dynamische Objektdatenbasis* (DOB). Jeder Prozess empfängt daraus seine benötigten Informationen, ohne wissen zu müssen, woher sie kommen, und speist seine Ergebnisdaten wieder ein.

Die Inhalte der DOB sind in Objekte gegliedert, die voneinander separierbare Einheiten darstellen, wie der Sensoreinbauort oder ein erkanntes Objekt der Umgebung. Objekte werden aus einer Menge von Basisklassen abgeleitet, die dem 4D-Ansatz [38] entsprechen. Sie sind in einer Baumstruktur organisiert, sodass sie einen Szenenbaum [37] bilden, in dem jedes Objekt seine Relativlage zu seinem Vaterknoten in homogenen Koordinaten kennt.

Datenaustausch in EMS-Vision

Auf jedem Rechner läuft eine Instanz der DOB. Jeder Prozess kann beim Verbindungsaufbau mit der DOB angeben, über welche Klassen von Objekten er informiert werden will. Diese bekommt er dann als Kopie in seinen lokalen Szenenbaum eingehängt und kann sie ändern, erweitern und wieder publizieren. Die Programmierer müssen darauf achten, dass jedes atomare Datenfeld in einem Knoten nur von einem Prozess gleichzeitig beschrieben wird, da dies nicht vom System überwacht wird. Zum Konsistenzerhalt bei der Änderung mehrerer Datenfelder kann der Datenversand programmgesteuert verzögert werden [165].

Das EMS-Vision System arbeitet mit einem Systemtakt, der *Zyklus* genannt wird, mit dem alle verteilten Datenbanken synchronisiert werden. Zu Beginn eines Zyklus tauschen erst die DOB-Instanzen auf jedem Rechner ihre Daten über SCI aus, dann werden die neuen Daten auf die lokalen Teilkopien der einzelnen Prozesse verteilt. So stehen die Ergebnisse eines Prozesses anderen Prozessen zu Beginn des nächsten Zyklus zur Verfügung. Da der Kontrollrechner über keinen *Framegrabber*(Videobildeinzug) verfügt, wird er über den Zyklusbeginn von anderen Rechnern informiert. Lediglich bei großer Auslastung kam es dabei zu Verzögerungen beim Datenaustausch [165].

Für einen Zyklus wurde, da die wichtigsten Sensoren der Versuchsträger Videokameras waren, ein 40 ms Takt festgelegt, der durch das V-Sync Signal der Kameras ausgelöst wird. Dieses Signal wird vom Framegrabber-Prozess auf jedem Rechner an die dort laufende DOB-Instanz weitergereicht. Damit dies auf allen Rechnern gleichzeitig geschieht, wurden die Kameras untereinander durch entsprechende Hardwarevorrichtungen synchronisiert.

Für eine bessere Zeitauflösung wurden die Uhren der PCs untereinander mit einer Auflösung von 1 ms synchronisiert. Dabei wurden auch auf den Windows-NT Systemen auftretende Unterbrechungen der Synchronisationsprozesse durch Ablauf ihrer 10 ms-Zeitscheiben berücksichtigt [165].

Die Systemarchitektur des EMS-Vision System hat viel dazu beigetragen, Einzelfähigkeiten zu einem System zu integrieren. Sie hat die Grundlagen einer vereinheitlichten Szenenrepräsentation und der einheitlichen Kommunikation über eine dynamische Objektdatenbasis geschaffen. Der Schwerpunkt der eingesetzten Wahrnehmungsmodule liegt auf der Auswertung von Videobildern nach dem 4D-Ansatz.

2.1.2 VITA und UTA (Daimler)

Bei der Daimler-Forschung (ehem. Daimler-Benz bzw. DaimlerChrysler) wurde im Rahmen des PROMETHEUS-Projekts ein Schwesterfahrzeug zu VaMP namens VITA II (Vision Technology Application) aufgebaut. Es war ebenfalls für Autobahnfahrten spezialisiert und wurde auch 1994 in Paris bei der PROMETHEUS-Abschlussdemonstration vorgeführt. Es konnte Fahrspuren, Verkehrszeichen und andere Verkehrsteilnehmer erkennen und demonstrierte Spurhaltung, Abstandhalten und automatischen Spurwechsel [210].

Das Rechnersystem von VITA II bildeten maßgeblich drei Rechner: Der Fahrzeugrechner zur Regelung des Fahrzeugs und der Kameraplattformrechner zur Stabilisierung und Steuerung der Kameraplattform bestanden aus einem Netzwerk von Transputerknoten. Der Bildverarbeitungsrechner bestand aus einem Transputernetzwerk sowie TMS 320 C40 Signalprozessoren und MPC 601-Prozessoren. Darauf kam ebenfalls eine gemeinsame *dynamische Datenbasis* (DDB) als Schnittstelle zwischen den Sensormodulen und dem Kontrollsystem zum Einsatz [210, 164]. Zur Übertragung von Videobildern waren die Transputer-Links zu langsam, daher wurde ein spezieller *Transputer Image-Processing* (TIP)-Bus mit 100 MB/s eingesetzt. Ein eigens entwickeltes Werkzeug (TRAPPER) [89] hilft bei der optimalen Verteilung der ca. 200 Rechenprozesse in der Designphase.

Die folgenden Versuchsträger UTA (Urban Traffic Assistant) und UTA II waren für Innenstadtszenarien konzipiert. Dabei treten im Gegensatz zur Autobahnumgebung zwar geringere Geschwindigkeiten auf, die Komplexität der Umwelt ist durch verschiedenste Verkehrsteilnehmer wie Radfahrer und Fußgänger jedoch deutlich höher [55, 54, 53].

Die wesentlichen Sensoren von UTA II, einem Mercedes E430, sind ein Stereo-Kamerasystem und eine Farbkamera sowie ein RADAR-Sensor. Die Aktoren Gas, Bremse und Lenkung werden von einem PowerPC 604e mit 333 MHz unter dem Echtzeitbetriebssystem

LynxOS geregelt. Dieser kommuniziert über Fast-Ethernet mit drei Anwendungsrechnern (Dualprozessor (SMP) Intel Pentium II mit 700 MHz) unter dem Standardbetriebssystem Linux [84, 61].

Agent Network System (ANTS)

Für die Versuchsträger UTA und UTA II wurde eine neue Softwarearchitektur namens „Agent NeTwork System“ (*ANTS*) [85, 86, 84] entworfen. ANTS definiert sich als Softwarearchitektur für Multi-Agentensysteme, da es ermöglicht, dass verschiedene Funktionseinheiten wie Agenten in einem Netzwerk zusammenarbeiten und zusammen Probleme lösen, die sie alleine nicht bewältigen könnten. ANTS dient dazu, Softwaremodule untereinander zu vernetzen und dynamisch zur Laufzeit zu konfigurieren. Einen Schwerpunkt bildet dabei das verteilte Rechnen auf mehreren Rechnersystemen.

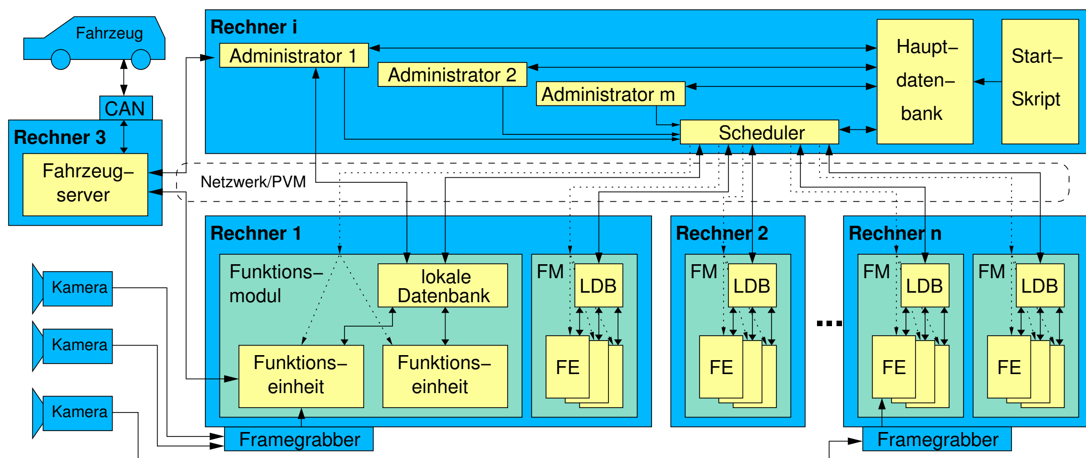


Abbildung 2.3: Architektur des *Agent NeTwork System* (verändert entnommen aus [84])

Die Struktur der Architektur von ANTS nach [84] skizziert Abb. 2.3: Alle Daten werden in *Datenbanken* gespeichert. *Funktionseinheiten* enthalten die eigentlichen Algorithmen und werden in *Funktionsmodulen* zusammengefasst. *Administratoren* bestimmen, welche Funktionseinheiten vom *Scheduler* ausgeführt werden.

Das Verständnis für das Gesamtsystem erschließt sich ab besten durch die im Folgenden wiedergegebenen Aufgaben und Eigenschaften der gezeigten Komponenten [84]:

Hauptdatenbank: In der Hauptdatenbank wird das Weltwissen der Agenten festgehalten und so Algorithmik und Datenrepräsentation getrennt. Die Definition der Datenbankeinträge ist relativ frei, vorausgesetzt werden die Ableitung von einer Datenbank-Basisklasse und die Serialisierbarkeit für die Verteilung über ein Netzwerk.

Die Daten werden uninterpretiert eingetragen und haben so keine Semantik außerhalb der Verarbeitungsmodule. Es wird absichtlich auf ein einheitliches Datenformat

wie z. B. XML verzichten, um das Kommunikationsnetzwerk nicht durch eine allgemeine Struktur zu belasten. Zudem soll keine Vollständigkeit durch Durchgängigkeit vorgetäuscht werden.

Die Persistenz der Daten ist auf die Laufzeit beschränkt, zur Beobachtung können jedoch alle augenblicklich enthaltenen Daten zu diskreten Zeitpunkten abgespeichert werden.

Lokale Datenbanken: Die verteilte Struktur der Datenbanken weist eine zweistufige Hierarchie aus einer Hauptdatenbank und beliebig vielen *lokalen Datenbanken* auf. Alle Administratoren arbeiten auf der Hauptdatenbank, haben aber auch Zugriff auf die einzelnen lokalen Datenbanken der jeweiligen Funktionsmodule durch die Verwendung von *entfernten Transaktionen*.

Die *Administratoren* bestimmen, welche Daten an die lokalen Datenbanken geliefert werden und welche zurückerwartet werden. Ein Funktionsmodul arbeitet stets auf einer lokalen Datenbank, die nur die benötigten Daten enthält. Zurückgeholt werden Daten durch entfernte Transaktionen der Administratoren oder durch den Scheduler am Ende eines Zyklus.

Damit beim gleichzeitigen Schreibzugriff auf die Datenbank keine Inkonsistenzen auftreten, werden exklusive Sperren eingesetzt. Die Dauer einer Schreibtransaktion wird durch die Funktionsmodule bestimmt, für die es prinzipiell keine Zeitbegrenzung gibt. So dauert z. B. die Verkehrszeichenerkennung aus Farbbildern mehrere 100 ms. Der gleichzeitige Lesezugriff ist jederzeit möglich, die maximale Transaktionsdauer ist nicht angegeben.

PVM-Kommunikationsnetzwerk: Zur Kommunikation zwischen zwei Prozessen auf beliebigen Rechnern wird eine auf PVM aufbauende Lösung namens CPPvm eingesetzt. PVM (*Parallel Virtual Machine*) stellt Grundfunktionen zur transparenten Kommunikation in heterogenen Rechnernetzen zur Verfügung. CPPvm (*C Plus Plus Pvm*) [83], eine Neuimplementierung von PVMCPP, ist eine C++ Klassenbibliothek, die auf PVM aufsetzt und das explizite Versenden und Empfangen von C++ Objekten implementiert.

Vom Einsatz von CORBA wurde wegen seines Verwaltungs- und Protokolloverheads abgesehen, da PVM damals über ein 10 MBit/s Netz bei 10-100 kB Blöcken 50-250 mal schneller als CORBA war [84]. PVM bietet im Vergleich zu CORBA nur Datenzugriff auf verteilte Objekte und keine Methodenaufrufe. Der Abgleich einer lokalen Instanz eines Objektes geschieht durch einen Synchronisierungsaufruf.

Administratoren: In den Administratoren wird entschieden, welche Funktionseinheiten mit welchen Aufrufparametern als Nächste zu starten sind. Dazu dienen hintereinandergeschaltete Filter mit einem Entscheider am Ende. Beispielsweise greift ein Filter in einem Administrator für Bildverarbeitung auf die Hauptdatenbank zu, um zwischen Autobahn und Innenstadt zu unterscheiden und den passenden Spurerkenner mit den geeigneten Parametern zu wählen.

Der Einsatz eines Agentenverhandlungsprotokolls wie Contract-Net [188] zur Auswahl der benötigten Funktionseinheiten brachte keine Vorteile, eine einfache Ablaufsteuerung wurde als ausreichend befunden.

Funktionsmodule und -einheiten: Eine *Funktionseinheit* enthält einen Algorithmus wie z. B. einen Objekterkenner, der auf die lokale Datenbank zugreift. Ein *Funktionsmodul* ist die Laufzeitumgebung ein oder mehrerer Funktionseinheiten und ist als ein Prozess implementiert. Die Aufteilung in einzelne Funktionseinheiten sollte wegen der erforderlichen Kommunikation mit mittlerer Granularität geschehen.

Ein Grund für die Zusammenfassung mehrerer Funktionseinheiten zu einem Funktionsmodul war der gleichzeitige Zugriff auf dieselben Betriebsmittel. So mussten z. B. Bildverarbeitungsmodule wie die Ampel- und Verkehrszeichenerkennung mit dem Einzug von Videobildern zusammengefasst werden, da das verwendete Betriebssystem nur einem Prozess die Kontrolle über die Framegrabber-Hardware erlaubte. Da der Datenaustausch von 384x288 Pixel großen Stereobildern via PVM über ein 10 MBit/s Netzwerk 100-150 ms dauerte, konnten diese auch nicht über die Hauptdatenbank verteilt werden.

Kurze Verbindungswege für schnelle Regelzyklen waren ein weiterer Grund für die Zusammenlegung einer Objektverfolgung mit einem Deichselregler zu einem Funktionsmodul.

Scheduler: Der *Scheduler* übernimmt als zentrale Ablaufsteuerung in jedem Zyklus den Aufruf der Funktionseinheiten so, wie es durch die Administratoren vorgegeben wurde, und gleicht zu Begin und am Ende die lokalen Datenbanken ab. Dazu schickt er ein Signal an die Ablaufsteuerung des einbettenden Funktionsmoduls mit den Aufrufparametern und dem Ausführungsmodus jeder enthaltenen Funktionseinheit.

Der *Scheduler* selbst wird vom ANTS-Hauptprogramm, das in Abb. 2.3 auf einem beliebigen PC i läuft, nach jedem Durchlauf eines Administrators aufgerufen. Die Zykluszeit für den Aufruf der Administratoren nacheinander kann vorgegeben werden. Da in den durchgeführten Versuchen die Ausführungszeit des Hauptprogramms mit den Administratoren kürzer als die Berechnungen der Funktionseinheiten war und normalerweise nur Daten auf symbolischer Ebene übertragen wurden, führte die zentrale Ablaufsteuerung zu keinem Engpass.

Server: Auf Server können Funktionseinheiten und Administratoren direkt zugreifen, im Gegensatz zur indirekten Kommunikation über die Datenbanken. Dies ist zum einen aus Geschwindigkeitsgründen für den direkten Ressourcenzugriff nötig, da sonst die Beendigung des Zyklus weiterer Funktionseinheiten im selben Funktionsmodul abgewartet werden muss.

Zum anderen werden für die schnelle Abarbeitung zeitkritischer Abläufe auf diese Weise Fahrbefehle an den Fahrzeugserver ohne den „Umweg“ über die Datenbank gegeben. Zur Gewährleistung der Echtzeitfähigkeit der Regelung wird als Fahrzeugserver ein separater PowerPC unter dem Echtzeitbetriebssystem LynxOS eingesetzt.

Script-Sprache: Durch die eingebaute Script-Sprache werden bei der Initialisierung des Systems die zu verwendenden Module ausgewählt und konfiguriert.

Außerdem wird der Aufbau der Datenbanken festgelegt, indem alle benötigten Objekte angelegt werden und ihnen eindeutige Element-Identifikationsnummern (IDs) durch ein globales ID-Skript zugewiesen werden.

ANTS wurde erfolgreich in den Versuchsträgern UTA und UTA II für die Umsetzung verschiedener Fahrerassistenzfunktionen wie Spurverlassenswarner, Tempomat mit Halt vor Stopp-Schild, Stop&Go, elektronische Deichsel, uvm. eingesetzt. Die entworfene Struktur, bestehend aus flexibel einsetzbaren Modulen, die einen Großteil ihre Daten über eine verteilte Datenbank austauschen, hat sich in ANTS als tragfähig erwiesen. Aus Geschwindigkeitsgründen wurde kein vorhandenes Datenbanksystem verwendet, sondern eine Eigenentwicklung gewählt.

Im Ausblick von [84] werden u. a. als zukünftige Entwicklungsrichtungen die Ausdehnung auf weitere Anwendungsfelder, eine Modifikation für den Einsatz auf eingebetteten Systemen und die Erfüllung harter Echtzeitanforderungen genannt. Unter dem Namen ANTSRT [170] wurde auf einer RTAI-Linux-Plattform eine Überarbeitung durchgeführt, um die Integration von echtzeitfähigen Reglersystemen zu ermöglichen. Es wird vom prototypischen Einsatz in einer Stop&Go-Anwendung berichtet und eine Verbesserung der Kommunikation für die verteilte Synchronisation angekündigt.

2.2 Autonome Fahrzeuge im DARPA Grand Challenge

Die DARPA (*Defense Advanced Research Projects Agency)* ist eine Forschungsbehörde des amerikanischen Verteidigungsministeriums. Sie hat vom US-Kongress den Auftrag, dafür zu sorgen, dass zum Jahr 2015 ein Drittel der militärischen Fahrzeuge unbemannt fahren. Um die Forschung auf diesem Gebiet zu stimulieren, hat sie 2004 einen ersten Wettbewerb namens Grand Challenge veranstaltet. Dabei mussten die teilnehmenden Fahrzeuge autonom in einem Wüstengebiet vorgegebene GPS-Punkte abfahren. Beim ersten Rennen erreichte kein Team das Ziel.

Erst beim zweiten Grand Challenge 2005 wurde diese Aufgabe von mehreren Fahrzeugen erfüllt. Das Siegerfahrzeug, ein modifizierter VW Touareg der Stanford Universität, fuhr 211 km in unter 7 Stunden, dicht gefolgt von Sandstorm und H1ghlander der Carnegie Mellon Universität [32].

2.2.1 DARPA Urban Challenge

Am 3.11.2007 fand der dritte Wettbewerb, die DARPA Urban Challenge [33], in Victorville, Kalifornien, USA, auf dem Kasernengelände des ehemaligen Luftwaffenstützpunkts George Air Force Base statt. Der Herausforderung lag dabei auf städtischer Umgebung und in der Interaktion mit andern Fahrzeugen,

In dem Rennen musste ein 97 km (60 Meilen) langer, durch GPS-Punkte vorgegebener Parcours durch die Straßen des bebauten Kasernengebiets in unter 6 Stunden abgefahren werden. Zudem mussten beim Zusammentreffen mit anderen Roboterautos oder von Menschen gelenkten Fremdfahrzeugen die kalifornischen Verkehrsregeln eingehalten werden, wie z. B. das Einhalten der Reihenfolge am sog. *Four-Way-Stop*.

Für die Urban Challenge hatten sich 89 Teams aus der Industrie und der universitären Forschung angemeldet, von denen 36 zum National Qualification Event nach Viktorville eingeladen wurden. Dort mussten sie auf drei Testflächen in einem mehrtägigen Qualifizierungsprogramm ihre Fähigkeiten zeigen und insbesondere beweisen, dass sie zu jeder Zeit ein sicheres Verhalten aufweisen. Nur 11 Teams konnten sich dabei für das finale Rennen qualifizieren.

Das Besondere an den DARPA Challenges ist, dass bei jedem Wettbewerb ein klares Ziel im Vordergrund stand. Daher durften nicht nur einzelne Fähigkeiten vorgeführt werden, sondern es musste jeweils ein funktionsfähiges Gesamtsystem entwickelt werden, das zudem auch noch robust und sicher ist, da keine Sicherheitsfahrer erlaubt waren. Lediglich einen von der DARPA auslösbaren Notstopp musste jedes Fahrzeug besitzen.

Durch den Wettbewerb fand zudem ein öffentlicher Vergleich aller Lösungen unter genau definierten Randbedingungen statt. Da die Menge der benötigten Fähigkeiten zu Entwicklungsbeginn bekannt waren, waren die Anforderungen an die Architektur anders als an einen reinen Versuchsträger. Für die Wettbewerbsfahrzeuge waren teils weniger generische Architekturen ausreichend, die dadurch weniger Overhead aufweisen. Bei einem Versuchsfahrzeug der Forschung müssen die eingesetzten Architekturen offen genug sein, um zu jedem Zeitpunkt neue Fähigkeiten und Funktionen zu integrieren, von denen man zum Zeitpunkt des Aufbaus des Versuchsträgers noch nichts wusste.

Die Anforderungen und Regeln der DARPA Urban Challenge waren im Wesentlichen:

- **Navigieren auf vorgegebenem Streckennetz (städtische Umgebung):**
  - Digitale Karte aus GPS-Punkten incl. Kreuzungen (RNDF) gegeben
  - Mission ist Liste von Checkpunkten und erlaubten Geschwindigkeiten (MDF)
- **Hindernisvermeidung:**
  - Umfahren statischer Hindernisse
  - Sicherheitsabstand beim Folgefahren
- **Einhalten der kalifornischen Verkehrsregeln:**
  - Abbiegen bei Gegenverkehr
  - Einfädeln in fließenden Verkehr
  - Vorfahrtsregeln an Kreuzungen (4-Way-Stop)
- **Routenplanung:**
  - Erkennung von Straßenblockaden
  - Wendemanöver (U/K-Turn)
- **Einparken in eine freie Parklücke**

- Sicheres Verhalten ohne jegliche Gefährdung

Im Folgenden werden die Architekturen einiger Fahrzeuge präsentiert. Für eine erschöpfende Darstellung sei an dieser Stelle auf die Beiträge aller Finalisten im *Journal of Field Robotics* verwiesen.

2.2.2 Tartan Racing (Carnegie Mellon University)

(a) Seitenansicht (b) Frontansicht

Abbildung 2.4: Das autonome Fahrzeug „Boss" des Tartan Racing Teams

Den ersten Platz belegte das Fahrzeug „Boss" des Tartan Racing Teams mit maßgeblicher Beteiligung der Carnegie Mellon University [35]. Boss ist ein modifizierter Chevrolet Tahoe (Bj 2007), in den ein kommerzielles Drive-by-Wire System eingebaut wurde und der eine zusätzliche 6 KW 24 V-Lichtmaschine für die Computersysteme sowie zur Speisung eines 120 V-Wechselspannungsnetzes besitzt. Ein hart echtzeitfähiges Steuergerät ist für die unterlagerte Fahrzeugregelung zuständig [212, 211].

Das Rechnersystem von Boss besteht aus einem CompactPCI Gehäuse mit 10 Rechnereinschüben mit je einem 2.16 GHz Core2Duo-Prozessor, 2 GB Speicher, zwei Gigabit-Ethernet-Schnittstellen und einer 4 GB Flash-Festplatte. Zwei Rechner sind zusätzlich mit jeweils einer 500 GB Festplatte zum Mitprotokollieren ausgestattet. Die Zeitsynchronisierung der einzelnen Rechner untereinander erledigen selbst gebaute Erweiterungsplatinen, die jede Sekunde ein Synchronisierungssignal austauschen.

Bei der Sensorauswahl wurden bevorzugt aktive Messsensoren wie LIDAR und RADAR verwendet, die eine direkte Messung von Entfernung und Geschwindigkeit eines Zielobjekts erlauben. Dabei wurde auf vollständige Abdeckung mit Redundanz geachtet. Videobilder wurden lediglich zur Fahrspurschätzung eingesetzt, die mit 10 Hz aktualisiert wurde und sich zusätzlich auf LIDAR-Messungen stützte. Begründet wird dies damit, dass ein Videobild zwar detailreicher, aber schwieriger zu interpretieren ist.

Zusätzlich wurden zwei bewegliche Sensorplattformen eingesetzt, die jeweils aus einem RADAR und LIDAR-Sensor bestanden. Vorne links und rechts montiert dienen sie v. a.

der Vorfahrbestimmung an Kreuzungen. Dazu wurden sie vom *Pan-Head Planner* automatisch in die entsprechende Richtung ausgerichtet.

Softwarearchitektur von Boss

Die Softwareinfrastruktur des *Tartan Racing Urban Challenge Systems* (TRUCS) [212] ist das Ergebnis der Erfahrungen aus den ersten beiden Grand Challenges und weiteren Felderfahrungen des Teams. Der Schwerpunkt liegt u. a. auf folgenden Qualitätsmerkmalen [140]:

Flexibilität des Informationsflusses: Die Fähigkeiten der Wissensverarbeitung wie der Entscheidungsfindung können es notwendig machen, dass an unerwarteten Stellen bestimmte Informationen benötigt werden. Daher sollten alle Informationen jederzeit verfügbar sein. In TRUCS ist eine Programmierschnittstelle vorhanden, mit der jederzeit nach neuen Eingangsdaten gefragt werden kann (*Polling*).

Unabhängigkeit der Tasks: Da TRUCS aus Dutzenden von Tasks besteht, sollten beim Absturz eines Tasks die anderen nicht einfrieren, sondern weiterlaufen. Daher sollte die Architektur synchrone Kommunikation meiden, die z. B. nach einem *Call/Return*-Schema abläuft. Aus diesem Grund wurde die Entscheidung für ein *anonymous Publish/Subscribe*-Schema getroffen.

Zustandslosigkeit der Tasks: Damit die Systemfunktion bei einem Neustart von Tasks nicht beeinträchtigt wird, sollten die Tasks keine inneren Zustände besitzen. Das Verhalten darf nur abhängig von den zuletzt empfangenen Eingangsdaten sein. Lediglich der Missionsplaner und der Straßensperrendetektor benötigen innere Zustände, um sich den aktuellen Missionsfortschritt bzw. aktuelle Sperren zu merken. Dies wurde durch eine Sicherung dieser Daten auf die Festplatte und die Wiederherstellung nach einem Programmabsturz gelöst.

Sichtbarkeit der Kommunikation: Nachrichten sollten auch ohne Kontext interpretierbar sein, damit neue gestartete Tasks nach einem Absturz oder zur Diagnose jederzeit losarbeiten können. Dies schließt insbesondere inkrementelle Aktualisierungen von Nachrichten aus. Zudem wurden keine Anfragen nach Zustandsinformationen zwischen den Tasks erlaubt, sondern dies durch regelmäßige Rundsendung dieser Daten ersetzt, auch wenn sie aktuell von keinem Task benötigt werden.

Interprozesskommunikation in TRUCS

Zur Interprozesskommunikation in TRUCS wurde ein eigenes Softwarepaket namens *SimpleComms* entwickelt. Dieses setzt das Konzept des „Anonymous Publish/Subscribe“ oder Ereignis-Busses ein. Den resultierenden Datenfluss zeigt Abb. 2.5. Auf jedem Rechner läuft ein *SimpleComms Server* (SCS), der die Kommunikation zwischen Rechnern über dedizierte TCP/IP Verbindungen abwickelt [140].

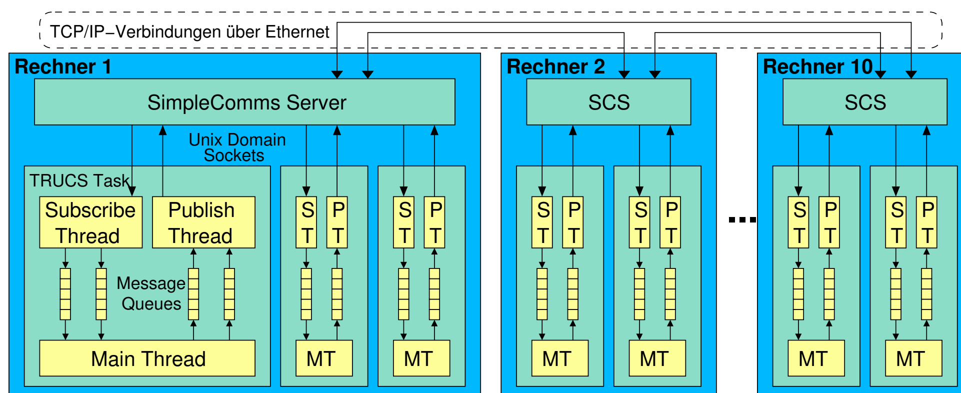


Abbildung 2.5: Datenfluss im *Tartan Racing Urban Challenge System* (erstellt in Anlehnung an [140])

SCS ist verantwortlich dafür, dass eine Nachricht an jeden anderen PC jeweils nur einmal über das Netzwerk zum korrespondierenden SCS geschickt wird. Die Verlustsicherung wird dem TCP-Protokoll überlassen. TRUCS hat dabei ca. 20 MB/s über das Netzwerk verteilt, was zu viel für einen zentralen Knotenpunkt (HUB) gewesen wäre. SCS verteilt Kopien der Nachrichten an alle lokalen Subscriber-Tasks. Für die Übertragung werden die Objekt-Daten in Nachrichtenstrukturen explizit binär kodiert (Marshalling).

Für die lokale Kommunikation von Tasks mit dem SCS werden automatisch mindestens 3 Threads pro Task erzeugt: Ein *Subscribe-Thread* empfängt Nachrichten vom SCS über einen *Unix Domain Socket* und schreibt sie in die entsprechende Nachrichtenwarteschlange für eingehende Nachrichten. Ein *Publish-Thread* reicht die ausgehenden Nachrichten einer zweiten Menge von Warteschlangen wieder über einen *Unix Domain Socket* an den SCS weiter. Die Warteschlangen besitzen eine feste Länge. Neue Nachrichten, die nicht mehr hineinpassen, werden verworfen.

Im *Main-Thread* wird dann nur aus diesen Nachrichtenwarteschlangen gelesen und geschrieben. Zu Testzwecken kann die Nachrichtenschnittstelle durch eine Dateischnittstelle ausgetauscht werden.

Im *Main-Thread* ruft das TRUCS-Framework die Hauptschleife eines Prozesses mit einer definierbaren Rate auf. Der Entwickler überprüft dann jeden Nachrichtenkanal manuell auf neue Nachrichten. Der Vorteil dabei ist, dass diese synchrone Schleife leichter zu programmieren ist als in einem ereignisgetriebenen Modell. Ein bedeutender Nachteil ist, dass dadurch die Latenz des Gesamtsystems in die Höhe getrieben wird, beispielsweise zwischen der Erfassung eines Hindernisses und der Reaktion.

Das Aufzeichnungssystem von TRUCS

TRUCS hat 84 Publish/Subscribe-Kanäle, von denen 67 mit einer Gesamtdatenrate von ca. 1 GB/min (ca. 17 MB/s) aufgezeichnet wurden. Ein Hindernis ist dabei die Schreibda-

tenrate der zwei eingesetzten Festplatten an den zwei Aufzeichnungs-PCs. Es wird jeder einzelne Kanal von einem eigenen *Logging-Task* in eine eigene Datei geschrieben, die als Berkely-DB [156] organisiert ist. Als Schlüssel dient dabei der Zeitpunkt, zu dem die Nachricht aus der Nachrichtenwarteschlange eingehender Nachrichten *entnommen* wurde. Dadurch ergab sich bei der Fehlersuche eine kleine zeitliche Unsicherheit, da dies nicht immer die Zeit war, zu der andere Tasks die Nachricht auch verarbeiteten.

Beim Wiedereinspielen werden aus einer gegebenen Menge von Dateien die Informationen gleichzeitig gelesen und mit einer kontrollierbaren Rate wieder auf den SimpleComms Ereignis-Bus gegeben.

Beim Einspielen von Daten zur anschließenden Filmerstellung wurde eine „Fehlanpassung“ zwischen dem in TRUCS implementierten Kommunikationsstil und den Anforderungen der Filmerstellung diagnostiziert [140]. Dies zeigte sich dadurch, dass die Wiedergabe beim Abspielen manchmal abgehackt war, was durch langsame Festplattenzugriffe verursacht wurde. Es wurden zusätzliche Anforderungen aufgestellt, die Abhilfe schaffen würden:

- Es sollte direkt kontrollierbar sein, welche Zeitsequenz gerade abgespielt wird.
- Es sollte bekannt sein, welche Zeit durch die abgespielten Daten gerade dargestellt wird.
- Das Abspielen von Daten sollte angehalten werden können, wenn Programme mit der Verarbeitung nicht nachkommen.
- Es sollte herauszufinden sein, wann alle Daten eines bestimmten Zeitraumes verfügbar sind. Dies erfordert zudem das Wissen, welche Kanäle gerade abgespielt werden.

Es wird angemerkt, dass Test- und Simulationswerkzeuge von diesen Verbesserungen profitieren könnten, wenn in zukünftigen Arbeiten diese Punkte berücksichtigt werden.

2.2.3 Junior (Stanford Racing)

Abbildung 2.6: Das Fahrzeug „Junior“ des Stanford Racing Teams

Den zweiten Platz im *Urban Challenge* belegte das Stanford Racing Team mit ihrem Fahrzeug „Junior“, einem modifizierten VW Passat. Dieser besitzt einen Lenkmotor mit Riemenantrieb, einen elektronischen Bremsverstärker, E-Gas, Getriebe und eine prototypische Hochstrom-Lichtmaschine sowie zusätzliche individuelle Raddrehzahlsensoren [144].

Seine Umwelt nimmt das Fahrzeug über LIDAR und RADAR-Sensoren wahr. Im Entwicklungsstadium [192] wurde noch von einem omnidirektionalen Kamerasystem mit grafikkartenbeschleunigter Spurmarkierungserkennung berichtet. In [144] werden die Straßenbeschaffenheit und die Markierungen durch zwei seitliche SICK LMS291 und einen nach vorne gerichteten RIEGL LMS-Q120 LIDAR-Sensor wahrgenommen. Hindernisse und andere Fahrzeuge werden über einen Velodyne HDL-64E 360 Grad 64-Strahl LIDAR sowie weitere 2 SICK LDLRS und 2 IBEO ALASCA XR LIDAR-Sensoren detektiert. Bewegte Objekte werden von 5 BOSCH LRR2 RADAR-Sensoren erfasst. Zur Positionsbestimmung dient das inertiale Navigationssystem POS LV 420 von Applanix.

Das Rechnersystem von Junior besteht aus zwei Intel Quad-Core Servern mit dem Betriebssystem Linux, die über Gigabit-Ethernet verbunden sind [144]. Darauf wird die gleiche Softwarearchitektur eingesetzt, wie sie bereits auf dem Vorgängerfahrzeug „Stanley“ eingesetzt wurde, das die DARPA Grand Challenge 2005 gewonnen hat. Auf den Fahrzeugen kommt eine datengetriebene Pipeline mit asynchroner Verarbeitung in unabhängigen Prozessen zur Anwendung. Die Interprozesskommunikation basiert auf dem plattformunabhängigen Softwarepaket zur netzwerkbasierten Nachrichtenverteilung namens „IPC“ [185], das aus der Kommunikationsinfrastruktur der Task Control Architecture stammt [184]. Die Zeit zwischen dem Eintreffen von Sensordaten bis zur Auswirkung auf die Fahrzeugaktorik wird mit ca. 300 ms angegeben.

Ein Schwerpunkt von IPC ist die Kommunikation in verteilten Systemen. So besaß Stanley beispielsweise 6 Pentium-M Rechner, von denen einer die Videoverarbeitung übernahm, einer die Protokollierung, zwei die übrigen Prozesse und zwei unbenutzt waren [205]. Nachrichten können in IPC gerichtet oder anonym über eine zentrale Server-Komponente zur Nachrichtenverteilung und -protokollierung verschickt werden. Dies geschieht entweder über TCP/IP oder lokale Sockets. Zum Empfang von anonymen Nachrichten müssen diese vorher abonniert werden, das Hauptprogramm eines jeden Moduls besteht gewöhnlich aus einer Endlosschleife zum Empfang von Nachrichten und anschließendem Aufruf der registrierten Behandlungsroutine [185].

2.2.4 Odin (Team VictorTango)

Den dritten Platz des Urban Challenge belegte „Odin“, ein modifizierter Ford Escape Hybrid, der in Kooperation der Virgina Tech Universität und TORC Technologies entstand. Das Fahrzeug wurde von TORC mit Drive-by-Wire für Lenkung, Schaltung und Gas sowie einem echtzeitfähigen National Instruments CompactRIO-System zur unterlagerten Regelung ausgerüstet und wird in dieser Konfiguration inzwischen als Produkt verkauft. Zur Energieversorgung der Elektronik war das Hybridsystem ausreichend [10].

Abbildung 2.7: Das Fahrzeug „Odin“ des Teams VictorTango

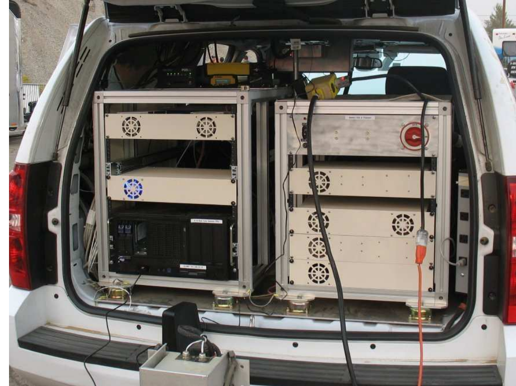
Ein großer Teil der sensoriellen Wahrnehmung wird von einem 260 Grad LIDAR-System, bestehend aus zwei zusammenhängenden 4-Ebenen IBEO Alasca XT Fusion Sensoren abgedeckt. Ein 150 Grad IBEO Alasca XT ist nach hinten gerichtet. Die dazugehörigen Steuergeräte des Herstellers führen die Segmentierung durch und liefern Objektdaten, die dann auf dem PC zusätzlich gefiltert werden. Aufgrund eines Fehlers der Steuergerätesoftware gingen in einem Fall Objekte verloren, sporadisch musste das Steuergerät nach längerem Betrieb neu gestartet werden. Vier SICK LMS-291 Laserscanner waren zusätzlich im Nahbereich für die Detektion von Hindernissen und Bordsteinkanten verantwortlich. Zwei Kameras waren für die Fahrbahnerkennung vorgesehen, wurden jedoch im Rennen nicht verwendet, da die entwickelten Algorithmen noch zu unzuverlässig waren. Zur Positionsbestimmung wurde ein Novatel Propak LB+ System eingesetzt [10].

Das Rechnersystem von Odin besteht aus zwei Servern mit jeweils zwei Quad-Core-Prozessoren. Auf dem einen, der zur Sensorverarbeitung dient, läuft Microsoft Windows XP, da dies das benötigte LabVIEW Vision Entwicklungsmodul erfordert. Auf dem anderen werden die Planungs- und Entscheidungsmodule unter Linux ausgeführt. Zur Prozessisolation wird er in vier virtuelle Maschinen unterteilt. Die Kommunikation zwischen den Systemen wird über ein Gigabit-Ethernet abgewickelt.

Die Kommunikation einzelner Softwaremodule untereinander geschieht nach dem vom amerikanischen Verteidigungsministerium initiierten Standard *JAUS* [99] (*Joint Architecture for Unmanned Systems*), der gerade als Automobilstandard SAE AS-4 übernommen wird. JAUS legt eine Referenzarchitektur fest, die die Systemtopologie aus Komponenten und deren Schnittstellen sowie Nachrichtentypen definiert. JAUS-Nachrichten können über UDP/IP, TCP/IP, serielle oder sonstige Schnittstellen transportiert werden. JAUS legt genaue Bitmuster der Nachrichten einschließlich zu verwendender Nomenklatur und Koordinatensysteme fest, zudem können eigene Nachrichten definiert werden. In Odin werden beispielsweise sensorunabhängige Wahrnehmungsnachrichten verschiedener Sensoren an die Planungskomponente versandt.

2.2.5 Rocky (Team Urbanator)

(a) Frontansicht mit Sensoren



(b) Kofferraum mit Rechnersystem

Abbildung 2.8: Autonomes Fahrzeug „Rocky“ (Team Urbanator, PercepTek Robotics)

Das Fahrzeug „Rocky“ des Team Urbanator von PercepTek Robotics (2008 von Lockheed Martin aufgekauft) kam nicht ins Finale, wird hier aber angeführt, da dort ein Multiprozessor-Mehrkern-Opteronsystem eingesetzt wurde, wie es auch in Kapitel 4.3.1 dieser Arbeit verwendet wird. Rocky ist ein Chevrolet Tahoe, dessen Aktorik mit einem AEVIT X-wire Primary RPV Control System zugekauft wurde. Eine zusätzliche Lichtmaschine mit Gleichrichter und Leerlaufkontrolle, AuraSystems G8500XM, liefert bis zu 8.5kW für die Sensorik und Elektronik [167].

Die Sensorik besteht aus 5 Sony DFW500VL Farbkameras zur Straßenerkennung, die über Firewire mit dem Rechner verbunden sind. 5 LIDAR-Sensoren dienen der Hindernis- und Fahrzeugdetektion sowie Bordsteinerkennung und werden von 5 RADAR-Sensoren u. a. beim Abbiegen und Abstandshalten ergänzt. Beide Systeme werden über CAN- bzw. Seriell-zu-Ethernet-Konverter angebunden. Zur Positionsbestimmung dient ein Applanix POS LV 220. Die Anbindung der Aktorik geschieht über einen eingebetteten Rechner mit einem 1.6GHz Pentium-M Prozessor, auf dem die unterlagerten Regelkreise ausgeführt werden und der ebenfalls über Ethernet angebunden ist [167].

Alle Verhaltens- und Wahrnehmungsmodule laufen auf einem einzigen Rechner, der 8 DualCore Opteron Prozessoren mit je 2.2 GHz auf einem Tyan Thunder Mainboard mit Tochterplatine enthält. Zur Interprozesskommunikation wird die *Neutral Message Language* (NML) aus der Real-Time Control Systems Library (RCS) des NIST eingesetzt. Dabei werden Daten zwischen den Modulen über von NML verwaltete gemeinsame Speicherbereiche ausgetauscht.

NML bietet dazu verschiedene Methoden zum Zugriffsschutz seiner statischen Puffer an: Die Standardeinstellung ist, dass bei jedem Lese- oder Schreibzugriff eine Betriebssystemfunktion zum Sperren einer Semaphore aufgerufen wird. Auf Einprozessormaschinen können alternativ alle Interrupts abgeschaltet oder die Präemption unterbrochen werden. Für Mehrprozessorsysteme steht eine experimentelle Variante mit doppelter Puf-

ferung bereit, die in den meisten Fällen sehr schnell ist, da sie ohne Betriebssystemaufrufe auskommt. Gelegentlich führt sie zu signifikant längeren Ausführungszeiten oder Zeitüberschreitungen, da noch keine Prioritätsvererbung vorgesehen ist [151].

2.3 Robotikarchitekturen

Ein weiteres großes Feld von Architekturen für autonome Systeme findet man in der Robotik. Je nach Einsatzgebiet finden sich jedoch vom Fahrzeug abweichende Rahmenbedingungen, die nur in der Robotik akzeptiert werden. So setzen viele bestimmte Sensoren oder Aktoren voraus und legen den Datenfluss fest. Damit ist z. B. die Integration einer aktiven Kameraplattform nicht ohne Weiteres möglich. Dafür bieten sie meist Funktionalitäten, die weit über die Interprozesskommunikation hinausgehen.

Die Anforderungen an harter Echtzeit sind hingegen oft geringer. So hat ein Fahrzeug aufgrund seiner Masse eine größere Trägheit als eine kleine Roboterplattform. Kann diese bei längeren Berechnungen kurzfristig angehalten werden, ist das bei der schnellen Fahrt eines Fahrzeugs auf öffentlichen Straßen nicht möglich. Die Echtzeitanforderungen bei der Reaktion auf Umwelteinflüsse sind in diesem Fall höher.

CORBA [155] (*Common Object Request Broker Architecture*) ist programmiersprachenunabhängig und plattformübergreifend, und damit gut für heterogene Umgebungen geeignet. Um die umfangreiche CORBA-Spezifikation mit ihren Service-Definitionen zu erfüllen, haben dessen Implementierungen durch die geforderte Komplexität erhöhte Speicheranforderungen und ihre Plattformtransparenz erzeugt einen gewissen Laufzeit-Overhead. Bei der Verwendung von CORBA geschieht die Interprozesskommunikation für den Benutzer sehr komfortabel, indem auf verteilte Objekte zugegriffen wird, die zu Beginn der Entwicklung mit Hilfe einer *Interface Definition Language* (IDL) deklariert wurden. CORBA wird gerne in größeren verteilten Robotikanwendungen wie in [110, 17] als Zwischenschicht (*Middleware*) eingesetzt.

In *RT-CORBA* [154, 172] lassen sich beliebige Quality-of-Service Vorgaben spezifizieren. Ob diese eingehalten werden können, ist u. a. vom darunterliegenden Betriebssystem abhängig, was nicht immer gegeben ist [123]. In [171] zeigen Messungen, wie konkurrierende Zugriffe niedrigpriorer Clients die Zugriffe eines hochprioren auf den Server bremsen. Dort wird der Speicherbedarf der verwendeten freien CORBA-Bibliothek *The ACE ORB* (TAO) mit > 1.5 MB im übersetzten Zustand angegeben. In einen Echtzeitnachweis durch statische Analyse müsste der aktive zugrundeliegende Quellcode vollständig miteinbezogen werden. TAO setzt auf dem *Adaptive Communication Environment* (ACE) auf. In [208] wurden Latenzzeiten des Ereignisdienstes von TAO mit ca. 0.5-0.7 ms gemessen.

MIRO (*Middleware for Mobile Robots*) [213] basiert auf CORBA und bietet Geräte- und Dienstabstraktionen für Sensoren und Aktoren von mobilen Roboterplattformen. Das Klassenframework enthält u. a. eine Verhaltenssteuerung und einen Pose-Schätzer. Miro setzt auf der CORBA-Implementierung TAO auf und nutzt besonders dessen Benachrichtigungsdienst. Dadurch wird eine plattformübergreifende Interprozesskommunikation

erreicht. Die Benachrichtigungen können aufgezeichnet und zur Simulation wieder eingespielt werden. Harte Echtzeitfähigkeit steht bei Miro nicht im Fokus, insbesondere da einige der verwendeten Roboterplattformen ihren Status in 100 ms Intervallen liefern. In Messungen, die die Aktorik und Sensorik einer Roboterplattform mit 10 ms Statusintervallen einbeziehen, wurden Latenzzeiten von ca. 5-27 ms ermittelt und es konnte kein signifikanter CORBA-Overhead festgestellt werden.

OROCOS (*Open Robot Control Software*) [21] hat seinen Schwerpunkt auf der modularen Steuerung von Robotern und Industriemaschinen. Es wurde zudem im Fahrzeug „Spirit of Berlin“ der FU Berlin eingesetzt, das bis ins Halbfinale des Urban Challenge kam. OROCOS bietet umfangreiche Softwarebibliotheken für Kinematik, Dynamik, Bayes-Filter, Steuerungskomponenten und ein Echtzeitframework. OROCOS ist komponentenbasiert und bietet Designmuster für Sensoren, Aktoren, Trajektoriengeneratoren, Beobachter und Regler sowie Infrastrukturkomponenten [22]. Eigene Komponenten werden implementiert indem der Anwendungscode in die vorgegebene C++-Klassenhierarchie eingebunden wird und dann von der OROCOS Ausführungseinheit aufgerufen wird. Komponenten können online über den „TaskBrowser“ kontrolliert und mit XML-Dateien konfiguriert werden. In OROCOS wird eine blockierungsfreie Methode zum lokalen Datenaustausch [189] eingesetzt, die a priori die Maximalanzahl der zugreifenden Tasks benötigt um dann entsprechend $2 \cdot n_{Tasks} \cdot size_{Buffer}$ ihren Speicher zu dimensionieren. Zur Netzwerkkommunikation besitzt OROCOS Schnittstellen zu CORBA.

MCA2 (*Modular Controller Architecture*) [175] ist ein modulares und netzwerktransparentes C++-Framework zur Implementierung von komplexen echtzeitfähigen Regelungen. Einzelne Softwaremodule werden an ihrem Sensor- und Steuerein- und Ausgängen verbunden und zu Gruppen hierarchisch zusammengefasst, die wiederum geschachtelt werden können. Ein MCA-Part ist die Ausführungsumgebung einer Modulgruppe. Es sorgt dafür, dass mit einer wählbaren Periode zuerst die Sensorfunktionen, dann die Steuerfunktionen der enthaltenen Module nacheinander aufgerufen werden und zwischendurch die Daten weitergereicht werden. Ein- und Ausgangsdaten werden einheitlich als Vektoren von Doublewerten modelliert und können jederzeit von außen über das graphische MCAadmin-Tool eingesehen und geändert werden. Zum Austausch großer Datenmengen existiert ein Blackboard, das jedes Modul für die Dauer seines Zugriffs sperren sollte, um Inkonsistenzen zu vermeiden [174].

Eine Stärke von MCA2 ist die Verschiebbarkeit der Regelalgorithmen auf Microcontroller des Typs C167 [176]. Diese besitzen digitale und analoge Sensoreingänge und Aktorausgänge einschließlich Motorleistungsstufen. Im Betrieb kommunizieren sie über CAN für den Benutzer transparent mit einem RT-Linux-System, auf dem überlagerte MCA-Parts laufen. In dieser Konfiguration lassen sich nicht nur Software- sondern auch Hardwaremodule wiederverwenden. MCA2 findet speziell in Robotern mit vielen Freiheitsgraden Anwendung, wie in der Regelung einer sechsbeinigen Laufmaschine [105]. In der Architektur eines mobilen humanoiden Serviceroboters [173] wird MCA2 für die unterlagerten Regelkreise der holonomen Plattform [181], des Schwenk-Neigekopfes mit Kameras und des Arms mit 7 Freiheitsgraden eingesetzt. MCA2 eignet sich auch zur Umsetzung der Regelung eines autonomen Smart Roadster [180].

Das *Player/Stage*-Projekt [64] hat seinen Schwerpunkt auf Systemen mit vielen Robotern und verteilter Steuerung sowie Sensornetzwerken. Player ist ein Serversystem mit Gerätetreibern für verbreitete Roboterplattformen und Sensoren. Es besitzt eine TCP/IP-Schnittstelle, die von verschiedensten Betriebssystemen und Programmiersprachen aus genutzt werden kann, um den Roboter zu steuern. Stage ist ein Multiroboter-fähiger Simulator für eine zweidimensionale Welt und besitzt die gleiche Schnittstelle wie Player. Entwickelte Programme zur Verhaltenssteuerung können so mit Stage getestet werden, bevor sie mit Player auf der richtigen Roboterhardware eingesetzt werden. Die Weiterentwicklung Gazebo dehnt die Simulation auf 3D aus.

Carmen (*Carnegie Mellon Navigation Toolkit*) [145] ist ein Framework zur Steuerung mobiler Roboterplattformen, die sich auf der Stelle drehen können und Sensoren zur Positions- und Entfernungsmessung besitzen. Es bietet Funktionen zur Navigation, Lokalisation, Pfadplanung, Hindernisvermeidung und Bewegungssteuerung. Das Designziel waren einfache Benutzbarkeit, Erweiterbarkeit und Robustheit. Zur Förderung der Transparenz werden Steuer- und Visualisierungsmodule separiert. CARMEN nutzt zur Interprozesskommunikation IPC [185], das bereits in Kapitel 2.2.3 vorgestellt wurde.

CLARAty (*Coupled Layer Architecture for Robotic Autonomy*) [148] wird von der NASA für Expeditionsroboter wie den Mars Rover eingesetzt. Diese unterscheiden sich jedoch in ihren physikalischen Fähigkeiten in Abhängigkeit von der Rad- oder Beinausstattung, der Hardwareansteuerung sowie der Sensorkonfiguration. Darauf sollen einheitlich neue Fähigkeiten integrierbar sein und zusammenarbeiten. Die Architektur verfolgt einen zweistufigen Ansatz, um den verwendeten Programmierparadigmen Rechnung zu tragen: Auf der unteren funktionalen Ebene dominiert die objektorientierte prozedurale Programmierweise mit einer genau vorgegebenen Ausführungsreihenfolge. Der deklarative modellbasierte Ansatz auf der oberen Entscheidungsebene beschreibt mögliche Handlungen, die entsprechend des Missionsziels von einem Planer ausgewählt und ausgeführt werden.

2.4 Nicht-Blockierende Datenaustauschprotokolle

Auf modernen Rechnersystemen werden meist mehrere Programmpfade gleichzeitig verfolgt. Dies geschieht zum einen durch den Einsatz von *Multitasking*-Betriebssystemen, die gewöhnlich präemptiv arbeiten und alternierend mehrere Prozesse oder Threads in kurzer zeitlicher Abfolge ausführen und wieder unterbrechen. Sind zum anderen mehrere Prozessorkerne (*multicore*) vorhanden, werden die lauffähigen Prozesse meist dynamisch darauf verteilt, und es laufen stets mehrere gleichzeitig ab.

Möchten verschiedene Prozesse auf gemeinsame Daten im Speicher zugreifen, besteht die Gefahr von Inkonsistenzen, da sich die Reihenfolge der Ausführung in präemptiven Multitaskingbetriebssystemen dynamisch zur Laufzeit ergibt und nicht statisch festgelegt ist. Daher werden Techniken zur Synchronisation mehrerer Prozesse beim gleichzeitigen Zugriff verwendet.

Die bekanntesten sind gegenseitige Ausschlusstechniken (*mutual exclusion*) wie Semaphoren [45]. Dabei werden Prozesse durch Sperren (*locks*) blockiert, wenn gerade ein anderer Prozess auf einen geschützten Speicherbereich zugreift. Es besteht jedoch die Gefahr der Prioritätsinversion, die durch Unterstützung des Schedulers und geeignete Prioritätsprotokolle, wie z. B. das *Priority Inheritance Protocol* (PIP) [182], abgewendet werden muss. Bei gleichzeitigem Einsatz mehrerer Sperren besteht zusätzlich das Problem von gegenseitigen Verklemmungen (*deadlocks*).

Eine Alternative dazu sind Datenaustauschprotokolle, die ganz ohne blockierende Synchronisierungsmechanismen des Betriebssystems auskommen. Der Zugriff auf gemeinsame Datenbereiche darf dabei nur unter der Kontrolle des jeweiligen Algorithmus geschehen, der sicherstellt, dass bei gleichzeitiger Ausführung selbst im ungünstigsten Fall keine Inkonsistenzen entstehen können. Dies wird zum einen durch die festgelegte Anweisungsfolge erreicht. Zum anderen werden atomare Operationen des Prozessors eingesetzt, die ein Speicherelement nicht unterbrechbar und ohne sichtbarem Zwischenschritt modifizieren.

2.4.1 Sperrenfreie Protokolle

Die Sperrenfreiheit (*lock-free*) bedeutet lediglich, dass keine blockierenden Ausschlusstechniken eingesetzt werden. Ein frühes Beispiel ist der Algorithmus für *einen* schreibenden Prozess und *mehrere* gleichzeitig lesende Prozesse [116]: Der gemeinsame Speicherbereich wird um zwei Versionszähler v_1 und v_2 ergänzt. Der Schreiber erhöht v_1 vor, und v_2 nach dem Überschreiben des Datenbereichs. Der Leser liest v_2 vor und v_1 nach dem Lesen. Die Lese- und Schreiboperationen eines mehrelementigen Zählers werden zudem in entgegengerichteter Reihenfolge ausgeführt. Stellt der Leser mit $v_1 \neq v_2$ die Inkonsistenz fest, muss er seinen Leseversuch wiederholen. Dabei besteht theoretisch die Möglichkeit immerwährender Wiederholungen, der Leser „verhungert“ (*starvation*). Der Wertebereich der Zählervariablen muss zudem die maximal mögliche Anzahl Schreiboperationen während einer Leseoperation abdecken können.

In [113] wird das Problem mit *einem* Schreiber und *mehreren* Lesern für den Spezialfall gelöst, dass der Schreiber auf einer unabhängigen Verarbeitungseinheit (*communication controller*) läuft und die minimale Zeit *mint* zwischen zwei Schreiboperationen a priori bekannt ist. Dadurch lässt sich die Analyse des Schedulings vereinfachen und die Anzahl der Leseversuche nach oben beschränken. Zur Konsistenzkontrolle wird ein Zähler, genannt *concurrency control field*, eingesetzt, der jeweils vor und nach einem Schreibvorgang inkrementiert wird. In einer Erweiterung des Algorithmus werden mehrere Puffer eingesetzt, um die Zeit zwischen zwei Änderungen eines Puffers zu verlängern.

Ein weiteres Verfahren, *Read-Copy-Update* [139], schreibt in eine Kopie und wartet danach in einer Ruhephase (*quiescent period*), bis alle vorhandenen Prozessoren einen Ruhezustand (*quiescent state*) durchlaufen haben. Erst dann wird die alte Version gelöscht. Ein wichtiges Anwendungsgebiet sind vor allem leseintensive Datenstrukturen in Betriebssystemen, die vergleichsweise selten geändert werden. Sind gleichzeitige Schreibzugriffe nötig, werden gewöhnliche Sperren eingesetzt.

2.4.2 Wartefreie Protokolle

Wartefreie Protokolle (*wait-free*) beschränken zusätzlich die maximal notwendige Anzahl von Wiederholungsversuchen. Dabei helfen sich die gleichzeitig aktiven Instanzen eines Algorithmus gegenseitig, ihre Operation abzuschließen, die sie in einem ersten Schritt zuvor ankündigen.

In [90] wird gezeigt, dass für jede exklusive Datenstruktur eine wartefreie Implementierung unter Verwendung verlinkter Listen konstruiert werden kann. Die Konstruktion ist abhängig von der Anzahl der zugreifenden Prozesse. Die universelle Implementierung hat einen Speicherbedarf von $O(n^3)$ Kopien des Ursprungsobjekts und beschränkt die Rechenzeit in der Größenordnung von $O(n^3)$. Die Hauptschleife weist dabei maximal $n + 1$ Iterationen auf.

Außerdem wird in [90] der Begriff einer Konsensnummer (*consensus number*) eingeführt, der angibt, wie viele Prozesse gleichzeitig eine Operation durchführen können, und sich danach stets einig über das Ergebnis sind. Wenn beispielsweise mehr als ein Prozess eine Speicherstelle gleichzeitig beschreiben will, „gewinnt“ der letzte Schreiber in dieser Wettlaufsituation (*race condition*). Ein einfacher Schreibzugriff hat daher die Konsensnummer 1. Eine nicht-triviale atomare *Read-Modify-Write* Operation wie *Swap* ist nur bei maximal 2 Prozessen konsensfähig. Mit beispielsweise *Compare-and-Swap* (Vergleich mit einem Wert und bedingter Austausch, kurz CAS) ist ein Konsens unter beliebig vielen Prozessen herstellbar und so eine entsprechende wartefreie Implementierung konstruierbar.

Für prioritätsbasiertes Scheduling sind nach [162] auf Ein- und Mehrprozessorsystemen schwächere Konsensoperationen ausreichend. Weitere wartefreie Protokolle mit gegenseitiger Hilfe präsentiert [3]. Dabei wird jeweils das prioritätsbasierte Scheduling vorausgesetzt. Da dort immer der Task ungestört weiterarbeiten kann, der gerade die höchste Priorität besitzt, wird er verwendet, um den anderen Tasks zu helfen. Voraussetzung ist, dass jeder Task seine augenblickliche Priorität kennt. Bei mehreren Prozessoren helfen sich die Prozesse reihum. Die maximale Schleifenanzahl lässt sich unter diesen Voraussetzungen signifikant verkleinern. Alle Zugriffe werden dabei sequentialisiert, sodass sich auch Lesezugriffe einreihen müssen. Dafür müssen keine Kopien angelegt werden.

Wartefreiheit in Verbindung mit dynamischen Schedulingprotokollen wird für Einprozessormaschinen in [7] untersucht. Dabei wird eine CAS-Instruktion für zwei Worte (CAS2) benötigt. Zur Implementierung auf Einprozessormaschinen wird ein nicht unterbrechbarer CAS2-Systemaufruf vorgeschlagen oder eine Softwareemulation. Die Softwareemulation für Einprozessormaschinen mit einem prioritätsbasierten Scheduler wird in [5] beschrieben. In [2] wird für Mehrprozessormaschinen ein Spinlock vorgeschlagen. In [6] wird für Multiprozessorsysteme der allgemeine Multi-Wort-CAS (MWCAS) durch einen einfacheren CCAS-Befehl (*Conditional-Compare-and-Swap*) emuliert. Dieser muss wiederum nachgebildet werden und benötigt dafür die kurzzeitige Deaktivierung aller Interrupts oder einen Betriebssystemaufruf, um vorübergehend die Präemption abzuschalten. In [8]

wird die Verwendung der Methoden für Transaktionen in speicherbasierten Datenbanken vorgeschlagen.

In [2] werden für zeitscheibenbasierte Systeme (*quantum scheduling*) Methoden zur Wartefreiheit vorgestellt. Dabei wird vorausgesetzt, dass eine Zeitscheibe länger als die Ausführungszeit des Algorithmus dauert und bei dieser optimistischen Annahme maximal zwei wiederholte Versuche notwendig sind. Auf Multiprozessorsystemen wird dieser Wiederholungsmechanismus zusammen mit einer präemptiblen Sperre mit angeschlossener Warteschlange kombiniert, um die Wartezeit zu begrenzen. Dazu ist jedoch die Unterstützung des Betriebssystems nötig: Es muss eine globale Variable setzen, die anzeigt, dass ein Task unterbrochen wurde und nun nicht mehr läuft.

Schließlich wird in [4] die Untersuchung auf hybride Systeme aus prioritäts- und zeitscheibenbasiertem Scheduling ausgedehnt. Dazu werden kaskadierte Konsenslevel eingesetzt, die u. a. abhängig von der Anzahl der Prozesse, der Prozessoren und der Zeitscheibenlänge sind.

Einen pragmatischen Ansatz verfolgt [92]: Da die verbreitete IA-32 Architektur keine CAS2-Instruktion besitzt und die Emulation aufwendig und von Bedingungen abhängig ist, wird auf den Einsatz von CAS2 verzichtet. Globale Daten im Betriebssystem werden bei der Modifikation durch Interruptabschaltung bzw. auf Mehrprozessorsystemen durch Spinlocks geschützt. Für nicht-echtzeitkritische Daten wird ein wartefreier Algorithmus implementiert, der seine gerade helfenden Threads als Stapel organisiert und den Prozessor aktiv an andere Threads im Stapel abgibt. Für die Benutzerebene wird ein betriebssystemunterstützter MWCAS-Systemaufruf oder eine präemptionsfreie Sperre vorgeschlagen.

2.4.3 Schleifenfreie Protokolle

Ein schleifenfreies Protokoll ist ebenfalls blockierungs- und wartefrei. Zusätzlich darf es bei einem Zugriff zu keinen durch andere Prozesse verursachten Mehrfachdurchläufen von Schleifen kommen, z. B. zu Lesewiederholungen bei Inkonsistenzen wie in [116].

Für nur jeweils *einen* Leser und Schreiber wird in [186] ein Protokoll vorgestellt, in dem keine immerwährenden Wiederholungen auftreten können. Es stellt keine Bedingungen an die relativen Ausführungsraten und Dauern der Operationen und benötigt keine Spezialoperationen, lediglich Schreib- und Lesezugriffe: Es werden vier Datenpuffer eingesetzt, die in einem 2×2 Feld angeordnet werden. Die eine Achse kontrolliert der Schreiber, die andere der Leser. So kann durch eine orthogonale Vermeidungsstrategie sichergestellt werden, dass beide immer ein freies Felderpaar zum konfliktfreien Schreiben und Lesen zur Verfügung haben.

Einen Algorithmus für einen Schreiber und mehrere Leser auf einem Multiprozessorsystem stellt [26] vor. Dieser ist scheduler- und betriebssystemunabhängig. Er benötigt bei n Lesern $n + 2$ gleichartige Puffer. Unter Verwendung gemeinsamer Variablen, die mit atomaren *Test-and-Set*-Operationen bearbeitet werden, wird zwischen dem Schreiber und

jedem Leser ein Konsens gefunden, welcher Puffer überschrieben werden kann, ohne eine der laufenden Leseoperationen zu stören.

Ohne Algorithmus zur Entscheidungsfindung kommt der Ansatz in [27] für einen Schreiber und mehrere Leser aus: Bei jedem neuen Schreibzugriff benutzt der Schreiber den nächsten Platz eines Ringpuffers und vermerkt dessen Position nach Abschluss der Operation in einer globalen Variablen. Diese benutzen die Leser als Index für jeden Lesezugriff. Ein Vorteil ist der sehr geringe Zeitaufwand dieser Methoden, ein Nachteil die Möglichkeit einer Inkonsistenz bei einem zu langsamen Lesezugriff. Daher wird zusätzlich eine Konfigurationsregel für die Anzahl der zu implementierenden Puffer in Abhängigkeit von der Zykluszeit des Schreibprozesses und den a priori bekannten Rechenzeiten aller beteiligten Tasks aufgestellt.

Eine Erweiterung um mehrere Schreibprozesse wird in [25] beschrieben. Dort wird für jeden Schreibprozess eine eigene Menge von Puffern angelegt. Nach Abschluss einer Schreiboperation wird eine globale Variable umgesetzt, über die die Leser einen Zeiger auf die eben geschriebenen Daten bekommen. Damit diese stets auf die Daten des zuletzt gestarteten Schreibprozesses zeigt, wird ein Kohärenzprotokoll zum Austausch der Variable eingesetzt: Die Variable besteht aus der Nummer des Schreibprozesses und einem Zeitstempel. Da sie durch eine *Compare-and-Swap*-Operation atomar ausgetauscht wird, müssen sich beide Teile ein Speicherwort teilen, was die Aufteilung der verfügbaren Bits erfordert. Der Zeitstempel ist ein globaler Zähler, der bei jeder Schreiboperation erhöht wird und ausreichend Werte besitzen muss, damit er nicht während einer Operation einmal umläuft.

Die in [201] zusammengefassten Arbeiten befassen sich hauptsächlich mit der Erstellung von *Snapshots* gemeinsamer Daten unter Einsatz eines Scanner-Tasks. Diese finden Anwendung in eingebetteten Systemen u. a. im Automobilumfeld. In einer der Arbeiten [202] wird ein registerbasierter Algorithmus zum Austausch einzelner Datenwerte zwischen n Schreibern und n Lesern entworfen: Er arbeitet auf einer Matrix von $n \times n$ Registern, bei der in Zeilen geschrieben und aus Spalten gelesen wird. Die Breite eines Registers ist durch die möglichen atomaren Schreib-/Lesezugriffe festgelegt. Diese verfügbaren Bits werden aufgeteilt in einen Nutzdatenteil und einen Zeitstempel. Durch den Zeitstempel wird sichergestellt, dass immer die neuesten Daten verwendet werden. Mit Kenntnis des Zeitverhaltens aller Tasks kann der Wertebereich des Zeitstempels eingeschränkt und alte Werte nach einem Überlauf wiederverwendet werden.

3 Grundlagen

Dieses Kapitel liefert die Grundlagen für das Verständnis der Architekturentscheidungen späterer Ausführungen. Es definiert den Begriff harter Realzeitsysteme und gibt einen Überblick über die Echtzeiteigenschaften moderner Computersysteme. Dabei werden die Schwerpunkte Hardwareschnittstellen, Betriebssystem und Speicherverwaltung beleuchtet. Anschließend wird der relevante Hintergrund zum betrachteten Anwendungsschwerpunkt „Kognitive Automobile“ erläutert.

Eine solche Darstellung kann nie vollständig sein, ohne den Umfang dieser Arbeit zu sprengen. So wird an dieser Stelle für die Definition des Architekturbegriffs einschließlich eines umfassenden Kriterienkatalogs auf die gute Darstellung in [84] verwiesen, an der sich einige Abschnitte in Kapitel 7.1 messen.

3.1 Automatisierung von Prozessen

Durch den technischen Fortschritt lassen sich der Komfort und die Lebensqualität für viele Menschen steigern. So werden schwere Arbeiten durch Maschinen erleichtert. Viele Tätigkeiten sind jedoch auf Dauer lästige Routine, sodass sich der Mensch stattdessen auf wichtigere kreative Tätigkeiten konzentrieren möchte. Daraus entsteht das Bestreben, solche Aufgaben zu automatisieren. *Automatisierung* ist nach [47] „das Ausrüsten einer Einrichtung, sodass sie ganz oder teilweise ohne Mitwirkung des Menschen bestimmungsgemäß arbeitet“. Im Vorgriff auf spätere Kapitel soll hier als Beispiel die (Teil-) Automatisierung der Fahraufgabe eines Kraftfahrzeugs angeführt werden. Bestimmungsgemäß meint in diesem Zusammenhang unter anderem unfallfrei, gesetzeskonform und ohne Behinderung anderer Verkehrsteilnehmer.

Für einfache Automatisierungsaufgaben genügen mechanische oder elektronische Vorrichtungen, wie z.B. ein Fliehkraftregler oder ein Spannungsregler. Für komplexere Aufgaben, insbesondere dann, wenn Entscheidungen getroffen werden müssen, sind programmgesteuerte Mikrocomputer verbreitet, die in diesem Zusammenhang *Prozessrechensysteme* [58] genannt werden. Die einem solchen System „aufeinander einwirkenden Vorgänge [...], durch die Materie, Energie oder auch Information umgeformt, transportiert oder auch gespeichert wird“ [47] sind ein *Prozess*. Man unterscheidet einen *Rechenprozess*, der auf dem Prozessrechensystem ausgeführt wird, und den *technischen Prozess*, „dessen Zustandsgrößen mit technischen Mitteln gemessen, gesteuert und/oder geregelt werden können“ [46].

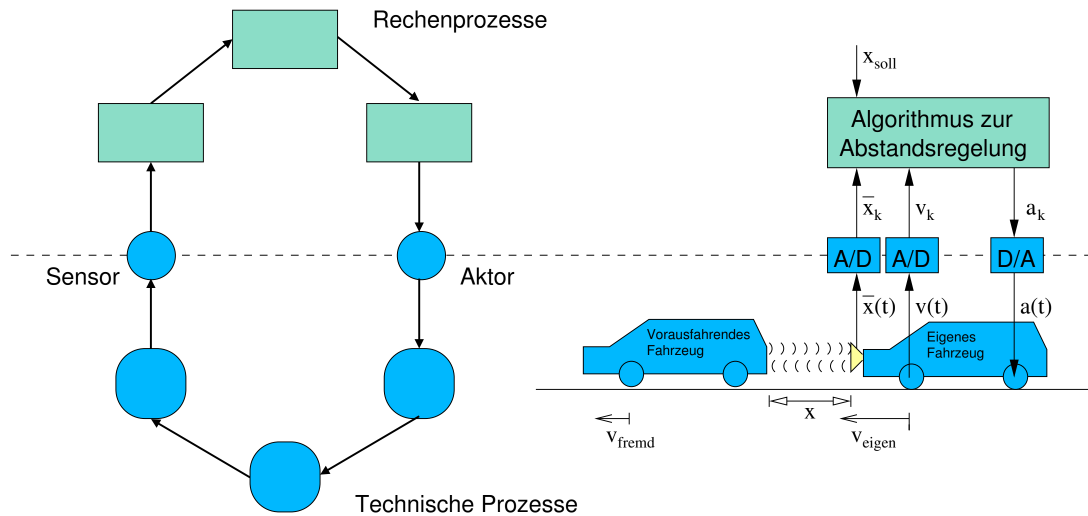


Abbildung 3.1: Zusammenwirken von Prozessen mit Beispiel aus dem Automobil

Abb. 3.1 zeigt links die Struktur einer solchen Prozesskette. An der Schnittstelle zu den Rechenprozessen sorgt ein *Sensor* für die Datenerfassung (ggf. angekoppelt über Bussysteme), ein *Aktor* gibt die berechneten Stellkommandos an den technischen Prozess zurück. Auf der rechten Seite von Abb. 3.1 zeigt ein praktisches Beispiel diese Einteilung: Um den Abstand des eigenen Automobils zum vorausfahrenden Fahrzeug konstant zu halten, werden Abstandsinformationen $\bar{x}(t)$ und die eigene Geschwindigkeit $v(t)$ mittels Sensoren gemessen und als $\bar{x}_k$,v_k zeitlich und räumlich diskretisiert durch Analog/Digital(A/D)-Wandler in ein Prozessrechensystem eingespeist. Ein digitaler Regelalgorithmus berechnet eine Sollbeschleunigung a_k, die durch einen Digital/Analog(D/A)-Wandler als $a(t)$ den technischen Prozess erreicht und ihn durch Aktoren an Gaspedal und Bremse beeinflusst. Wäre $\bar{x}_k$ kein Vektor von Entfernungsmessungen, die zuvor zusammengefasst und klassifiziert werden müssten, würde in diesem einfachen Beispiel auch eine analoge Regelstrecke ausreichen.

3.2 Realzeitsysteme

Als *Echtzeit* oder *Realzeit* bezeichnet man die Zeit, wie sie in der realen Welt kontinuierlich abläuft. Im Gegensatz dazu kann die *Modellzeit* in einer Simulation beliebig gesteuert und bei Bedarf angehalten werden. Werden die Zeitbedingungen von einem technischen Prozess vorgegeben, müssen die Rechenprozesse mit diesem Schritt halten, das Prozessrechensystem muss *echtzeitfähig* sein. So definiert [47] die *Echtzeitfähigkeit* als „Eigenschaft eines Rechensystems, [das] die Rechenprozesse ständig ablaufbereit hält, derart, dass innerhalb einer vorgegebenen Zeitspanne auf Ereignisse im Ablauf eines technischen Prozesses reagiert werden kann.“

In der Literatur werden Echtzeitsysteme weiter klassifiziert. Zur Einstufung dient unter anderem die Brauchbarkeit verspäteter Ergebnisse, der durch Verspätung verursachte

Schaden und ob zugesicherte Zeitschranken deterministischer oder probabilistischer Natur sind [128]:

Harte Echtzeit (hard real-time): In einem harten Echtzeitsystem sind verspätete Ergebnisse nutzlos und können katastrophale Auswirkungen haben. Daher muss eine vorgegebene Zeitschranke (*Deadline*) in jedem Fall eingehalten werden, das Zeitverhalten muss deterministisch sein. Die gegebenen Garantien müssen validierbar sein, entweder durch einen mathematischen Beweis oder eingehende Simulationen und Tests.

Weiche Echtzeit (soft real-time): In einem weichen Echtzeitsystem sind verspätete Ergebnisse unerwünscht, sie haben zumindest noch einen geringen Nutzen. Das gelegentliche Verletzen einer Zeitschranke ist hinnehmbar, lediglich die Qualität sinkt. Andere Parameter wie der gesamte Durchsatz haben oftmals einen ähnlichen Stellenwert. Ein Beispiel ist z.B. die Videodekodierung, wo ein einzelnes verlorenes Videobild (*Frame Loss*) wenig auffällt und vom Zuschauer schnell wieder vergessen ist. Häufungen werden jedoch als schlechte Qualität wahrgenommen.

Daneben findet man in Literatur weitere Bezeichnungen wie *firm real-time*, die Systeme charakterisieren, die zwar nicht 100% hart echtzeitfähig sind, aber mehr als weiche Echtzeit bieten. Streng genommen finden sich auch bei harten Echtzeitsystemen je nach Definition unwahrscheinliche externe Ursachen für eine Verletzung der Zeitbedingungen (z.B. Systemausfall durch Fremdeinwirkung wie mutwillige Zerstörung).

In dieser Arbeit wird unter Echtzeit, wenn nicht explizit angegeben, stets harte Echtzeit verstanden. Für das Beispiel aus Abb. 3.1 bedeutet das, dass wenn das vorausfahrende Fahrzeug bremst, das eigene ebenfalls innerhalb einer festen, durch die Verzögerungsmöglichkeiten der Bremsanlage vorgegebenen Zeitschranke, bremsen muss. Bremst es zu spät, passiert ein folgenschwerer Auffahrunfall und die Information, dass man hätte bremsen sollen, ist wertlos.

3.3 Realzeiteigenschaften von PC-Hardware und Peripherie

Um harte Zeitbedingungen einzuhalten, dürfen nicht nur einige als besonders wichtig erachtete Komponenten eines Systems echtzeitfähig sein, sondern jede an der Informationsverarbeitung und -weitergabe beteiligte Komponente muss berücksichtigt werden. In die Berechnung maximaler Zeitdauern müssen auch alle möglichen Störungen einfließen. Bei kleinen Systemen aus einfachen Mikroprozessoren mit überschaubarer Hard- und Software ist dies mit vertretbarem Aufwand machbar. Daher werden solche Systeme nach wie vor in sicherheitskritischen Anwendungen mit kurzen Deadlines bevorzugt.

Soll ein größeres Datenaufkommen in Echtzeit verarbeitet werden, sind dafür höhere Rechenkapazitäten notwendig. Die Daten werden beispielsweise von leistungsfähigen Sensoren geliefert, die eine Vielzahl von Werten gleichzeitig und mit hohen Auflösungen er-

zeugen. Sie erlauben eine immer genauere Überwachung und Kontrolle des technischen Prozesses durch höhere Ableitungen und exaktere Prädiktionen. Der Einsatz von Kameras mit der dafür notwendigen Bildverarbeitung ist nur ein weiteres Beispiel dafür. Ein Realzeitsystem, das zur Steuerung eingesetzt werden soll, muss diese Menge an Daten schritthaltend verarbeiten können. Die dafür nötigen Systeme werden immer umfangreicher und komplexer, sodass die Entwicklung passender Speziallösungen sehr teuer wird.

Eine attraktive Alternative bietet der Einsatz von PC–Standardkomponenten als *Prozessrechensysteme*. Die Komponenten werden in großen Stückzahlen gefertigt, der sich ergebende Massenmarkt macht sie deutlich billiger als eigens angefertigte Speziallösungen. Die Kompatibilität der PC–Komponenten untereinander gewährleistet die Unabhängigkeit von einem bestimmten Hersteller und gibt die Sicherheit, dass ein bestehendes Design mit geringem Aufwand auch unter Verwendung zukünftiger Hardware realisiert werden kann. Voraussetzung ist, dass eine Lösung gefunden wird, die ohne Hardwareänderungen auskommt.

PC–Standardkomponenten wurden allerdings nicht für den Einsatz in Realzeitsystemen entworfen. Als Problem erweist sich die Tatsache, dass bei der Entwicklung der Hardware eine möglichst hohe *durchschnittliche* Verarbeitungsgeschwindigkeit und ein möglichst großer *Gesamt*datendurchsatz im Vordergrund stehen. Beim Design der Hardware werden zudem oft Kompromisse eingegangen, um die Rückwärtskompatibilität zu erhalten. Maßgeblich für das Design eines Standard–PCs ist der bedienende Anwender. Diesem fallen mögliche Blockierungen der Hardware erst bei einigen Zehntelsekunden auf.

Daher bleibt die Minimierung einzelner, sporadisch auftretender maximaler Laufzeiten und Blockierungen unberücksichtigt. Diese werden in einem PC-System nicht nur durch den verwendeten Prozessor selbst, sondern in zunehmendem Maße auch von der ihn umgebenden Hardware verursacht. Hierzu zählen, wie in Abb. 3.2 gezeigt, maßgeblich die Bausteine des Chipsatzes, wie die North–Bridge und die South–Bridge, die die Prozessoren eines klassischen PC–Systems mit dem Hauptspeicher und den zahlreichen Bussystemen verbinden. Die Auswirkungen der Bussysteme wurden bereits in [200, 68, 197] untersucht, die der Peripherieanbindung in [196] und die der Caches in [24].

3.4 Realzeiteigenschaften von Betriebssystemen

Das Betriebssystem ist der Teil der Software auf einem Computer, das die Ressourcen wie Prozessoren, Speicher und Ein-/Ausgabegeräte verwaltet und die Ausführung von Anwendungsprogrammen ermöglicht. Die Anwendungsprogramme werden vom Betriebssystem kontrolliert und nutzen dessen Dienste. Es wird unterschieden:

Standardbetriebssysteme (General Purpose Operating System, GPOS): Sie sind auf einer Vielzahl von Rechnersystemen einsetzbar. Bekannte Vertreter sind das kommerzielle Microsoft Windows oder das freie Unix-Derivat GNU/Linux. Ihr Verhalten ist speziell auf Durchsatz, Sicherheit und Bedienbarkeit optimiert. Das Zeitverhal-

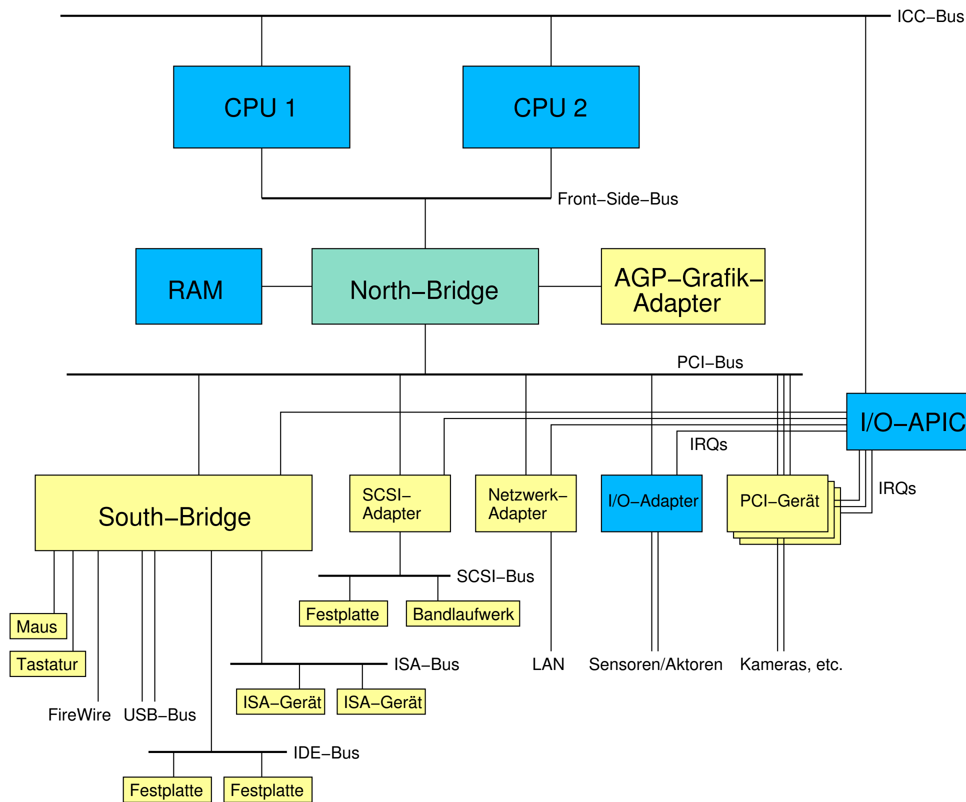


Abbildung 3.2: Hardwarearchitektur eines Mehrprozessor-PCs

ten im Extremfall spielt eine untergeordnete Rolle, solange es dem Benutzer nicht unangenehm auffällt.

Echtzeitbetriebssysteme (Real-Time Operating System, RTOS): Systeme wie z.B. LynxOS, VxWorks, pSOS+, QNX, eCos und RTEMS sind in erster Linie auf ihr Zeitverhalten hin optimiert. Denn Realzeitsoftware wird durch ihre maximale Laufzeit (*Worst-Case Execution Time*, WCET) charakterisiert. Daher muss der zeitliche Ablauf von Programmen genau gesteuert und vorhergesagt werden können. Konkurrierende Ziele, wie ein hoher Durchsatz, müssen dabei in den Hintergrund treten.

Ein wichtiges Element in einem RTOS ist der *Scheduler*, der bestimmt, wann welche Task zur Ausführung kommt. Das System kann mehrere Aufgaben quasi-gleichzeitig erledigen, sofern die gesamte Rechenlast gewisse Grenzen [58] nicht überschreitet. Dabei ist meist präemptives Multitasking möglich, durch das noch zeitkritischere Tasks gerade laufende Tasks unterbrechen können. Für die Scheduling-Entscheidung existieren zahlreiche Algorithmen [128], ein einfacher oft gewählter Ansatz ist eine statische Prioritätsvergabe.

Das RTOS muss in jedem Fall sicherstellen, dass die lauffähige Task mit der höchsten Priorität störungsfrei ablaufen kann. Daher darf es keine Stellen geben, wo es zu Blockierungen z.B. beim Datenaustausch oder durch externe Signale kommen kann. Dies wird durch den Einsatz von *Synchronisierungsprimitiven* wie Mutex (*Mutual*

Exclusion) und Semaphoren [45], in Kombination mit Scheduler-unterstützten Prioritätsprotokollen erreicht [128]. So sorgen das *Priority Inheritance Protocol* (PIP) und das *Priority Ceiling Protocol* (PCP) dafür, das Problem der Prioritätsinversion und das Entstehen von gegenseitigen Blockierungen (*Deadlock*) beim Zugriff auf kritische Ressourcen zu vermeiden [182].

Der Quellcode von GPOS ist gewöhnlich sehr umfangreich, bietet dafür viele Funktionen. RTOS sind meistens sehr schlank und auf die nötigsten Funktionen beschränkt. Es gibt zahlreiche Ansätze, GPOS um Echtzeitfähigkeiten zu erweiteren.

3.4.1 Standardbetriebsystem Linux

Das freie und unter einer OpenSource-Lizenz stehende GPOS GNU/Linux ist für eine Vielzahl verschiedener Prozessorplattformen verfügbar, darunter x86, MIPS, ARM, Alpha und PowerPC. Es wird nicht nur auf Desktop-Rechnern und Servern eingesetzt, sondern zunehmend auf eingebetteten Systemen wie Routern, Multimediageräten und Mobilcomputern [230, 191, 178].

Als GPOS ist der Betriebssystemkern (*Kernel*) von Linux auf gute durchschnittliche Antwortzeiten und guten Datendurchsatz optimiert. So wird in den Routinen zur Behandlung von Hardwareunterbrechungen (*Interrupt Request Handler*, IRQ-Handler) der Prozessor so konfiguriert, dass er bis zum Abschluss der Ausnahmebehandlung keine weiteren Unterbrechungen zulässt. Trifft kurz nach der Unterbrechungsanforderung eines unkritischen Gerätes (z.B. Tastatur) eine kritische Anforderung (z.B. Endlagenschalter einer Maschine) ein, so kann Letztere erst nach Ablauf der vorherigen behandelt werden. Des Weiteren ermöglicht es das Betriebssystem, sämtliche Unterbrechungsanforderungen gezielt zu sperren (`local_irq_disable()`), beispielsweise um kritische Programmabschnitte zu schützen oder die Konsistenz globaler Datenstrukturen zu sichern. Auf Mehrprozessorsystem werden zusätzlich aktive Warteschleifen (*Spinlocks*) eingesetzt (`spin_lock()`). Versucht ein Prozessor, einen Programmabschnitt zu betreten, der durch ein gesperrtes Spinlock gesichert ist, verfällt er in eine Warteschleife, in der er ununterbrochen prüft, ob die Sperre wieder freigegeben ist [19]. Für die Dauer des Wartens wechselt der Scheduler zu keinem anderen Task. Diese auf den ersten Blick als Verschwendung von Rechenzeit erscheinende Lösung führt jedoch dann zum optimalen Durchsatz, wenn die Sperrzeiten so kurz sind, dass der zeitliche Aufwand für einen Kontextwechsel des wartenden Prozessors und eine spätere Benachrichtigung höher sind.

Die längste mögliche Latenz bis zur Behandlung eines echtzeitkritischen Interrupts wird daher von dem längsten vorhandenen dermaßen geschützten Abschnitt hervorgerufen. Da der Linux-Kernel im Quellcode vorliegt [126], lassen sich theoretisch alle Bereiche identifizieren. Der Quellcode besteht jedoch aus $> 8 \cdot 10^6$ Zeilen (lines of code, LOC) [115], davon $> 50\%$ Gerätetreiber, um die erwähnte breite Hardwareunterstützung zu bieten.

Die Programmierschnittstelle (*Application Programming Interface*, API) des Kernels steht jedem Betriebssystemmodul offen [29]. In ein laufendes System können mit den entsprechenden Rechten jederzeit zusätzliche Module geladen werden. Durch die Erweiterung um ein zusätzliches Treibermodul, das ausgiebigen Gebrauch von Interruptsperren macht, kann sich das Zeitverhalten eines laufenden Systems im Betrieb verschlechtern.

Modifikationen zur Senkung der Latenzzeiten

Um das Zeitverhalten des Linux-Kernels zu verbessern, existieren Ansätze, die relevanten Stellen im Code zu modifizieren (*patch*). So ersetzt der „Real-Time Preemption Patch“ [143] die meisten Spinlocks durch Mutexes, die Priority Inheritance und Präemption ermöglichen. Dies erfordert jedoch Änderungen an > 500 Dateien. Eine weitere Modifikation lagert alle IRQ-Handler in eigene Threads aus, die dadurch ebenfalls unterbrechbar werden.

Durch diese Ansätze werden in den allermeisten Fällen deutlich bessere Latenzzeiten erreicht [187], können jedoch nicht garantiert werden. Bei Messungen treten sporadisch unerwartet hohe Latenzen auf, da aufgrund der Komplexität des Kernels nicht alle Kontrollflusspfade erfasst werden können. Außerdem fällt der Gesamtdatendurchsatz geringfügig ab. Solche Systeme eignen sich daher aktuell nur für weiche Echtzeitsysteme.

3.4.2 Echtzeiterweiterungen

Für harte Echtzeitanwendungen sind, wie in Kapitel 3.4 beschrieben, native RTOS verfügbar. Dennoch wird nach Wegen gesucht, bestehende GPOS wie Linux echtzeitfähig zu machen. Vorteile sind unter anderem eine große Basis an verfügbarer Software, an nutzbaren Programmbibliotheken und Gerätetreibern. Auf diese kann bei der Entwicklung neuer Applikationen zurückgegriffen werden und so können Entwicklungskosten gespart werden. Durch die große Zahl von Entwicklern, die Linux stetig weiterentwickeln, ist die zukünftige Verfügbarkeit aktueller Treiber für neue Hardware garantiert. Gerade für hybride Echtzeitsysteme, die sowohl einen echtzeitkritischen Teil und einen unkritischen Teil für die Benutzerschnittstelle benötigen, vereinfacht und verbilligt ein echtzeitfähiges GPOS die Entwicklung.

GNU/Linux unterliegt der GPL [56], der GNU General Public License, die beliebige Änderungen am Quellcode und deren Weitergabe erlaubt. Aufgrund dieser Open–Source Natur bietet sich Linux gerade auch für die Forschung an, da sich sämtliche Kontrollflusspfade im Quellcode analysieren lassen. Da auch umfangreiche Änderungen am Quellcode den Linux-Kernel bisher nicht nach der Definition von Kapitel 3.2 hart echtzeitfähig gemacht haben, dominiert der *Dual-Kernel-Ansatz* unter den verfügbaren Lösungen, der sich durch folgende Eigenschaften auszeichnet:

- Ein zusätzlicher schlanker Echtzeitkernel arbeitet neben dem eigentlichen Linux-Kernel.

- Zwischen der Hardware und dem eigentlichen Linux-Kernel ist ein weiterer Layer eingezogen. Dieser sorgt dafür, dass Echtzeitinterrupts direkt an den Echtzeitkernel weitergeleitet werden.
- Der Linux-Kernel läuft als Leerlaufprozess (*Idle-Task*) des Echtzeitkernels. Nur wenn keine Echtzeitaufgaben zu erledigen sind, kommen die GPOS-Applikationen zur Ausführung.
- Sämtliche in Abschnitt 3.4.1 beschriebenen Funktionen des Linux-Kernels zur Sperrung von Interrupts werden modifiziert, sodass lediglich die Auslieferung von Interrupts an Linux ausgesetzt werden kann, nicht aber das Auftreten höherpriorer Echtzeitinterrupts.

Einen Auszug aus den verfügbaren Lösungen gibt folgende Aufstellung:

RT-Linux: RT-Linux [231] ist eine der ältesten verbreiteten Linux-Echtzeiterweiterungen. Echtzeitprozesse werden als Kernel-Module implementiert und verwenden das API von RT-Linux. Kernel-Module unterliegen jedoch einigen Einschränkungen und haben insbesondere keinen Speicherschutz. Da das Verfahren, Linux u.a. um einen Interrupt-Emulator zu erweitern, um Interrupts zu verzögern, patentiert [232] ist, gibt es gegen RT-Linux in der OpenSource-Gemeinschaft Vorbehalte.

RTAI: RTAI (Real-Time Application Interface) [15] genießt eine höhere Akzeptanz als RT-Linux. Es wird der Ansatz verfolgt, möglichst kleine Änderungen am eigentlichen Linux-Kernel vorzunehmen. So werden Zeiger (*Pointer*) auf alle hardwarerelevanten Funktionsaufrufe (z.B. *cli*()/*sti*() (clear/set interrupts) bzw. *local_irq_disable*()...) und Datenbereiche von Linux in einer Struktur zusammengefasst (*Real-Time Hardware Abstraction Layer*, RTHAL). Beim Laden des eigentlichen Echtzeitbetriebssystems in Form eines Kernelmoduls werden diese Pointer lediglich auf dessen Funktionen umgesetzt.

Um Rechtssicherheit zum RTLinux-Patent zu schaffen, benutzt RTAI mittlerweile anstelle des RTHAL den im Folgenden beschriebenen ADEOS-Nanokernel [48].

RTAI-LXRT: Mit Hilfe von RTAI-LXRT [14] können Applikationen auch außerhalb des Kernels aus dem geschützten Benutzerbereich heraus das RTAI-API nutzen. Die Verwendung einiger Linux-eigener Diagnosewerkzeuge ist jedoch nicht möglich und kann wie auch die Verwendung von Linux-Signalen zu folgenschweren Abstürzen des Linux-Kernels führen.

ADEOS: ADEOS (*Adaptive Domain Environment for Operating Systems*) [228] ist kein richtiges RTOS, sondern ein Nanokernel, der in erster Linie der generischen Interruptverwaltung dient. Dazu werden alle Hard- und Softwareinterrupts über eine zentrale Ereignisleitung namens I-Pipe (*Interrupt-Pipeline*) geführt. An dieser können sich Betriebssysteme, in diesem Kontext *Domänen* genannt, mit festen Prioritäten registrieren. So ist es möglich, verschiedene GPOS parallel zu betreiben [227] oder ein RTOS mit hoher Priorität und ein GPOS mit niedriger Priorität gemeinsam [229].

ADEOS sorgt dafür, dass einzelne Domänen nur ihren eigenen Empfang von Interrupts abschalten und nicht die Hardwarequelle selbst.

Xenomai: Xenomai [65] war zeitweise Teil des RTAI-Projekts unter dem Namen RTAI/fusion und verfolgt den Ansatz der Zusammenarbeit mit dem Linux-Kernel, anstatt ihn zu isolieren. So können Entwicklungen wie der „Real-Time Preemption Patch" für weiche Echtzeittasks zusätzlich genutzt werden. Xenomai besitzt einen kleinen generischen Kern (*nukleus*), der erst durch die Verwendung eines *Skin* ein Benutzer-API bekommt. So können die APIs bestehender RTOS emuliert und die Portierung existierender Software erleichtert werden. Zur Wahl stehen u.A. POSIX, VxWorks, pSOS+ und ein natives API.

Unter Xenomai werden Echtzeitapplikationen gewöhnlich als Linux-Tasks gestartet und können beispielsweise nicht-echtzeitfähige Dateisystemfunktionen nutzen, um Konfigurationsdateien zu lesen. Dabei wird ihnen unter dem Linux-Scheduler lediglich weiche Echtzeit garantiert, sie befinden sich im *Secondary*-Modus. Sobald sie einen Echtzeitaufruf nutzen, kommen sie in den *Primary*-Modus, unterstehen dem Xenomai-Scheduler und werden hart echtzeitfähig. Solange sie Berechnung durchführen und nur Echtzeitaufrufe nutzen, behalten sie Ihren Status. Sie können jederzeit Linux-Systemaufrufe wie `write(STDOUT,...)` absetzen und Debugging-Techniken, wie `ptrace()` nutzen. Dabei werden sie in den *Secondary*-Modus zurückversetzt und wieder vom Linux-Scheduler verwaltet.

Ein weiterer Ansatz ist die Verwendung von Mikrokerneln, wie L4 [125] und dem echtzeitfähigen Fiasco [92]. Diese sind aufgrund ihres überschaubaren Umfangs in ihrem Verhalten deutlich deterministischer als ein GPOS, haben aber gewöhnlich ihr eigenes API. [141] beschreibt eine Emulation des RT-Linux APIs für einen L4-Mikrokern, [94] den Betrieb eines „gezähmten" Linux [95] unter L4. Damit sind gleichzeitig der harte Echtzeitbetrieb und der Ablauf von Standardapplikationen möglich. Ziel der Kapselung von Linux ist die Isolation zwischen den Echtzeittasks und den Standardtasks [96].

Aufgrund der strikten Trennung zwischen Echtzeit und nicht-Echtzeit-Domänen ist in allen gezeigten Varianten die Kommunikation zwischen den Domänen aufwendig, und für jede Task muss eine feste Zuordnung getroffen werden. Darüber existieren meist unterschiedliche APIs für beide Teile, was den Bruch verstärkt und eine Integration erschwert.

3.5 Echtzeiteigenschaften der Speicherverwaltung

Ein Computerprogramm benötigt für seine Arbeit Speicher. Je mehr Daten verwaltet werden müssen, umso mehr Hauptspeicher (*Random Access Memory*, RAM) ist notwendig. Lediglich der Programmcode kann in einem Nur-Lesespeicher (*Read Only Memory*, ROM) untergebracht werden. Eine Instanz eines Programms, das sich zur Ausführung im Speicher befindet, ist ein Prozess. Die Anordnung der Daten eines Prozesses im Speicher wird *Speicherlayout* [193] genannt. Dieses besteht in der Regel der Reihe nach aus folgenden Abschnitten:

- Das *Text Segment* enthält den eigentlichen Programmcode. Es sollte gerade in kritischen Echtzeitsystemen nur lesbar sein, da selbst-modifizierende Programme einen Echtzeitnachweis unnötig erschweren. Der Inhalt des Text Segments wird meist erst bei Bedarf aus der Programmdatei im Rahmen des Seitenaustauschverfahrens (*demand paging*) automatisch nachgeladen.
- Das *Datensegment* beinhaltet globale Daten. Dies sind zum einen vordefinierte Konstanten als auch nicht initialisierte Variablen (bezeichnet als BSS, *Block Started by Symbol*).
- Aus dem *Haldenspeicher* (engl. Heap) werden dynamische Speicheranforderungen des Programms zur Laufzeit bedient. Dieser wird explizit angefordert (z.B. über die Funktion `malloc()` der C-Laufzeitbibliothek) und muss auch wieder explizit freigegeben werden (`free()`), sonst entstehen Speicherlecks (*Memory Leaks*), die u.U. erst nach einiger Laufzeit auffallen. Für die Verwaltung des Haldenspeichers existieren zahlreiche Algorithmen zur dynamischen Speicherzuteilung (*Dynamic Storage Allocation*, DSA). Die Größe des Haldenspeichers ist flexibel und wird von der C-Laufzeitbibliothek bei Bedarf vergrößert, unter Linux z.B. mit dem Systemaufruf `brk()`.
- Der *Kellerspeicher* (*Stack*) dient zum einen beim Aufruf von Funktionen zur Zwischenspeicherung der Rücksprungadresse und ggf. der Übergabe von Parametern. Zum anderen werden dort Daten gespeichert, die nur lokal sichtbar sind und nach Abschluss eines Programmblocks wieder gelöscht werden. Die Verwaltung des Stacks geschieht automatisch durch den Compiler, die Größe der dort abgelegten Datenstrukturen ist schon zur Übersetzungszeit bekannt. Der Stack wächst ausgehend von hohen Adressen nach unten, neuer Speicher wird linear vergeben und wird prinzipbedingt in der gleichen Reihenfolge wieder freigegeben. Die Rechenzeit der Stackverwaltung ist daher gering.

Aus der Sicht von Echtzeitsystemen verursacht die Speicherverwaltung zwei Probleme: Zum einen muss sichergestellt werden, dass jede Speicheranforderung erfüllt wird, zum anderen muss die dafür notwendige Rechenzeit beschränkt sein. Daher wird schon in der ersten Version von MISRA-C [142], einem Standard aus der Automobilindustrie für sicheres Programmieren zur Vermeidung von Laufzeitfehlern, die dynamische Speicheranforderung ausgeschlossen:

- Um einen Programmabbruch wegen Stapelüberlaufs zu vermeiden, werden in Regel 70 („Functions shall not call themselves, either directly or indirectly“) rekursive Funktionen verboten. Damit ist nach Analyse aller möglichen Programmpfade eine maximale Stapelgröße unabhängig von den Eingangsdaten berechenbar.
- In Programmierregel 118 („Dynamic heap memory allocation shall not be used“) wird die dynamische Speicheranforderung aus dem Haldenspeicher zur Laufzeit (z.B. `malloc()`) untersagt.

Bei Einhaltung dieser Regeln können keine Laufzeitprobleme durch dynamische Speicherverwaltung auftreten. Bei der Abbildung hochdynamischer Umgebungen in einem Compu-

ter tritt jedoch die Frage auf, wie diese dann in den Algorithmen abgebildet werden sollen. Oft wird der Ansatz verfolgt, beispielsweise eine maximal mögliche Anzahl von Objekten zu erlauben. Wird diese Anzahl überschritten, kann es zu Fehlfunktionen des Systems kommen. Damit ist das umgangene Problem der dynamischen Speicherverwaltung nur verlagert worden.

3.5.1 Dynamische Speichervergabe

Zur Lösung der dynamischen Speicherzuteilung (*Dynamic Storage Allocation*, DSA) existieren zahlreiche Algorithmen [224], an die von Echtzeitsystemen teilweise konkurrierende Anforderungen gestellt werden [161]:

- Die Reihenfolge und die Größe der Speicheranforderungen und -freigaben zur Laufzeit sind variabel und a priori nicht bekannt, da sie vom technischen System bestimmt werden. So entstehen freie „Löcher" im belegten Speicher. Weil ein einmal vergebener Speicher nicht ohne Softwareaufwand wie in [223] oder Hardwareunterstützung wie in [225] verschoben werden kann, kann diese Fragmentierung über die Zeit zunehmen. Zur Minimierung der Speicherfragmentierung existieren verschiedene Strategien, eine aktuelle Evaluierung echtzeitrelevanter Algorithmen findet sich in [30].
- Der Zeitbedarf für eine beliebige Speicheranforderung oder -freigabe muss zu jedem Zeitpunkt beschränkt und unabhängig von der Vorgeschichte sein. Dabei darf die Schranke auch nicht zu hoch sein. [133] vergleicht dazu die Ausführungszeiten verschiedener DSA-Algorithmen im Worst-Case sowohl analytisch als auch in Messungen.

In GPOS ist meist die durchschnittliche Zeit (*Average Case*) für eine Anforderungs- bzw. Freigabeoperation ausschlaggebend. So enthält ein Algorithmus beispielsweise eine heuristische Verbesserung durch Aufschieben der Wiedervereinigung freier Speicherblöcke [120] (*Deferred Coalescing*), die aber im Worst-Case, signifikant längere Laufzeiten kostet. Da viele der auf GPOS ausgeführten Programme nur eine Laufzeit von wenigen Minuten oder Stunden haben, bis sie beendet werden und ihr gesamter Speicher wieder freigegeben wird, tritt dieser Fall dort oftmals gar nicht erst auf.

3.5.2 Echtzeitfähigkeit von Speicherverwaltungsalgorithmen

Für harte Realzeitsysteme ist bei jeder Operation jedoch stets der Worst-Case ausschlaggebend. In [134] wird ein zeitbeschränkter *good-fit* Allokationsalgorithmus vorgestellt, der *Two-Level Segregated Fit Memory Allocator* (TLSF) genannt wird und in [135] ausführlich beschrieben wird. Dabei werden freie Blöcke ihrer Größe nach sortiert und in einem zweistufigen Listensystem verwaltet. *Good-fit* ist ein Kompromiss bzgl. der Fragmentierung, der bei der Suche nach einem freien Block deterministisch ist, aber im Gegensatz zu *best-*

fit nicht immer den kleinstmöglichen findet, und so je nach Anfrage eine bestimmbare Menge „Speicherverschnitt“ verschenken kann.

Jeder Speicherblock beginnt mit einem eingebetteten Kopfteil („Header“), der seine Größe und einen Zeiger zum Kopf des vorhergehenden Blocks enthält. Dadurch können Speicherblöcke bei der Freigabe unter festem kurzen Rechenaufwand mit ihrem freien Vorgänger oder Nachfolger wiedervereint werden. Diese Technik wird in [111] als *Boundary Tags* beschrieben.

Die Komplexität sowohl für Anforderung als auch Freigabe von Speicherbereichen ist von der Größenordnung von O(1), bei brauchbaren durchschnittlichen Ausführungszeiten. Die Fragmentierung im Worst-Case wird mit $< 30\%$ angegeben [133].

3.5.3 Speicherzuteilung im Betriebssystem Linux

In Betriebssystemen muss die Speicherverwaltung die Eigenheiten der darunter liegenden Hardware berücksichtigen. Außer in sehr kleinen eingebetteten Systemen besitzen die meisten Prozessorarchitekturen eine Verwaltungseinheit für virtuellen Speicher (*Memory Management Unit*, MMU). Diese setzt die logischen virtuellen Adressen der Software in physikalische Adressen der Hardware um. Die kleinste dabei verwendete Speichereinheit ist eine Speicherseite (*page*) [204], deren Größe hardwareabhängig ist. Beim Einsatz des Betriebssystems Linux auf einem Intel Pentium Prozessor wird beispielsweise eine Speicherseitengröße von 4096 Bytes verwendet.

Buddy Allocator

Unter dem untersuchten GPOS Linux kommt für die Vergabe von Speicher ein hierarchischer Ansatz zum Einsatz [19, 132]. Auf oberster Stufe werden ganze Speicherseiten von einem *Binary Buddy Allocator* basierend auf [111] verwaltet. Dieser führt Listen von freien physikalisch zusammenhängenden Seitenbereichen in den binären Größenstaffelungen von 2^x mit $x \in \{0, ..., 10\}$. Zur Allokation eines Speicherbereichs von x Seiten wird folgender Algorithmus eingesetzt:

- Die Anzahl angeforderter Seiten x wird zu $\hat{x} = \lceil \log_2 x \rceil$ auf die nächste Zweierpotenz $2^{\hat{x}}$ aufgerundet.
- In der Liste zu $2^{\hat{x}}$ wird ein Eintrag entfernt. Dieser zeigt auf $2^{\hat{x}}$ freie Seiten. Wenn die Liste leer ist, wird in den Listen der nächstgrößeren Bereiche $2^{\hat{x}+1}$, $2^{\hat{x}+2}$,... gesucht.
- Die verbleibenden $\hat{x} - x$ Seiten werden in möglichst große Einheiten von 2^i aufgespalten und in die entsprechenden Frei-Listen wieder einsortiert.

Beim Freigeben von Speicherseiten ist zu beachten, dass ein Block aus x Seiten wieder in $2^{\hat{x}}$-Einheiten in die Frei-Listen eingetragen wird. Werden dabei zwei aufeinanderfolgende $2^{\hat{x}}$-Blöcke (sog. Kameraden, engl. „Buddies“), deren Startadresse an einem Vielfachen von $2^{\hat{x}+1}$ beginnt, gefunden, können diese zu einem $2^{\hat{x}+1}$-Block wiedervereinigt werden.

Durch den „Buddy Allocator“ wird externe Fragmentierung vermieden, jedoch durch interne Fragmentierung erkauft, da jede Anfrage auf ganze Seiten aufgerundet wird. Zudem müssen bei der Vergabe physikalisch zusammenhängender Seitenbereiche keine Seitentabellen zusätzlich modifiziert werden. Im ungünstigsten Fall, der aber für Echtzeitsysteme ausschlaggebend ist, muss für eine Speicherseite ein Block aus 1024 Seiten iterativ geteilt und die überschüssigen Seiten wieder auf die kleineren Listen verteilt werden.

Slab Allocator

Auf zweiter Stufe wird im Linux Kernel ein sogenannter *Slab Allocator* [18] eingesetzt. In großer Zahl benötigte Datenstrukturen des Betriebssystems wie Task-Deskriptoren (`task_struct`), Verzeichnis- und Dateiinformationen (`dentry/inode_cache`) sind deutlich kleiner als eine Speicherseite. Daher werden mehrere gleichartige Elemente auf einmal angelegt, für diese der benötige Speicher in ganzen Seiten angefordert und in einem Schnellzugriffsspeicher (*Cache*) vorgehalten. Wird vom Betriebssystem ein weiteres dieser Elemente benötigt, kann diese Anfrage meistens aus dem *Slab Cache* befriedigt werden.

Durch den *Slab Allocator* wird der Durchsatz eines GPOS erhöht, im Worst-Case fällt jedoch die Zeit für die Erweiterung des *Slab Caches* um neue Elemente und die dazu erforderliche Allokation von Speicherseiten mittels des o.g. *Buddy Allocators* an.

3.6 Kognitive Automobile

Der Begriff der Kognition stammt aus dem lateinischen Wort *cognoscere* für „erkennen“. Nach [16] ist er der „Sammelbegriff für alle Prozesse und Strukturen, die mit dem Wahrnehmen und Erkennen zusammenhängen“. Die menschliche Kognition umfasst die Wahrnehmung, in der beispielsweise das Umfeld visuell erfasst wird, bis hin zum Verständnis der Umwelt und der Interpretation der Aktionen anderer Menschen. Aufgrund erlangter Erkenntnisse können dann zielgerichtete Aktionen ausgeführt werden. Langfristig nähren diese Erkenntnisse auch einen Lernprozess, in dem das eigene Verhalten sukzessive optimiert wird.

3.6.1 Sonderforschungsbereich/Transregio 28

Der im Jahr 2006 von der Deutschen Forschungsgemeinschaft (DFG) eingerichtete Sonderforschungsbereich/Transregio 28 (SFB/TR 28) „Kognitive Automobile“ hat zum einen das Ziel, die maschinelle Kognition mobiler Systeme als Grundlage maschinellen Handelns systematisch und interdisziplinär zu erforschen. Zum anderen soll dies anhand von Automobilen im Straßenverkehr demonstriert werden: Das Verhalten anderer Verkehrsteilnehmer muss erfasst und verstanden werden. Daraus muss sodann ein eigenes Verhalten generiert werden [195].

In *intelligenten Fahrzeugen* (siehe Kapitel 2) werden oftmals einzelne Funktionen demonstriert. Ein *Kognitives Automobil* umfasst ein Gesamtsystem aus vielfältigen Funktionen, es kennt seine eigenen Fähigkeiten und setzt diese wissentlich ein. Dafür muss es seine Umwelt wahrnehmen, sinnvolle Entscheidungen treffen und in Echtzeit handeln. Darüber hinaus soll es sein Wissen über den Straßenverkehr selbständig erweitern und strukturieren.

Kooperative Kognitive Automobile [194] sind zudem in der Lage, ihr Wissen mit anderen Fahrzeugen zu teilen und sich so ein umfassenderes gemeinsames Lagebild zu erstellen, als es einem menschlichen Fahrer möglich ist. Sie können zudem ihr Handeln gemeinsam planen und auszuführen. Für den Straßenverkehr bedeutet dies einen deutlichen Effizienz- und Sicherheitsgewinn.

3.6.2 Projektüberblick

Projektpartner im SFB/TR 28 sind die Universität Karlsruhe (TH), die Technische Universität München, die Universität der Bundeswehr München und das Fraunhofer Institut IITB Karlsruhe. In 12 Projekten werden interdisziplinär an 12 Instituten wissenschaftliche Fragestellungen kognitiver Automobile untersucht.

Die am SFB/TR 28 in der ersten Phase in den Jahren 2006-2009 beteiligten und genehmigten Teilprojekte [190] sind in 3 Projektbereiche gegliedert und decken folgende Themen ab:

- Projektbereich A: Verteilte sensorielle Wahrnehmung
  - A1: Aufmerksamkeitsgesteuerte mono-/stereoskopische Wahrnehmung komplexer Verkehrsräume
  - A2: Entdeckung, Klassifikation und Zustandsschätzung verkehrsrelevanter Objekte
  - A3: Integration komplementärer Sensorik und Sensordatenfusion
  - A4: Kooperative Wahrnehmung vernetzter Kognitiver Automobile
- Projektbereich B: Verhaltensentscheidung, -planung, -ausführung
  - B1: Situationsbewertung und Verhaltenserkennung
  - B2: Kognitive Verhaltensentscheidung und Bahnplanung
  - B3: Verteilte Kooperation
  - B7: Sicherheitsbewertung von (autonomen) Verhaltensentscheidungen durch Methoden der hybriden Verifikation
- Projektbereich C: Fahrzeuge und IT-Basissystem
  - C1: Fahrzeugbereitstellung und Einsatzsicherheit
  - C2: Fahrzeugausrüstung und -bereitstellung

– C3: Hardware-/Software-Architektur (Schwerpunkt dieser Arbeit)

– C4: Kommunikation zwischen Fahrzeugen

Die Forschungsergebnisse dieser Teilprojekte müssen für gemeinsame Simulationen und für die Demonstration an realen Fahrzeugen zu einem Gesamtsystem integriert werden. Dies ist insbesondere für die Bewältigung komplexer Fahraufgaben essentiell. Zu einer erfolgreichen Integration leistet das Architekturteilprojekt C3 einen wichtigen Beitrag, dessen wissenschaftliche Ergebnisse sind in diese Arbeit eingeflossen.

3.6.3 Funktionale Architektur

Abb. 3.3 gibt einen Überblick über die funktionale Architektur eines kognitiven Automobils. Sie lehnt sich an die in [137] vorgeschlagene Architektur an, und ist in der Lage, Module zu integrieren, die den 4D-Ansatz [38] implementieren. Sie unterstützt jedoch auch alternative Ansätze, die im SFB/TR 28 untersucht werden. Durch die Kombination verschiedener Methoden lassen sich so noch bessere kognitive Ergebnisse erzielen.

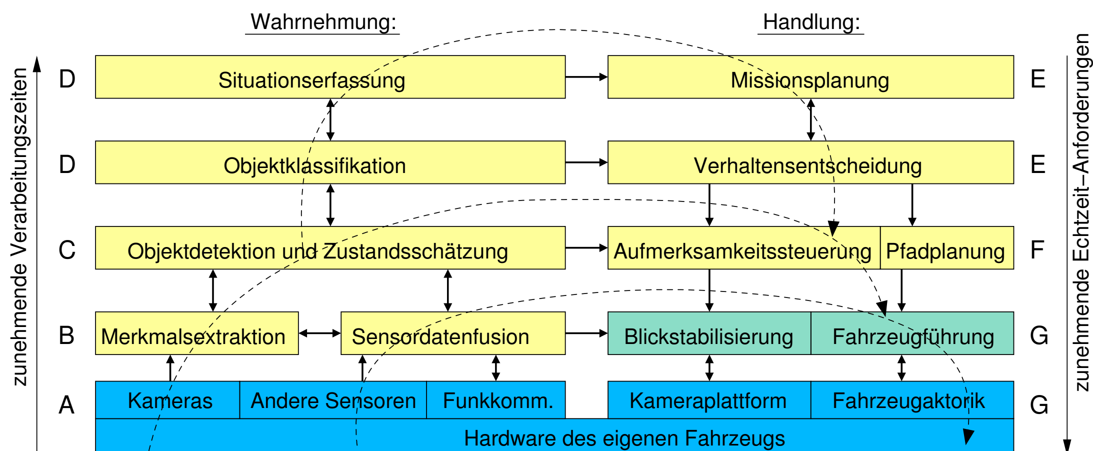


Abbildung 3.3: Funktionale Architektur eines kognitiven Automobils

Die in Abb. 3.3 gezeigten Funktionen lassen sich in Bereiche mit folgenden charakteristischen Eigenschaften einteilen. Dabei wird zwischen einer Wahrnehmungsseite (links) und einer Handlungsseite (rechts) unterschieden.

A. Sensorschnittstellen:
Die Rohdaten der Sensoren werden in ein Rechnersystem eingelesen und der auswertenden Software zur Verfügung gestellt. Dies geschieht oftmals durch herstellereigene Programmierschnittstellen. Wenn die Schnittstellenfunktionen von verarbeitender Software separiert sind, können die Sensordaten universell verwendet werden.

B. Rohdatenauswertung:
Zu dieser Kategorie gehören alle Wahrnehmungsmodule, die auf Rohdaten rechnen. In der Bildverarbeitung sind das Messungen auf Pixelebene, wie z.B. Merk-

malsextraktion und -verfolgung [49, 149] oder die Erstellung von Tiefenkarten aus Stereobildern [31]. In der Auswertung von LIDAR-Daten sind das Entfernungsmessungen [100]. Dies umfasst ebenso die Fusion auf Rohdatenebene. Module, die auf dem 4D-Ansatz [41] basieren, beinhalten die modellbasierte Ebene für ihre internen Zustände. Die Modellzustände können, wenn sie verfügbar gemacht werden, durch andere Ansätze genutzt werden, und die Modelle durch komplementäre Algorithmen gestützt werden.

C. Modellbasierte Sicht und Zustandsschätzung:
Diese Ebene beinhaltet Module, die Merkmale zu Objekthypothesen gruppieren und sie über die Zeit plausibilisieren. Dabei können Merkmale aus verschiedenen Sensorsystemen [207] oder von anderen Fahrzeugen übermittelte [147] Informationen unter Anwendung von Fusionsmethoden miteinbezogen werden. Das Ergebnis sind erkannte Objekte mit ihren Bewegungszuständen.

D. Wissensbasierte Ebene mit Situationsanalyse:
Auf dieser Ebene wird das Verhalten von erkannten Objekten und dessen Auswirkungen unter Verwendung von Vorwissen analysiert [219]. Dazu wird meist eine Objektklassifikation durchgeführt, das Ergebnis ist eine Situationsbewertung [214]. Oftmals wird zuvor eine qualitative Beschreibung der Situation erstellt [118].

E. Wissensbasierte Handlungsentscheidungen:
Auf der obersten Handlungsebene benötigt das kognitive Automobil einen Plan oder eine zu erfüllende Mission. Im Wettbewerb *DARPA Urban Challenge 2007* [33] aus Kapitel 2.2 bestand die Aufgabe darin, eine Liste von Kontrollpunkten abzufahren. Unter Einbeziehung des Umgebungswissens kann das Fahrzeug dann entscheiden, was als Nächstes zu tun ist [66]. Dies führt zur Erzeugung eines angemessen Verhaltens [179]. Dies kann regelbasiert geschehen wie in [137, 138]. Bei der Entscheidungsfindung sind die eigenen Fähigkeiten mit einzubeziehen [234]. Dazu zählen die Funktionsfähigkeit benötigter Teilsysteme und die Überprüfung, ob überhaupt mit aktuellen Daten gearbeitet wird.

F. Handlung auf systemdynamischer Ebene:
Hier wird die abstrakte Verhaltensentscheidung in ein konkretes Fahrkommando umgesetzt, beispielsweise ausgedrückt durch eine vorgegebene Fahrtrajektorie [221] als Ergebnis einer Pfadplanung [177]. Für eine aktive Kameraplattform muss die Aufmerksamkeitssteuerung entscheiden, wohin die nächste Blickzuwendung zu erfolgen hat.

G. Aktorikschnittstellen und unterlagerte Regelkreise:
In vielen Fällen besitzen die Fahrzeugaktoren einen angegliederten unterlagerten Regelkreis, beispielsweise um eine kommandierte Lenkrate zu erzielen oder den Bremsdruck für eine vorgegebene Verzögerung einzuregeln.

Dieser Überblick über die enthaltenen Funktionen erhebt keinen Anspruch auf Vollständigkeit im SFB/TR 28, sondern soll vor allem die unterschiedlichen Wahrnehmungsebenen und ihre Beziehungen untereinander darstellen. So ist zu erkennen, dass der kaskadierte

Regelkreis von Umwelt–Wahrnehmung–Handlung–Umwelt, angedeutet durch Pfeile, an mehreren Stellen und auf unterschiedlichen Ebenen geschlossen wird:

- Nahe der physikalischen Ebene (unterster Regelkreis) sorgen hochfrequente Regelungen für das Einregeln von Sollgrößen, die von höherer Ebene vorgegeben werden. Darunter fällt z.B. die Nickwinkelstabilisation einer Telekamera, die auf Sensorseite einen Drehratensensor und auf Aktorseite einen Motor benötigt, der zum Ausgleich entweder eine Videokamera direkt bewegt, oder das Bild mittels eines Spiegels stabilisiert. Ein weiteres Beispiel ist das Einregeln einer Soll-Geschwindigkeit mittels einer Beschleunigungsvorgabe unter Verwendung einer gemessenen Ist-Geschwindigkeit.

  Um in beiden Fällen eine stabile Regelung zu erreichen, kann der Regelkreis nicht über die Wissensebene geschlossen werden, sondern benötigt die gezeigte Abkürzung auf unterster Ebene. Denn die beteiligten Funktionen haben, verglichen mit der Wissensebene, kurze Rechenzeiten bei hohen Echtzeitanforderungen.

- Auf der systemdynamischen Ebene müssen die Funktionen quantitative Ergebnisse liefern. Da die Wissensebene jedoch oftmals nur qualitative Vorgaben liefert, werden zusätzlich die quantitativen Daten der Wahrnehmung benötigt, wodurch sich eine weitere Schleife schließt.

Die anspruchsvolle Aufgabe einer Systemarchitektur ist nun, die Koexistenz dieser Wahrnehmungs- und Regelkreise mit ihren verschiedenen Anforderungen zu ermöglichen, ohne harte Brüche zwischen den genannten Wahrnehmungs- und Handlungsebenen zu verursachen.

4 Hardwareplattform für kognitive Fahrzeuge

In diesem Abschnitt werden die Hardwareanforderungen kognitiver Funktionen gesammelt und daraus eine Hardwareplattform für entsprechende Fahrzeuge abgeleitet. Im Mittelpunkt steht dabei die Auswahl und Analyse eines geeigneten leistungsfähigen Rechnersystems. Als Ausgangspunkt dient die Architektur heutiger Fahrerassistenzsysteme in Fahrzeugen.

4.1 Heutige Bordnetzarchitekturen

In heutigen Serienfahrzeugen besitzen verbaute Funktionen aktuell noch eine recht bodenständige Wahrnehmung mit begrenzten Aktionen. Sie werden unter der Bezeichnung Fahrerassistenzsysteme (FAS) zusammengefasst. Trennbare FAS-Funktionen wie z. B. ein Abstandsregeltempomat (*Adaptive Cruise Control*, ACC) oder ein Spurverlassenswarner (*Lane Departure Warning*, LDW) werden bevorzugt in separate, geschlossene Steuergeräte gekapselt, die ihren primären Sensor wie einen RADAR-Sensor oder eine Kamera enthalten. Die Integration geschieht durch Verbinden mit einem der verbauten Fahrzeugbusse.

Unter Architektur wird in diesem Kontext die Vernetzung der Steuergeräte verstanden und von *Bordnetzen* gesprochen. Schnittstelle ist das Bussystem wie CAN oder FlexRay [50]. Sogar der kommende AUTOSAR-Standard [88] spricht von einem *Virtual Functional Bus*. Dabei stehen meist Kostenaspekte im Mittelpunkt. Die Modularität auf Steuergeräteebene bietet den Vorteil, dass die meisten FAS-Funktionen als optionale Sonderausstattung verkauft werden können und so den Grundpreis des Basismodells nicht in die Höhe treiben. Allerdings zeigt sich, dass Funktionen zunehmend voneinander abhängig sind, sodass die Wahl einer höherwertigen Funktion die Hinzunahme einer Anzahl von Basisfunktionen erfordert. Die Maximalkonfiguration eines Bordnetzes ist zudem bei Produktionsstart festgelegt.

In zukünftigen Fahrerassistenzsystemen besteht jedoch der Trend zu einer engen Vernetzung der Funktionen: Gerade sicherheitsrelevante Systeme erfordern von der Wahrnehmung eine hohe Verlässlichkeit, die oftmals nur durch Einbeziehung verschiedener Informationsquellen erreicht werden kann. Die optimale Erkennungsleistung ergäbe die Fusion der Daten aller verbauten Sensoren, jeweils ausgewertet mit funktionsspezifischen Algorithmen. Dies setzt jedoch die Verfügbarkeit aller Informationen einschließlich der Sensorrohdaten voraus, was in heutigen Bordnetzarchitekturen noch nicht vorgesehen ist.

4.2 Hardwarearchitektur für ein kognitives Automobil

Die Entwicklung neuer kognitiver Funktionen benötigt eine funktionsunabhängige und offene Architektur, die genügend Schnittstellen zum Nachrüsten neuer Funktionen bietet. Ein kognitives Automobil muss, basierend auf der Wahrnehmung seiner Umgebung, seine eigene Situation verstehen und sich angemessen verhalten. Die dazu nötigen Funktionen des Fahrzeugs setzen sich aus vielen einzelnen Kognitionsmodulen zusammen, die in ein Gesamtsystem integriert werden müssen. Aus Gründen der Flexibilität wird daher die Funktionsintegration auf Softwarelevel angestrebt.

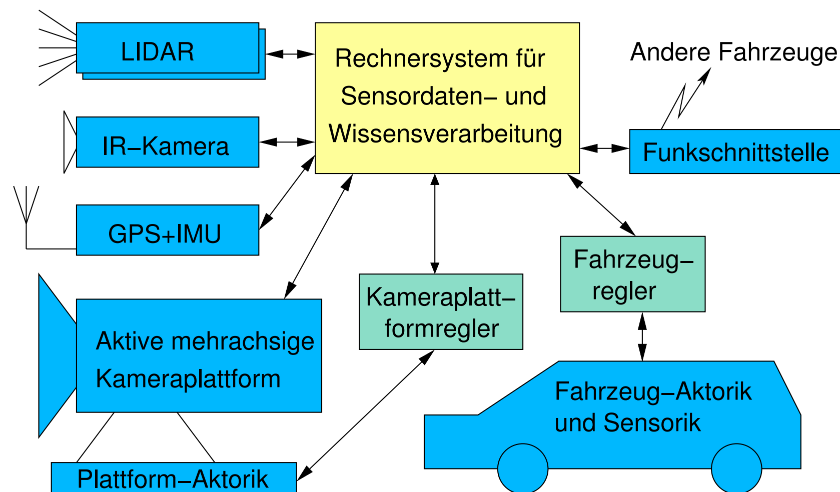


Abbildung 4.1: Hardwarearchitektur eines kognitiven Automobils

Abb. 4.1 zeigt die Hardwarearchitektur, wie sie in den kognitiven Automobilen des SFB/TR 28 aus Kapitel 6.4 zum Einsatz kommt. Auf der linken Seite befindet sich eine Auswahl von Sensoren zur Umfeldwahrnehmung: Eine Kameraplattform mit aktiver Blickrichtungssteuerung dient der videobasierten Erkennung von Fahrspuren, Fahrzeugen und sonstiger Objekte. Zwei in der Gierachse bewegliche Weitwinkelkameras decken das Nahfeld ab und dienen der Tiefenschätzung nach [31], eine zur Blickstabilisierung zusätzlich in der Nickachse bewegliche Telekamera liefert Bilder für die frühe Erkennung entfernter Objekte. LIDAR-Sensoren (*Light Detection and Ranging*) liefern Entfernungsdaten zur ergänzenden Objektdetektion. Ein Trägheitsnavigationssystem (*Inertial Navigation System, INS*) liefert eine exakte Positionsreferenz basierend auf Satellitenpositionsdaten des GPS (*Global Positioning System*) und Eigenbewegungsmessungen der IMU (*Inertial Measurement Unit*).

Sämtliche Aufgaben der Bild- und Wissensverarbeitung erledigt ein Fahrzeugrechnersystem, an das sämtliche Sensoren ihre Rohdaten liefern. Eine dSpace Autobox bindet die Fahrzeugsensorik und -aktorik an und dient der Sicherheitsüberwachung [218]. Ein eingebettetes System (MPC) regelt die vier Achsen der Kameraplattform (siehe Abb. 6.10(a)) und ist dort insbesondere für die Nickwinkelstabilisierung der Telekamera verantwortlich. Dies geschieht wie in [87] bioanalog durch technische Nachbildung des Vestibulo-Okulären

Reflexes (VOR) des menschlichen Auges. So wird einem Verschmieren des Videobildes während der Belichtung bei schwachen Lichtverhältnissen entgegengewirkt.

4.3 Fahrzeugrechnersystem

Auf dem Fahrzeugrechnersystem müssen die Softwarekomponenten aller kognitiven Funktionen ausgeführt werden, die jedoch recht unterschiedliche Anforderungen stellen:

- Die videobasierte Spurhaltung verlangt *kurze Latenzzeiten* für eine stabile Regelung.
- Eine umfassende Umgebungserfassung benötigt *viele Kameras*.
- Die Datenaufzeichnung schreibt mit *hoher Datenrate* auf Festplatte.
- Die Wissensverarbeitung möchte *große Speicherbereiche* schnell durchsuchen.
- Zur effizienten Nutzung des im Fahrzeug vorhandenen Umfeldwissens muss jedes Modul schnell auf *alle Daten* anderer Module zugreifen.

Durch den Einsatz mehrerer vernetzter Rechner können zwar beliebige Leistungsstufen realisiert werden, bei intensiver Kommunikation von Modulen über Rechnergrenzen hinweg können sich jedoch die entstehenden Kommunikationstotzeiten zum beschränkenden Faktor entwickeln. Zudem müssen Daten für die Übertragung serialisiert und zur Bestimmung der Übertragungsabfolge priorisiert werden.

Für das Ziel, eine enge Kooperation zu fördern, darf jedoch kein nennenswerter Kommunikationsaufwand entstehen, um im Fahrzeug verfügbares Wissen abzurufen. Eine Partitionierung in mehrere Rechnersysteme erhöht zudem die Gefahr der Entstehung von später nicht mehr zu vereinheitlichenden Software- und Hardwareinselsystemen, welche die Entwicklung neuer übergreifender Funktionen erschweren. Der Verzicht auf ein Rechnernetz erlaubt zudem eine Softwarearchitektur, wie sie in Kapitel 5 vorgestellt wird.

Eine weitere Anforderung an das Fahrzeugrechnersystem ist eine günstige Duplizierbarkeit. Deswegen werden bevorzugt Standardkomponenten (*Commercial-off-the-Shelf, COTS*) eingesetzt, deren Echtzeiteigenschaften bereits u. a. in [200, 197, 198] untersucht wurden. Der Preis schneller Clusterschnittstellen, wie das in Kapitel 2.1.1 eingesetzte SCI (*Scalable Coherent Interface*), bleibt aufgrund des fehlenden Massenmarkts so hoch, dass es für Projektpartner unattraktiv wäre, eine identische Konfiguration als Laborsystem zu betreiben. Für eine spätere Integration ist dies jedoch sehr förderlich.

4.3.1 Ausgewählte AMD Opteron Architektur

Den Kern des Entwicklungssystems bildet ein zentraler Mehrkern-Mehrprozessor-Rechner (*multicore-multinode*), auf dem alle Wahrnehmungsmodule ausgeführt werden. In [72] wurden moderne Multiprozessorarchitekturen verglichen und AMD Opteron NUMA-Systeme

als die am besten skalierende x86-Architektur ausgewählt. Hier können weitere Prozessoren ohne Chipsatzeinschränkung hinzugefügt werden, weil dafür keine spezielle Northbridge mit begrenzendem Front-Side-Bus benötigt wird, wie sie in Kapitel 3.3 für SMP-Systeme vorgestellt wurde.

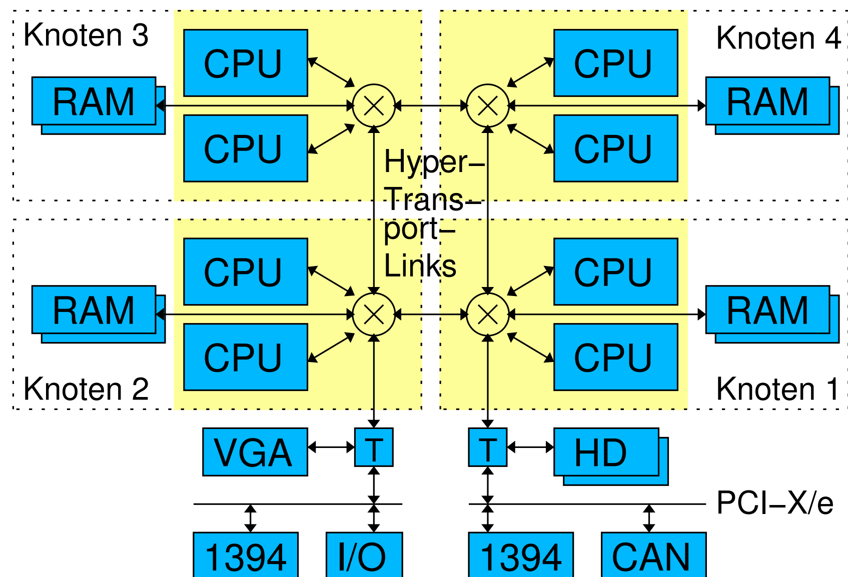


Abbildung 4.2: Hardwarearchitektur eines AMD Opteron Systems

Die Struktur des gewählten Rechnersystems veranschaulicht Abb. 4.2: Jeder „Knoten“ besteht aus einem Opteron-Prozessor und hat einen eigenen Speichercontroller mit lokalem Speicher „RAM“. Über einen schnellen Verteiler „X“ (*Crossbar-Switch*) sind die Rechenkerne „CPU“ mit diesem verbunden. Jeder Prozessor hat drei bidirektionale *HyperTransport*-Verbindungen [209] (*HT-Links*) nach außen, über die er auf den Speicher anderer Prozessoren und auf Peripheriegeräte zugreifen kann. Existiert keine direkte Verbindung, werden Daten über HT weitergereicht.

Die 16 Bit breiten HyperTransport-Links zwischen den Prozessoren bieten eine nominelle Bandbreite von $3.2 \cdot 10^9 Byte/s$ je Verbindung und Richtung. Eigene Messungen mit einer Read-Modify-Write-Schleife über einen linearen Speicherbereich von $8 \cdot 10^9$ Bytes in 32 Bit Zugriffen ergaben einen nutzbaren Speicherdurchsatz von $2.62 \cdot 10^9 Byte/s$ für den entfernten Speicher auf einem Dual-AMD Opteron 275HE (2.2GHz, DDR-400 RAM). Verglichen mit $3.11 \cdot 10^9 Byte/s$ bei derselben Messung für den eigenen Speicher ist das lediglich 15.8% weniger. Die Latenz für einen einzelnen Speicherzugriff liegt deutlich unter 1μs, sie wurde in [104] mit $110ns$ gemessen und in [196] mit $330ns + 130ns \cdot (n-1)$ für n konsekutive 64 Bit Zugriffe bestimmt, bei jedoch leicht unterschiedlichen Taktfrequenzen.

Damit hat diese Konfiguration klare Zeitvorteile verglichen mit einem Rechnernetzwerk: Schon bei unbelastetem Ethernet liegen dort die Latenzzeiten in einer Größenordnung von 100μs bei einer nominalen Bandbreite von maximal $10 \cdot 10^9 bit/s$. Zudem ist für jeden Datenaustausch zusätzlicher Rechenaufwand auf beiden Seiten einer Kommunikationsbeziehung notwendig. Hier beschränken sich die hardwarebedingten Kommunikationszeiten

auf die geringe Latenzzeit für den Zugriff auf den gemeinsamen Hauptspeicher, sodass in Kombination mit einer effizienten Softwareimplementierung umfangreicher Datenaustausch in Echtzeit möglich ist.

Die gewählte Opteron-Architektur ermöglicht Designs mit weiteren Prozessoren, die sich durch HyperTransport zu einem einheitlichen System zusammenfügen. Bei großen Systemen mit 8-32 Prozessoren verschlechtert zwar das Cache-Kohärenzprotokoll die Skalierbarkeit etwas, was jedoch durch zusätzliche Bausteine (*Snoopfilter*) behoben werden kann [114]. Das ausgewählte System kann auch als „Cluster-in-a-Box“ betrachtet werden, das zudem eine hohe Interkonnektivität aufweist. Im Vergleich zu einem richtigen Rechnercluster werden jedoch Infrastrukturkomponenten wie Festplattensysteme, Konsole oder Netzteile nur einmal benötigt, was im Fahrzeug Energie und Gewicht spart.

Die Zukunftsfähigkeit der gewählten Architektur bestätigen auch die von Intel seit Ende 2008 erhältlichen Prozessoren mit der Nehalem-Mikroarchitektur. Diese besitzen ebenfalls integrierte Speichercontroller, außerdem wird der Front-Side-Bus auch durch schnelle Punkt-zu-Punkt Verbindungen ersetzt, die dort *QuickPath Interconnect* (QPI) heißen.

Der Leistungsbedarf einer Rechnerkonfiguration mit zwei energiesparenden Zweikernprozessoren AMD Opteron 275HE (2.2GHz) liegt mit u. g. Festplatten und Schnittstellenkarten bei ca. 350 Watt aus dem 12V-Bordnetz über einen DC/DC-Wandler. Davon entfallen ca. 160 Watt auf die Hauptplatine Tyan Thunder K8WE2 (S2895A2NRF) mit Prozessoren und die 4 GB DDR-400 RAM [69].

4.3.2 Datenfluss bei der Aufzeichnung

In der AMD Opteron Architektur werden Peripheriegeräte und -busse über HyperTransport-Hubs und Tunnels „T“ angebunden. Mit den Peripheriebussen wie PCI, PCI-X und PCI-Express sind weitere Peripheriegeräte wie Firewire (IEEE 1394), CAN-Schnittstellen und sonstige E/A-Karten verbunden.

Im kognitiven Automobil werden typischerweise bis zu vier Kameras betrieben (vgl. Tabelle 6.5), davon zwei Weitwinkel-, eine Tele- und eine Innenraumkamera zur Beobachtung des Sicherheitsfahrers. Eine Kamera liefert ca. 9.7 MB/s. Wie Abb. 4.2 zu entnehmen ist, ist es in der gewählten Architektur möglich, diese Kameras auf zwei getrennte IEEE 1394-Schnittstellen zu verteilen, die wiederum über unterschiedliche HyperTransport-Links angebunden sind. Werden die E/A-Schnittstellenprozesse und deren Zielspeicher nun auch auf die angrenzenden Prozessorknoten „1“ und „2“ verteilt, läuft der Datentransfer parallel ab.

Für die anschließende Aufzeichnung ist es günstig, den Aufzeichnungsprozess auf dem Knoten „1“ auszuführen, der dem Festplattensystem „HD“ am nächsten ist. Dank der Aufzeichnungsmethode aus Anschnitt 5.7.1 bekommt der Aufzeichnungsprozess lediglich Ereignistupel mitgeteilt und holt sich die aufzuzeichnenden Daten eigenständig aus den Historienspeichern (siehe Kapitel 5.3.4). Dadurch entsteht für unbeteiligte Rechenkerne

kein Verarbeitungsaufwand. Wie in Kapitel 6.2.1 gemessen, können damit mühelos Datenraten von 40 MB/s bewältigt werden.

Als Datenträger zur Aufzeichnung werden „Raptor“-Platten von Western Digital verwendet. Diese zeichnen sich aus durch eine hohe Datenrate bei 10000 U/Min aus und sind dabei recht erschütterungsbeständig bis 3.0 g. Durch ihre „hot-plug“ fähige SATA-Schnittstelle und Einschubkanister können sie nach einer Aufzeichnungsfahrt leicht für eine Auswertung im Labor mitgenommen werden. Die Systemsoftware befindet sich auf einem robusten Solid-State-Datenträger. Sollte eine SATA-Platte unterwegs ausfallen, blockiert nur der Aufzeichnungsprozess und es fehlen zwar Aufzeichnungsdaten, das restliche System läuft jedoch ungestört weiter.

5 Realzeitfähige Softwarearchitektur

In diesem Kapitel werden zunächst die Anforderungen an eine Fahrzeugarchitektur für kognitive Automobile gezeigt. Daraufhin wird das Konzept der *Realzeitdatenbasis für kognitive Automobile* (*KogMo-RTDB*) vorgestellt. Dabei werden relevante Aspekte wie die Zeit- und Speicherverwaltung, Zugriffsmethoden und Schnittstellen sowie das blockierungsfreie Schreib-/Leseprotokoll vorgestellt. Anschließend wird die Realzeitfähigkeit des gesamten Systems analysiert und die entwickelten Methoden zur störungsfreien Kooperation verschieden harter Echtzeitprozesse präsentiert. Zum Abschluss wird das Vorgehen bei der zeitgenauen Datenaufzeichnung erläutert.

5.1 Anforderungen

Wesentliche Anforderungen an die notwendige Softwarearchitektur entstanden aus dem zugrundeliegenden Forschungsprojekt: Der Sonderforschungsbereich/Transregio 28 *Kognitive Automobile* (vgl. Kapitel 3.6.1) hat unter anderem das Ziel, seine Forschungsergebnisse anhand von realen Automobilen zu demonstrieren [195]. Die Ergebnisse entstehen in 12 interdisziplinären Teilprojekten an verschiedenen Instituten der beteiligten Universitäten und müssen für die Demonstration in *ein* Fahrzeug zu *einer funktionierenden Einheit* integriert werden. Daraus folgt für die Architektur als wichtige Aufgabe, die Integrierbarkeit verteilt entstehender Komponenten sicherzustellen.

An Kognitiven Automobilen forschen Wissenschaftler verschiedener Fachrichtungen: Ingenieure des Maschinenbaus, der Elektro- und Regelungstechnik, Informatiker, Experten für Bild- und Wissensverarbeitung. Jedes Institut besitzt üblicherweise seine eigenen etablierten Methoden, Werkzeuge, Software und Entwicklungsumgebungen. Am deutlichsten wird dies sichtbar durch die Verwendung unterschiedlicher Programmiersprachen, wie C, C++, Java, Ada und andere. Eine fruchtbare Zusammenarbeit benötigt gemeinsame Standards, Schnittstellen und Werkzeuge. Um eine erfolgreiche Kooperation auf der Softwareebene sicherstellen und die notwendigen Brücken zu bauen, muss eine Softwarearchitektur von allen Beteiligten akzeptiert und eingesetzt werden.

Die in Kapitel 3.6.3 vorgestellten kognitiven Ebenen stellen teils gegensätzliche Anforderungen an die Architektur:

- Auf den hardwarenahen Regelungsebenen werden eine harte Echtzeit, hohe Regelfrequenzen und niedrige Latenzzeiten gefordert. Dabei bleibt selten Zeit, nützliche Daten für andere Funktionen zu sammeln und entsprechend aufzubereiten.

- Die Funktionen der Wissensebene benötigen Zugriff auf möglichst alle im Fahrzeug verfügbaren Informationen, möglichst zeitlich gepuffert, da auf der Wissensebene in längeren Zyklen gerechnet wird. Dadurch dürfen sie jedoch in keinem Fall andere Funktionen beeinträchtigen, sonst würden beispielsweise unterlagerte Regelkreise instabil werden.
- Leistungsfähige Sensoren wie Kameras oder Mehrstrahl-Laserscanner erzeugen mit mindestens $3 \ldots 8 \cdot 10^6$ Byte/s große Datenmengen, deren Handhabung rechenintensiv ist. Daher besteht das Verlangen, diese Daten unmittelbar auszuwerten und nur die um Größenordnungen reduzierten Ergebnisse an andere Funktionen weiter zu reichen. Damit wird aber eine alternative Rohdatenauswertung durch komplementäre Funktionen, wie z.B. eine Sensor-Datenfusion auf Pixel-Ebene, unmöglich gemacht.

Gerade für einen Forschungsbereich ist es wichtig, dass eine Softwarearchitektur viele Freiheitsgrade lässt und die Entwicklungen nicht auf einen bestimmten Ansatz festlegt oder in eine einzige, möglicherweise suboptimale, Richtung lenkt. Die im Folgenden präsentierte Architektur versucht alle Anforderungen zu befriedigen, aber gleichzeitig die gemeinsame Integration zu ermöglichen. Dafür werden wenige grundsätzliche Regeln für Schnittstellen bei der Entwicklung von Funktionen aufgestellt.

5.2 Integrationsframework

Wie in Kapitel 2 gezeigt, existieren eine Reihe von Lösungen für intelligente Fahrzeuge. Diese bieten jedoch oftmals Schnittstellen für eine Teilmenge der in Kapitel 3.6.3 genannten Wahrnehmungsebenen und spezielle Anwendungsfälle. Dabei setzen sie für das jeweilige Forschungsgebiet übliche Softwareumgebungen und -bibliotheken voraus, die Forscher anderer Gebiete eher als störend empfinden könnten.

Daher wird als günstigste Lösung eine leichtgewichtige generische Architektur vorgeschlagen, die einheitliche Schnittstellen zur Integration aller Softwaremodule im Fahrzeug bereitstellt [75, 77]. Dies kann als äußeres Framework betrachtet werden, da Softwaremodule weiterhin ein inneres Framework enthalten können, das durch bestehende Softwarestrukturen eines Forschungsinstituts vorgegeben wird.

5.2.1 Datengetriebener Ansatz

Es existieren fortgeschrittene Designparadigmen, wie beispielsweise das objektorientierte Design. Dennoch ist es oft unmöglich, sich in einem Projekt, an dem beispielsweise verschiedene Ingenieursdomänen involviert sind, sich auf ein gemeinsames Designparadigma zu einigen. Beteiligte Forscher müssten, um ein neues Paradigma anzunehmen, die damit verbundenen Programmiertechniken übernehmen, hätten Anlaufverluste und müssten teils etablierte Lösungen umstellen.

Aufgrund dieser Situation wird der datengetriebene Ansatz als der kleinste gemeinsame Nenner für gemeinsame Schnittstellen untersucht. Das gesamte Softwaresystem besteht aus Modulen, die beliebige Daten austauschen. Der Vorteil ist, dass sich darin mit einfachsten Mitteln spezifizieren lässt, welche Informationen ein Softwaremodul am Eingang benötigt, welche es liefert und innerhalb welcher Zeit das zu geschehen hat. Die Verifikation, ob ein Modul seine zeitlichen Spezifikationen erfüllt, ist wie in Kapitel 6.3 gezeigt, leicht durchführbar.

5.2.2 Datenfluss

Für ein situationsgerechtes Handeln benötigen die Module eines kognitiven Automobils zahlreiche Eingangsgrößen. Der sich ergebende Datenfluss wurde bereits in Kapitel 3.6.3 skizziert. Wie in Abb. 3.3 zu erkennen ist er schon für eine einfache Aufgabe wie „Spur fahren mit Abstand halten" ein Graph mit zahlreichen Zyklen. Ein Beispiel dafür ist der Übergang von der systemdynamischen zur Wissensebene: Hier werden auf der Wahrnehmungsseite die quantitativen Daten für die Situationsanalyse klassifiziert und in qualitative Daten transformiert. Nach beispielsweise der Entscheidung, dem vorausfahrenden Fahrzeug zu folgen, benötigt der Längsregler aus Abb. 3.1 die exakte Geschwindigkeit des Eigen- und die geschätzte des Fremdfahrzeugs. So werden auf der Handlungsseite die ursprünglichen quantitativen Daten wieder benötigt. Eine Lösung wäre, alle Daten von Modul zu Modul und Ebene und Ebene weiter zureichen. Dies würde aber bei anwachsenden Datenmengen sehr rechenaufwendig. Ein Hinzufügen neuer Eingangsdaten würde zudem die Modifikation aller Softwaremodule entlang des Verarbeitungsweges erfordern. Ein effizienterer Weg wäre, die qualitativen Daten mit ihren quantitativen Quelldaten zu verknüpfen, ohne sie explizit transportieren zu müssen.

Um komplexe Entscheidungen sicher treffen zu können, sollte ein kognitives Fahrzeug unabhängige Wahrnehmungs- und Situationsanalysemodule besitzen. Damit Entscheidungsmodule auf deren Ergebnisse zurückgreifen können, benötigen sie entsprechende Schnittstellen dorthin. Der Kommunikationsaufwand zwischen n Wahrnehmungs- und m Entscheidungsmodulen liegt damit in der Größenordnung $n \cdot m$. Möchten alle Module untereinander Daten austauschen, wächst die Anzahl der Kommunikationsbeziehungen auf $(m+n-1)!$. Muss für jeden Informationstyp eine eigene Schnittstelle eingesetzt werden, multipliziert sich die Menge an Kommunikationsverbindungen.

5.2.3 Datenbanken zur Informationsdistribution

In der Literatur finden sich, wie bereits in Kapitel 2.1 vorgestellt, vielversprechende Ansätze, Datenbanken für die Informationsverteilung in Fahrzeugen einzusetzen: In den in [41] zusammengefassten Arbeiten wurde eine Dynamische Objektdatenbasis (DOB) eingesetzt, in der vor allem Schätzwerte dynamischer Objektzustände gespeichert wurden. Sensorrohdaten wurden nicht gespeichert, da erstens kein Bedarf bestand und zweitens die Verarbeitungsleistung der Rechner damals nicht ausreichte. Im Projekt COMET [153]

wurde der Einsatz von Datenbanken in Fahrzeugen unter dem Aspekt des Software-Engineerings mit positiven Ergebnissen untersucht.

Der Einsatz einer Datenbank-ähnlichen Architektur in kognitiven Fahrzeugen hat zahlreiche Vorteile:

Überschaubare Anzahl notwendiger Schnittstellen: Die Verwendung einer Datenbank macht es erforderlich, jede zu speichernde Information zuvor in einheitlicher Weise zu spezifizieren. Eine bekannte Sprache dafür ist beispielsweise SQL [36] für relationale Datenbanken. Unter Schnittstellengesichtspunkten benötigen m Module bei einem konsequenten Datenaustausch nur über die Datenbank, lediglich identische m Datenbankverbindungen. Für n verschiedene Informationen werden zusätzlich n Spezifikationen benötigt, die aber die Anzahl notwendiger Datenbankverbindungen nicht erhöhen.

Transparente Kommunikation und Diagnoseeigenschaften: Bei der Verteilung einer Information an mehrere Konsumenten ist der Einsatz einer Datenbank unmittelbar einsichtig. Aber auch bei der direkten Kommunikation zweier Module macht der Weg über eine Datenbank Sinn: Wenn ein Modul Daten in die Datenbank schreibt, auf die ein zweites Modul wartet und sie wieder ausliest, kann dieser Vorgang von einem dritten Modul beobachtet werden. Die sich ergebende Transparenz in der Kommunikation vereinfacht die Diagnose signifikant. Darüber hinaus können die an dieser Schnittstelle abgreifbaren Daten von weiteren Modulen genutzt werden.

Einheitliche Situationsrepräsentation: Eine Datenbank ist ein zentraler Informationsspeicher. Füllt man ihn mit allen verfügbaren Informationen über die Umwelt, wie z.B. aktuellen Messungen und Schätzungen der Bewegungen erkannter Umweltobjekte, sowie der daraus erfolgten Situationsbewertung, trägt dies zu einer ganzheitlichen Sicht der Umgebung bei. Auch das Fehlen, sowie das Alter (sofern verfügbar) von Informationen kann dabei berücksichtigt werden.

Diese einheitliche Situationsrepräsentation nutzt dann unmittelbar den Modulen der Wissensverarbeitung: In [118] wird die gefundene Objektkonstellation mittels einer Inferenzmaschine mit gelernten Situationen verglichen. Im Ansatz von [158] wird ein Fähigkeitennetzwerk mit Experten verwendet, die eine unscharfe Suche nach bekannten Situationsmustern durchführen. Zum Einsatz automatischer Lernverfahren ist eine einheitliche Zusammenstellung aller Informationen ebenfalls hilfreich.

Technische Voraussetzungen Eine zentrale Komponente wie eine Datenbank trägt die Gefahr in sich, ein möglicher Engpass zu werden. Daher ist es essentiell, verfügbare Transaktionen voneinander zu entkoppeln, sodass eine Aktion eine andere nicht unbeabsichtigt blockieren oder verzögern kann. Andernfalls wird die Verwendung der Datenbank für die Entwickler unattraktiv, sie erfinden womöglich alternative Kommunikationswege. Des weiteren müssen Daten bei einer Abfrage so schnell verfügbar sein, dass es nicht zum Anlegen privater Datenbankkopien zur eigenen Zugriffsbeschleunigung und damit zum Verlust der Einheitlichkeit kommt.

Für einen Einsatz innerhalb echtzeitfähiger Funktionen muss die gesamte Datenbank echtzeitfähig sein. Das erfordert entweder vollständige Blockierungsfreiheit oder durchgängigen Einsatz eines Prioritätsmanagementprotokolls, wie z.B. dem *Priority-Inheritance-Protocol* (*PIP*), und deterministischen Blockierungszeiten. Einen Überblick über Echtzeitdatenbanken gibt [13, 103].

5.3 Die Realzeitdatenbasis für kognitive Automobile (KogMo-RTDB)

Aufgrund der vorangegangenen Überlegungen wurde die *Realzeitdatenbasis für kognitive Automobile* [73, 74], abgekürzt *KogMo-RTDB*, entwickelt. Sie ist das zentrale Element der Softwarearchitektur für kognitive Fahrzeuge im SFB/TR 28 [71, 102, 220, 101, 206] und erfüllt die erwähnten Anforderungen. Sie ist kostenfrei als *Open-Source* erhältlich [67] und findet bereits in weiteren Forschungsprojekten Anwendung, beispielsweise im Demonstrator einer kognitiven Fabrik innerhalb des Exzellenzclusters *Cognition for Technical Systems* (*CoTeSys*) [217, 122, 11, 166].

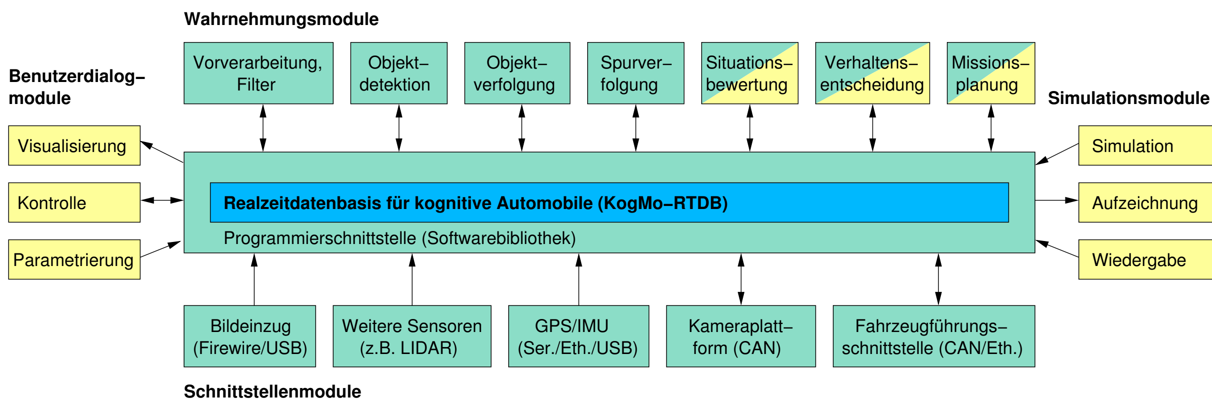


Abbildung 5.1: Softwarearchitektur der KogMo-RTDB

Abb. 5.1 zeigt die Struktur der Fahrzeugsoftware mit der KogMo-RTDB in der Mitte als gemeinsames Integrationsframework. Alle relevanten im Fahrzeug anfallenden Daten werden über die in Kapitel 5.3.2 ausgeführten Methoden gespeichert und können dann beliebig wieder abgerufen werden. Die einzige Kommunikationsverbindung zweier Module besteht über die KogMo-RTDB, sonstige Verbindungen sollten vermieden werden. Dies garantiert die bestmöglichste Transparenz der Kommunikation, da sämtliche Änderungen in der Datenbank beobachtbar sind. Mit den in Kapitel 5.7 beschriebenen Methoden können alle Daten einschließlich ihres zeitlichen Verlaufs auf Massenspeicher mitprotokolliert und anschließend in eine laufende KogMo-RTDB in Echtzeit wieder eingespielt werden.

Der untere Teil von Abb. 5.1 zeigt eine Auswahl an *Ein/Ausgabe-Schnittstellenmodulen* (*Input/Output-Interfacemoduls*) zur Peripherie. Sie haben zum einen die Aufgabe, die Rohdaten von Sensoren möglichst unverändert abzulegen und zyklisch zu aktualisieren.

Zum anderen müssen sie Kommandos und Regelvorgaben an die entsprechenden Aktoren weiterreichen. Dabei werden in der Regel hardware- und sensorspezifische Programmierschnittstellen benutzt, wie z.B. CAN-Bus-Treiber für die Fahrzeuganbindung oder IEEE1394-Firewire- und USB-Treiber für den Kamerabildeinzug. Es wird empfohlen, die Sensordaten möglichst ohne Vorverarbeitung in der KogMo-RTDB zu publizieren, da sich so alternative (Vorverarbeitungs-)Algorithmen erproben lassen. Sinnvollerweise sollte ein Verarbeitungsalgorithmus besser durch ein weiteres Modul realisiert werden, das seine Daten aus der KogMo-RTDB bezieht und das seine Ergebnisse wieder dort ablegt. Im Falle einer fehlerhaften Implementierung des Moduls lassen sich zumindest die mitprotokollierten Rohdaten einer KogMo-RTDB-Aufzeichnung im Labor zur Analyse und erneuten Verarbeitung nutzen.

Die gezeigten Wahrnehmungsmodule der Bild- und Wissensverarbeitung beziehen alle ihre Eingabedaten aus der KogMo-RTDB. Dadurch ist es einfach, diese Module im Labor anhand von aufgezeichneten oder simulierten Daten zu testen, wie es in Kapitel 5.7.4 demonstriert wird.

Von Vorteil ist es, die graphische Bedienoberfläche (*Graphical User Interface, GUI*) von den verarbeitenden Modulen abzutrennen und diese über die KogMo-RTDB kommunizieren zu lassen. So lässt sich eine KogMo-RTDB-Aufzeichnung mit identischen GUI-Modulen visualisieren. Im autonomen Betrieb kann Rechenzeit gespart werden, indem unnötige GUI-Module abgeschaltet werden. Zudem sind die Schnittstellen zum Benutzer gewöhnlich nicht hart echtzeitfähig und würden einem Verarbeitungsmodul, das auf Eingaben aus einer GUI wartet, einen harten Echtzeitbetrieb unmöglich machen.

5.3.1 RTDB-Objekte

Die kleinste von der RTDB verwaltete Informationseinheit ist ein RTDB-Objekt, ein definierter Container, der eine Menge zusammengehöriger Daten aufnimmt. Daraus kann bei Bedarf ein C++-Objekt abgeleitet werden. Beispiele für RTDB-Objekte sind unter anderem:

Rohdaten von Sensoren: Position aus einer satellitengestützten Positionsbestimmung (*Global Positioning System, GPS*), Daten aus einem System zur Eigenbewegungsmessung (*Inertial Measurement Unit, IMU*), Videobilder aus Graustufen-, Farb- und Infrarotkameras, Kameraparameter (Kalibrierung, Verschlusszeiten, ...), Entfernungsmesswerte zu Objekten der Fahrzeugumgebung mittels *LIDAR* (*Light Detection and Ranging*) und *RADAR* (*Radio Detection and Ranging*), Winkel der Kamerastellungen einer aktiven Kameraplattform, Fahrzeugdaten (Geschwindigkeit, Lenkwinkel, Raddrehzahlen), ...

Ergebnisse der Umfeldwahrnehmung: Fahrspurschätzwerte, verfolgte Fahrzeuge, Hindernisse, Belegungskarten der Umgebung, Visualisierungselemente, Flussvektoren im Videobild, ...

Tabelle 5.1: Struktur eines RTDB-Objekts

Statische Objektmetadaten ($\mathcal{M}$)	
Vom Anwender spezifiziert	$name$, TID, OID_{parent}, $T_{history}$, $t_{cycle,max}$, $t_{cycle,min}$, $n_{bytes,max}$, $flags$
Von der KogMo-RTDB vorgegeben	$t_{created}$, $PID_{created}$, $t_{deleted}$, $PID_{deleted}$, OID
Dynamische Objektnutzdaten ($\mathcal{D}$)	
Vom Anwender spezifiziert	$DATA_{user}$, n_{bytes}, t_{data}
Von der KogMo-RTDB vorgegeben	$t_{committed}$, $PID_{committed}$

Ergebnisse höherer Wahrnehmungsebenen: Klassifizierte Objekte, bewertetes Fahrzeugumfeld, analysierte Situation, eigene Verhaltensentscheidung, ...

Steuergrößen: Soll-Beschleunigung, Lenkrate, Fahrkorridor, Trajektorie, Blickrichtungswunsch, Stellwerte, ...

Sonstige Daten: ASCII-Texte, XML-Daten, Konfigurationsdateien, beliebige serialisierbare Daten, ...

Neue Datenobjekte können frei definiert und somit die obige Liste beliebig erweitert werden. Ein RTDB-Objekt besteht grundsätzlich aus folgenden, in Tabelle 5.1 zusammengefassten, Elementen:

Statischer Metadatenblock $\mathcal{M}$: Dieser enthält eine vorgegebene Menge statischer Informationen über ein RTDB-Objekt und wird einmalig beim Anlegen eines Objekts in der KogMo-RTDB erzeugt. Jedes Objekt bekommt automatisch eine Identifikationsnummer (OID, Object-ID) zugewiesen. Diese ist für die Laufzeit der KogMo-RTDB eindeutig und wird auch nach dem Löschen eines Objekts nicht neu vergeben.

Des weiteren hat ein Objekt einen Namen ($name$) und eine Typenkennung (TID), die den Inhalt des Nutzdatenblocks festlegt. Die Angabe der Objekt-ID eines Vaterobjekts (OID_{parent}) ermöglicht es, einen Szenenbaum nach [41] in der KogMo-RTDB aufzubauen. $flags$ enthält Konfigurationsparameter und Zugriffsrechte des Objekts, auf die in den Kapiteln 5.3.5 und 5.5.7 genauer eingegangen wird.

PID bezeichnet eine interne Identifikationsnummer, die jeder mit der KogMo-RTDB verbundene Prozess automatisch zugewiesen bekommt, t einen Zeitpunkt: Der Prozess $PID_{created}$ hat das Objekt zur Zeit $t_{created}$ angelegt. Wenn das Objekt gelöscht wird, wird dies in $PID_{deleted}$ und $t_{deleted}$ vermerkt, bis es endgültig aus der KogMo-RTDB entfernt wird.

Dynamischer Nutzdatenblock $\mathcal{D}$: Dieser Datenblock enthält die Nutzdaten ($DATA_{user}$) eines RTDB-Objekts und wird im Betrieb von den Softwaremodulen laufend ak-

tualisiert. Vorgeschrieben ist nur ein kurzer Satz von Basisdaten. Diese umfassen die Kennung des schreibenden Prozesses ($PID_{committed}$) mit dem Schreibzeitpunkt ($t_{committed}$). Der Datengültigkeitszeitpunkt (t_{data}) wird vom Benutzer spezifiziert und kann genutzt werden, um Nutzdaten eines bestimmten Zeitpunkts zu selektieren oder um nach (5.4) das Datenalter zu berechnen. Für die aktuelle Datengröße (n_{bytes}) gilt stets:

$$n_{bytes} \leq n_{bytes,max} \tag{5.1}$$

Die Struktur der Nutzdaten kann frei definiert werden, sie werden von der KogMo-RTDB als Container mit einer Menge zusammengehöriger Daten betrachtet. Die Typidentifikation (TID) eines Objekts legt fest, welcher Art die Daten sind. Um deren Einheitlichkeit zu garantieren, muss die Liste aller TIDs, zusammen mit den Nutzdatendefinitionen an zentraler Stelle, beispielsweise einem Quellcode-Repository wie Subversion [160], für alle Nutzer verfügbar und erweiterbar sein. Zur Nachvollziehbarkeit wird die Revisionsnummer der RTDB-Objektdefinitionen in jeder RTDB-Aufzeichnung mitprotokolliert.

Auf die verbleibenden Parameter, insbesondere die Handhabung von Zeitstempeln, wird in Kapitel 5.3.4 detaillierter eingegangen.

5.3.2 Zugriffsmethoden

Da bekannte textbasierte Datenbankschnittstellen wie SQL [36] für jede Anfrage eine rechenintensive lexikalische Analyse erfordern, wurden neue Methoden für einen schnellen effizienten Zugriff entwickelt. Deren Ausführungszeiten sind, wie in Kapitel 6.1 präsentiert, echtzeitfähig und selbst für Regelschleifen im kHz-Bereich geeignet.

Die verfügbaren Methoden können in folgende Gruppen eingeteilt werden:

Verbindungsmanagement:
Damit wird die Verbindung eines Prozesses zu einer laufenden KogMo-RTDB auf- und wieder abgebaut. Bei Aufbau wird jeder Prozess als RTDB-Objekt in die Datenbank abgebildet, sodass für alle Module ersichtlich ist, welche Prozesse laufen. Beim Verbindungsaufbau wird in der Realzeitkonfiguration der Prozess zusätzlich auf den Wechsel in den harten Echtzeitbetrieb vorbereitet, wie in Kapitel 5.5.8 vorgeführt wird. Für jeden Prozess muss eine Zykluszeit $t_{process}$ spezifiziert werden, die jedoch nicht zwingend eingehalten werden muss. Sie wird als $t_{cycle,min}$ in das Prozessobjekt eingetragen und dient lediglich zur Dimensionierung dessen Historie.

Metadatenverwaltung:
Diese Gruppe beinhaltet unter anderem folgende Methoden:

- Anlegen eines neues Objekts
- Löschen eines existierenden Objekts, sofern erlaubt
- Abfrage der Metadaten eines bestimmten Objekts

- Suche nach Objekten (OIDs), dessen Metadaten in ein gegebenes Suchmuster ($name$,TID,OID_{parent},t) passen
- Blockierende Suche nach einem Objekt mit Aufwecken des Prozesses, sobald das gewünschte Objekt verfügbar wird
- Warten auf Änderungen am Bestand von Objekten und Rückgabe einer Liste neu hinzugekommener bzw. gelöschter Objekte

Nutzdatenzugriff:
Ist die Objekt-ID (OID) durch eine Metadatensuche oder nach dem Anlegen bekannt, kann auf die Nutzdaten des RTDB-Objekts zugegriffen werden:

- Datenaktualisierung: Damit können die Nutzdaten eines RTDB-Objekts aktualisiert werden. Dies geschieht blockierungsfrei, wenn nur der Besitzer eines Objektes Schreibrechte hat. Andernfalls wird der Schreibvorgang durch Mutexes abgesichert. Die geänderten Daten werden für jedes Objekt in Ringspeichern organisiert, sodass diese für eine gegebene Zeitspanne erhalten bleiben. Kapitel 5.3.4 beschreibt die zugrundeliegenden Konzepte dazu, Kapitel 5.5.7 erläutert die Zeitbeschränkung bei möglichen Blockierungen.
- Datenabfrage: Sie erlaubt Prozessen, Daten von RTDB-Objekten zu lesen. Unter Angabe eines Zeitpunkts kann in den Ringspeichern navigiert werden und Datenblöcke älter oder jünger als $t_{committed}$ oder t_{data} abgerufen werden.
- Warten auf Aktualisierung: Dieser Methode wird als Parameter der letzte bekannte Aktualisierungszeitpunkt $t_{committed,known}$ mitgegeben. Die Anfrage `readdata_waitnext(`$t_{committed,known}$`)` blockiert dann, bis neuere Daten verfügbar sind. Sind bereits neuere Daten verfügbar, werden die aktuellsten Daten sofort zurückgegeben (vgl. Abschnitt 5.3.6).

Die KogMo-RTDB wird von vielen Wissenschaftlern des SFB/TR 28, einschließlich deren studentischen und wissenschaftlichen Hilfskräften, eingesetzt. Für eine breite Akzeptanz ist es wichtig, überschaubare und leicht zu lernende Programmierschnittstellen zur Verfügung zu stellen. Daher wurden obige intuitive, datenbankähnliche Zugriffsmethoden gewählt. Sie werden durch Einbinden einer dynamischen Softwarebibliothek (*shared library*) in der eigenen Software genutzt.

5.3.3 Zeitverwaltung

Das kognitive Automobil ist mit einer Vielzahl von Sensoren ausgestattet, die jeweils eine individuelle Zykluszeit besitzen. Zur Erleichterung der Datenfusion wäre es wünschenswert, alle Sensoren zu synchronisieren. Bei gleichartigen Sensoren wie Videokameras ist dies problemlos möglich. Für andere ist es aufwendig, gerade wenn Sensoren von verschiedenen Instituten beigesteuert werden. Aus physikalischen Gründen kann es sogar unmöglich sein: Ein rotierender Laserscanner liefert kontinuierlich Daten, der Startzeitpunkt einer Umdrehung lässt sich nicht schlagartig verschieben und frei synchronisieren,

da er sich aufgrund seiner Masse nicht beliebig schnell beschleunigen lässt. Die Bestimmung eines taktgebenden Sensors ist ebenfalls problematisch, da es Module gibt, für die nur wenige Sensoren interessant sind. Beispielsweise sind für einen Fremdfahrzeug- oder Fahrspurtracker wie [149] nur das Videobild und die Fahrzeugodometrie relevant, ein synchronisationsbedingtes Warten auf einen weiteren Sensor würde dabei nur die Stabilität z. B. einer Spurhaltung beeinträchtigen.

Im Projekt *ASKOF* (*Architektur und Schnittstellen für kognitive Funktionen in Fahrzeugen*) [59], einem Vorprojekt zum SFB/TF 28, wurde in einer frühen Version der KogMo-RTDB die Eignung diskreter Zeitpunkte, sog. *Ticks*, für die zeitliche Synchronisation untersucht. Für eine konkrete Anwendung, wie der videobasierten Spurhaltung mit einem dominanten Sensor, ist dies akzeptabel. Für kognitive Automobile, die beispielsweise auf der DARPA Urban Challenge [102, 101] mit einem rotierenden Laserscanner ausgestattet waren, ist dies zu unflexibel.

In der präsentierten Architektur wird daher ein anderer Ansatz gewählt: Es werden die sensoreigenen Zykluszeiten zugelassen und die Daten eines jeden Sensors von einem zugehörigen E/A-Schnittstellenmodul nach ihrem Eintreffen unverzüglich in die KogMo-RTDB geschrieben. Die gespeicherten Daten werden dann von der KogMo-RTDB mit genauen Zeitstempeln versehen und für ein wählbares Zeitfenster (Historie) vorgehalten. So stehen die Sensordaten allen Wahrnehmungsmodulen unmittelbar zur Verfügung, und können mittels geeigneter Interpolation auf einen gemeinsamen Zeitpunkt gebracht werden.

Zeitauflösung

Für eine einheitliche Spezifikation eines absoluten Zeitpunkts (*Zeitstempel* t, *Timestamp*) wurden verschiedene Kombinationen aus maximaler Auflösung und Speicherbedarf in Bit (ohne Vorzeichen) untersucht:

Auflösung	**Speicherbedarf**	**Darstellbare Zeitspanne**
Millisekunden	32 Bit	< 50 Tage
Mikrosekunden	32 Bit	< 2 Stunden
Nanosekunden	32 Bit	< 5 Sekunden
Nanosekunden	64 Bit	< 585 Jahre

Eine Millisekundenauflösung ist dabei zu ungenau für schnelle Regelkreise und eine Mikrosekundenauflösung erzeugt innerhalb einer längeren Fahrt mehrmals einen Überlauf bei 32 Bit. Daher erscheint der doppelte Speicherbedarf gerechtfertigt, 64 Bit Zeitstempel zu verwenden. Damit lässt sich eine Nanosekundenauflösung ohne absehbare Überläufe realisieren, bei vorzeichenbehafteten (*signed*) Werten erreicht man immerhin über 292 Jahre. In der KogMo-RTDB werden daher absolute Zeitstempel mit 64 Bit verwendet. Nullzeitpunkt ist der 1.1.1970 UTC (die sog. „UNIX Epoche“, ohne Schaltsekunden). Sie sind nutzbar bis zum Jahr 2262. Ein Überwachungsprozess wie in Kapitel 6.3.3 kann einen

drohenden Überlauf des Zeitstempels sowie der Objektidentifikationsnummer frühzeitig erkennen und das System sicher anhalten.

Ein weiterer Vorteil dieser Zeitstempel zeigt sich bei der Aufzeichnung. So kann anhand der absoluten Zeit und der GPS-Position beispielsweise die Jahreszeit berechnet werden. In Kapitel 6.3.2 wird gezeigt, wie unter Verwendung der automatisch erzeugten Zeitstempel der Rechenzeitbedarf von Softwaremodulen ermittelt werden kann.

Uhrensynchronisation

Die Genauigkeit der Zeitstempel wird bestimmt durch die im System verwendeten Uhren. In der betrachteten Systemarchitektur werden dafür das Register *Timestamp-Counter* (*TSC*) verwendet, das jeder moderne x86-kompatible Prozessor seit der Pentium-Serie besitzt. Der TSC wird bei jedem Taktimpuls des Prozessors inkrementiert, bei einem Prozessortakt von $f_{CPU} = 2 \cdot 10^9$ Hz geschieht dies alle $\frac{1}{f_{CPU}} = 0.5$ ns, ein Überlauf dieses 64 Bit-Registers erfolgt so nach 293 Jahren.

Die Synchronisation der TSCs aller Prozessoren eines Multiprozessorrechners untereinander muss einmalig initial erfolgen, zur Laufzeit werden alle Prozessoren mit dem gleichen Systemtakt versorgt. Im Kern des Linux-Betriebssystems [126] geschieht das bei Systemstart [68]. Die dabei erreichbare Genauigkeit liegt im Bereich der Zeit, die eine Cacheline benötigt, um von einem Prozessor zum anderen weitergegeben zu werden. Bei einem Opteron-System dauert der Zugriff auf einen entfernten Speicher nach [104] unter 110 ns und wurde in [196] auf 330 ns + 130 ns $\cdot$ $(n - 1)$ für n aufeinanderfolgende Zugriffe beschränkt. Die Genauigkeit liegt damit deutlich unter 1 μs. Im Betrieb ist darauf zu achten, dass keine Energiespartechniken aktiviert werden, die möglicherweise die Frequenzteiler der Prozessoren modifizieren.

Eine externe Synchronisation zwischen Fahrzeugen oder mehreren Rechnersystemen stand nicht im Fokus dieser Arbeit, wurde aber exemplarisch mit einem GPS-Empfänger als externer Taktquelle erprobt. Die hier erreichbare Genauigkeit wird zusätzlich durch die Interruptlatenzzeit vermindert, die im Falle des in Kapitel 4.3.1 beschriebenen Opteron-Systems mit 6.2 μs-26.4 μs gemessen wurde.

Einsatz von Zeitstempeln

Jedes Objekt wird bei seiner Aktualisierung automatisch mit dem aktuellen Zeitstempel $t_{committed}$ (vgl. Tabelle 5.1) versehen (*Committed-Timestamp*). Dieser markiert den Zeitpunkt der Übergabe der Daten an die KogMo-RTDB. Zusätzlich müssen Softwaremodule den Gültigkeitszeitpunkt t_{data} der geschriebenen Daten mit angeben (*Data-Timestamp*). Dies ist bei Sensorwerten der Entstehungszeitpunkt, z.B. der mittlere Belichtungszeitpunkt bei Videobildern. Bei der Weiterverarbeitung der Daten muss t_{data} in die Ergebnisobjekte, wie z.B. ein Objekt mit der erkannten Fahrspur, übernommen werden.

Durch die konsequente Nutzung von Zeitstempeln ist eine konsistente Sicht auf alle Daten jederzeit gewährleistet: Sämtliche Abfragen von Objekten werden ebenfalls mit einem Zeitpunkt parametrisiert. Der Einsatz interner Ringspeicher stellt sicher, dass der anfragende Prozess genau die Daten bekommt, die zu einem gegebenen Zeitpunkt aktuell sind bzw. waren. Dies liefert langsameren Prozessen ein konsistentes Abbild aller Daten und ist vor allem für die Entkopplung von Wahrnehmungsprozessen unterschiedlicher zeitlicher Auflösung relevant. Anhand von $t_{committed}$ und t_{data} lässt sich sogar in der Historie der Daten navigieren, wie in Abschnitt 6.4.1 gezeigt wird. In der Fusion kann die verfügbare Historie der Daten zur Interpolation verwendet werden, um beispielsweise fehlende Sensorwerte zu berechnen.

5.3.4 Historienkonzept

Um die blockierungsfreie Zusammenarbeit von Softwaremodulen mit unterschiedlichen zeitlichen Anforderungen zu ermöglichen, setzt die KogMo-RTDB ein System von internen Ringspeichern ein. Dabei wird unterschieden zwischen den Nutzdaten $\mathcal{D}[i]$, die sich zur Lebenszeit eines RTDB-Objekts dynamisch ändern, und Metadaten $\mathcal{M}$, die statisch sind. Nur für die Nutzdaten ist es daher notwendig, diese in verschiedenen Versionen vorzuhalten. Abb. 5.2 zeigt die Struktur eines für jedes einzelne Objekt verwalteten Ringspeichers. Die Verwaltung geschieht aus der Sicht von darauf aufsetzenden Softwaremodulen transparent. Bei jeder Aktualisierung der Nutzdaten wird ein neuer Speicherplatz im Ringspeicher verwendet. So stehen vergangene Versionen der Daten weiterhin zur Verfügung.

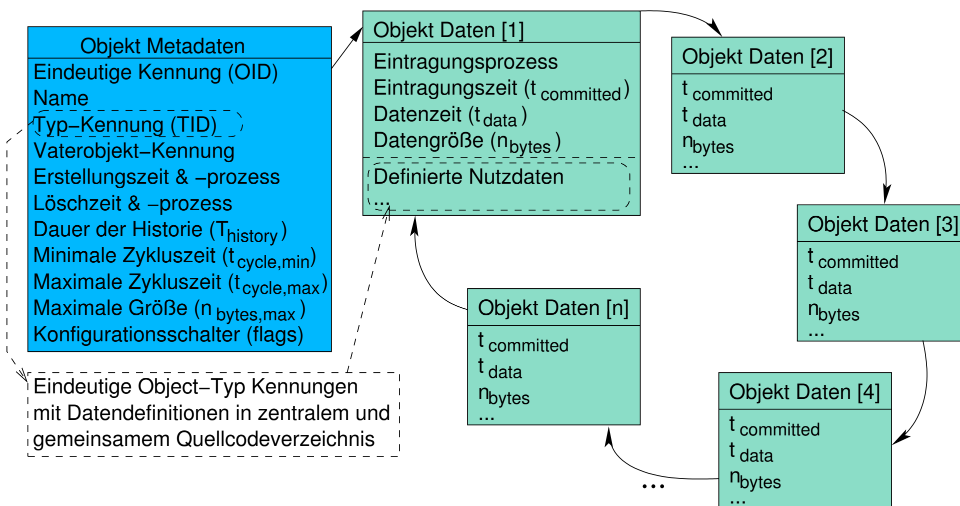


Abbildung 5.2: Historienkonzept der KogMo-RTDB

Um das System deterministisch zu machen, muss der von jedem Objekt einnehmbare Speicherplatz beschränkt werden und eine sinnvolle Anzahl alter Versionen gefunden werden. Um die erforderliche Größe des für ein RTDB-Objekt benötigten Ringspeichers zu berechnen, sind daher bei Erstellung eines neuen Objekts folgende Parameter in den Metadaten $\mathcal{M}$ aus Tabelle 5.1 zu spezifizieren:

- $T_{history}$ gibt an, für welchen Zeitraum vergangene Versionen der Nutzdaten vorgehalten werden sollen.
- $t_{cycle,min}$ ist das kürzeste erlaubte Aktualisierungsintervall. Standardmäßig wird dafür der Wert $t_{process}$ verwendet, der die Zykluszeit des erstellenden Prozesses darstellt und beim Verbindungsaufbau mit der KogMo-RTDB spezifiziert werden muss.
- $t_{cycle,max}$ gibt das längste erlaubte Intervall zwischen zwei Aktualisierungen an.

Die Anzahl der notwendigen Speicherplätze (*slots*) berechnet sich mit diesen Werten zu:

$$n_{slots} = \left\lceil \frac{T_{history}}{t_{cycle,min}} \right\rceil + 1 \qquad (5.2)$$

Der zusätzliche Speicherplatz ist notwendig, da für die Zeitdauer einer Aktualisierung der älteste Platz als ungültig markiert werden muss und nicht mehr lesbar ist.

Auswahl von Objekten zu einem Zeitpunkt

Wie in Abb. 5.2 zu sehen, wird jede neue Version der Nutzdaten eines RTDB-Objekts $\mathcal{D}$ unter anderem mit zwei Zeitstempeln versehen:

$$\mathcal{D} = (t_{committed,\mathcal{D}}, t_{data,\mathcal{D}}, n_{bytes,\mathcal{D}}, DATA_{user,\mathcal{D}}, \dots)$$

$t_{committed}$ ist die automatisch hinzugefügte Eintragungszeit in die KogMo-RTDB. t_{data} ist der vom eintragenden Prozess zu liefernde Gültigkeitszeitpunkt der Daten, der bei einer optisch erkannten Fahrspur beispielsweise der Entstehungszeitpunkt des Videobildes ist. Er wird in der Wahrnehmungskette einfach weitergereicht, sofern keine zeitliche Interpolation stattfindet. Für ein aus den Daten von Objekt $\mathcal{D}_1$ entstandenes Ergebnisobjekt $\mathcal{D}_2$ ist vom verarbeitenden Modul zu setzen:

$$t_{data,\mathcal{D}_2} := t_{data,\mathcal{D}_1} \qquad (5.3)$$

Mithilfe der Datenzeitstempel t_{data} lassen sich in der Realität zeitlich zusammengehörige Daten $\mathcal{D}_i$ zum Zeitpunkt t in der Datenbank finden, auch wenn diese aufgrund unterschiedlicher Verarbeitungszeiten erst zu verschiedenen Zeitpunkten $t_{committed,\mathcal{D}_i}$ in der KogMo-RTDB veröffentlicht und möglicherweise inzwischen durch eine neuere Version überschrieben wurden. Dazu dient die Methode `readdata(`$\mathcal{D}, t_{data} = t$`)`.

Für ein beliebiges aus der KogMo-RTDB entnommenes Objekt $\mathcal{D}$ lässt sich dessen Datenalter zum Zeitpunkt t berechnen:

$$t_{age,\mathcal{D}} = t - t_{data,\mathcal{D}} \qquad (5.4)$$

Zeitbedingungen für lesende Algorithmen

Aufgrund der Natur eines Ringspeichers steht eine Information dort nicht unbegrenzt zur Verfügung, sondern wird, wenn inzwischen alle anderen Speicherplätze belegt sind, überschrieben. Die Gefahr von Inkonsistenzen wird durch das im nächsten Abschnitt präsentierte Protokoll vermieden.

Das absehbare Verlorengehen von Informationen älter als $T_{history}$ setzt Grenzen für die maximale Ausführungszeit von (Lese-)Algorithmen, die eine Arbeitsmenge von n Datenobjekten $\mathcal{S}_{work} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ benötigen. Um harten Echtzeitanforderungen gerecht zu werden, kann es die KogMo-RTDB, wie in Abschnitt 5.5.5 erläutert, nicht zulassen, Schreiboperationen zu blockieren.

Die Verarbeitung kann frühestens beginnen, wenn alle Daten in der KogMo-RTDB verfügbar sind:

$$t_{start,proc} = \max_{\mathcal{D}_i \in \mathcal{S}} \{t_{committed,\mathcal{D}_i}\}$$

Um mit einem zeitlich konsistenten Abbild der Umwelt zu arbeiten, benötigt ein Verarbeitungsalgorithmus gewöhnlich alle Daten eines bestimmten Zeitpunkts $t_{data,proc}$. Die verwendeten Daten haben damit alle das Alter

$$t_{age,proc} = t_{start,proc} - t_{data,proc} \tag{5.5}$$

Unter Berücksichtigung der verfügbaren Historie jedes Datenobjekts muss die Ausführungszeit $c_{read,proc}$ im ungünstigsten Fall folgender Bedingung genügen:

$$c_{read,proc} < \min_{\mathcal{D}_i \in \mathcal{S}} \{T_{history,\mathcal{D}_i}\} - t_{age,proc}$$

Die Zeit $c_{read,proc}$ beinhaltet nur denjenigen Teil eines jeden Algorithmus, der vom Startzeitpunkt $t_{start,proc}$ bis zum Lesen des letzten RTDB-Objektes $\mathcal{D}_n$ reicht. Die gesamte Rechenzeit c_i nach (5.17) ist eine obere Grenze für $c_{read,proc}$ und kann verwendet werden, wenn Letztere unbekannt ist.

Mit den Methoden der KogMo-RTDB ist es möglich, auf eine vergangene Version eines RTDB-Objektes unter Angabe von $t_{committed}$ oder t_{data} zuzugreifen. Hierbei kann auch nach verfügbaren früheren oder späteren Versionen gefragt werden, beispielsweise um Tendenzen zu berechnen oder nach Mustern zu suchen. Werden Objekte benötigt, die vor einer Zeit Δt veröffentlicht wurden, steht dem abrufenden Prozess eine noch kürzere Zeit zur Verfügung:

$$c_{read,proc} < \min_{\mathcal{D}_\rangle \in \mathcal{S}} \{T_{history,D_i}\} - t_{age,proc} - \Delta t \tag{5.6}$$

5.3.5 Blockierungsfreiheit und Datenkonsistenz

Das Konzept der KogMo-RTDB hat die Absicht, die Offenheit und den Austausch von Daten zwischen Modulen verschiedener Entwickler zu fördern. Daher dürfen einem Softwaremodul, das Daten publiziert, keine Nachteile entstehen, wie beispielsweise die Gefahr

von Blockierungen. Wäre das nicht gewährleistet, würde das dazu führen, dass Entwickler Daten gar nicht oder erst dann veröffentlichen, wenn das eigene Modul sonst nichts anderes mehr zu tun hat.

Blockierungsfreies Schreib-/Leseprotokoll

Aus diesem Grund wurde für die KogMo-RTDB ein Schreib- und Leseprotokoll entwickelt, das ganz ohne Blockierungen auskommen kann, ähnlich dem in [113] vorgestellten Ansatz. Damit können schnelle zeitkritische Wahrnehmungs- und Regelungsprozesse ihre Daten über die RTDB austauschen, ohne durch langsamere Module unkontrolliert blockiert zu werden.

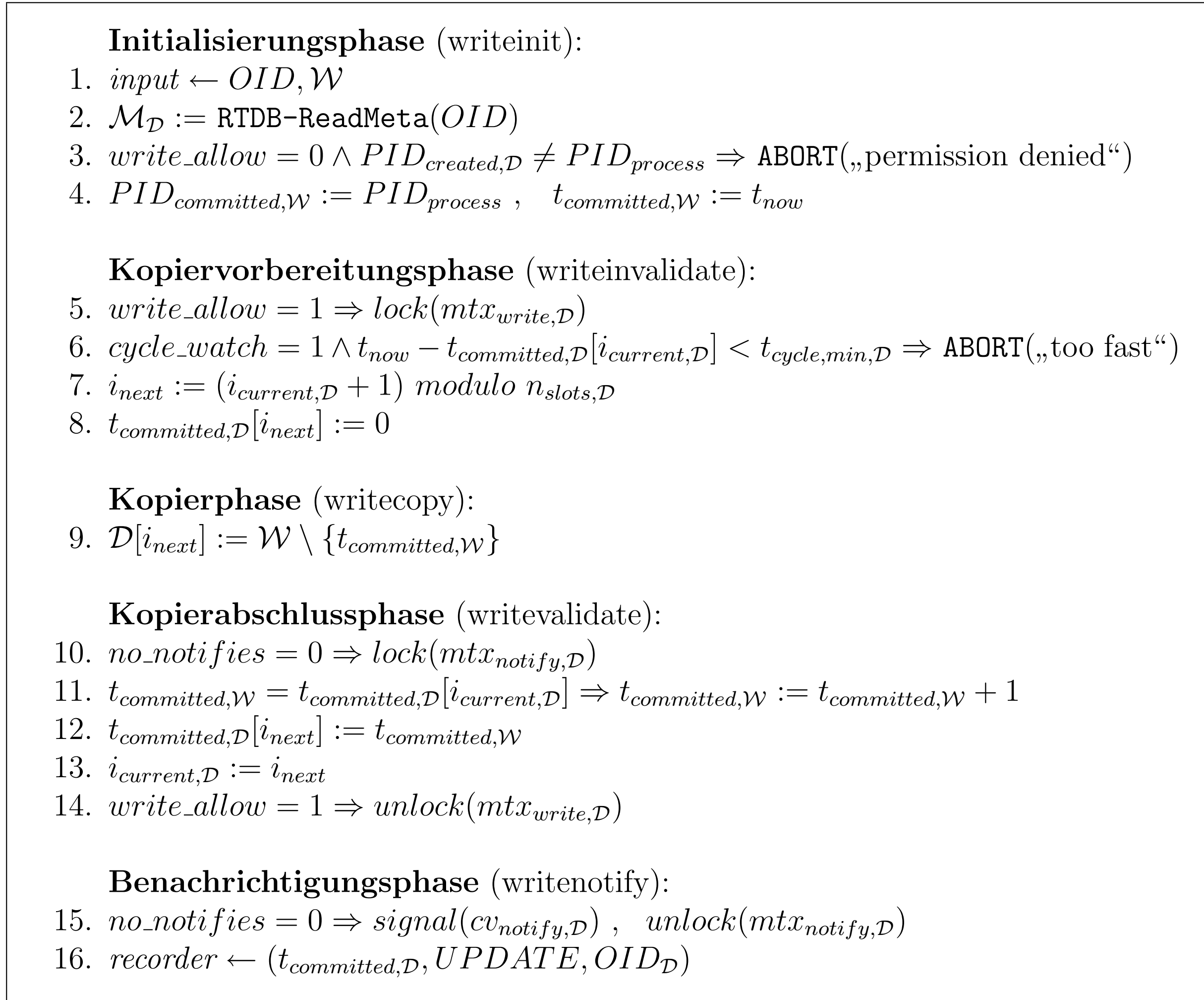
Der im Kognitiven Automobil beobachtete Regelfall ist, dass ein Objekt von einem einzelnen Modul beschrieben und von mehreren gelesen wird. Daher wurde die Blockierungsfreiheit für diesen Standardfall vorgesehen. Im Fall mehrerer Schreibprozesse, der vom Objekt-erstellenden Prozess explizit durch Vergabe eines öffentlichen Schreibrechts genehmigt werden muss, wird für die Dauer der Schreiboperation ein Mutex gesperrt, das gleichzeitige Schreibzugriffe verhindert. Lesezugriffe geschehen grundsätzlich blockierungsfrei und damit ohne Wechselwirkung mit Schreiboperationen.

Schreibalgorithmus

Den Algorithmus der KogMo-RTDB zum Schreiben eines neuen Nutzdatensatzes $\mathcal{W}$ in den Ringspeicher eines RTDB-Objektes zeigt Abb. 5.3. Sein Ablauf wird in Anhang A.1 detailiert erläutert. Er ist in 5 Phasen unterteilt, die für den Echtzeitnachweis in Gleichung (5.23) maßgeblich sind:

- In der *Initialisierungsphase* werden Eingangsdaten beschafft und Rechte geprüft.
- Während der *Kopiervorbereitungsphase* wird der Zielspeicherplatz i_{next} errechnet und mit $t_{committed}[i_{next}] := 0$ invalidiert. Bei öffentlichem Schreibrecht wird dies durch ein Mutex abgesichert.
- Erst in der *Kopierphase* wird der neue Datenblock $\mathcal{W}$ an sein Ziel kopiert, das dabei weiterhin ungültig markiert bleibt.
- Nach der *Kopierabschlussphase* sind die neuen Daten durch korrekten Zeitstempel und aktualisierten Index $i_{current,\mathcal{D}}$ zum Lesen freigegeben.
- In der abschließenden *Benachrichtigungsphase* werden wartende Prozesse aufgeweckt. Zur Vermeidung einer Wettlaufsituation wird die Signalisierung einer Zustandsvariable mit einem Benachrichtigungsmutex (vgl. Abschnitt 5.3.6) geschützt. Zudem erhält der Aufzeichnungsprozess das Ereignistupel EV aus Kapitel 5.7.1.

Durch das Design des Schreibalgorithmus ist sichergestellt, dass ein Prozess die während der Kopierphase inkonsistenten Daten niemals als gültig vorfinden kann. Schreibopera-



Initialisierungsphase (writeinit):

1. $input \leftarrow OID, \mathcal{W}$
2. $\mathcal{M}_{\mathcal{D}} :=$ `RTDB-ReadMeta`(OID)
3. $write_allow = 0 \wedge PID_{created,\mathcal{D}} \neq PID_{process} \Rightarrow$ `ABORT`(„permission denied“)
4. $PID_{committed,\mathcal{W}} := PID_{process}$, $t_{committed,\mathcal{W}} := t_{now}$

Kopiervorbereitungsphase (writeinvalidate):

5. $write_allow = 1 \Rightarrow lock(mtx_{write,\mathcal{D}})$
6. $cycle_watch = 1 \wedge t_{now} - t_{committed,\mathcal{D}}[i_{current,\mathcal{D}}] < t_{cycle,min,\mathcal{D}} \Rightarrow$ `ABORT`(„too fast“)
7. $i_{next} := (i_{current,\mathcal{D}} + 1)$ $modulo$ $n_{slots,\mathcal{D}}$
8. $t_{committed,\mathcal{D}}[i_{next}] := 0$

Kopierphase (writecopy):

9. $\mathcal{D}[i_{next}] := \mathcal{W} \setminus \{t_{committed,\mathcal{W}}\}$

Kopierabschlussphase (writevalidate):

10. $no_notifies = 0 \Rightarrow lock(mtx_{notify,\mathcal{D}})$
11. $t_{committed,\mathcal{W}} = t_{committed,\mathcal{D}}[i_{current,\mathcal{D}}] \Rightarrow t_{committed,\mathcal{W}} := t_{committed,\mathcal{W}} + 1$
12. $t_{committed,\mathcal{D}}[i_{next}] := t_{committed,\mathcal{W}}$
13. $i_{current,\mathcal{D}} := i_{next}$
14. $write_allow = 1 \Rightarrow unlock(mtx_{write,\mathcal{D}})$

Benachrichtigungsphase (writenotify):

15. $no_notifies = 0 \Rightarrow signal(cv_{notify,\mathcal{D}})$, $unlock(mtx_{notify,\mathcal{D}})$
16. $recorder \leftarrow (t_{committed,\mathcal{D}}, UPDATE, OID_{\mathcal{D}})$

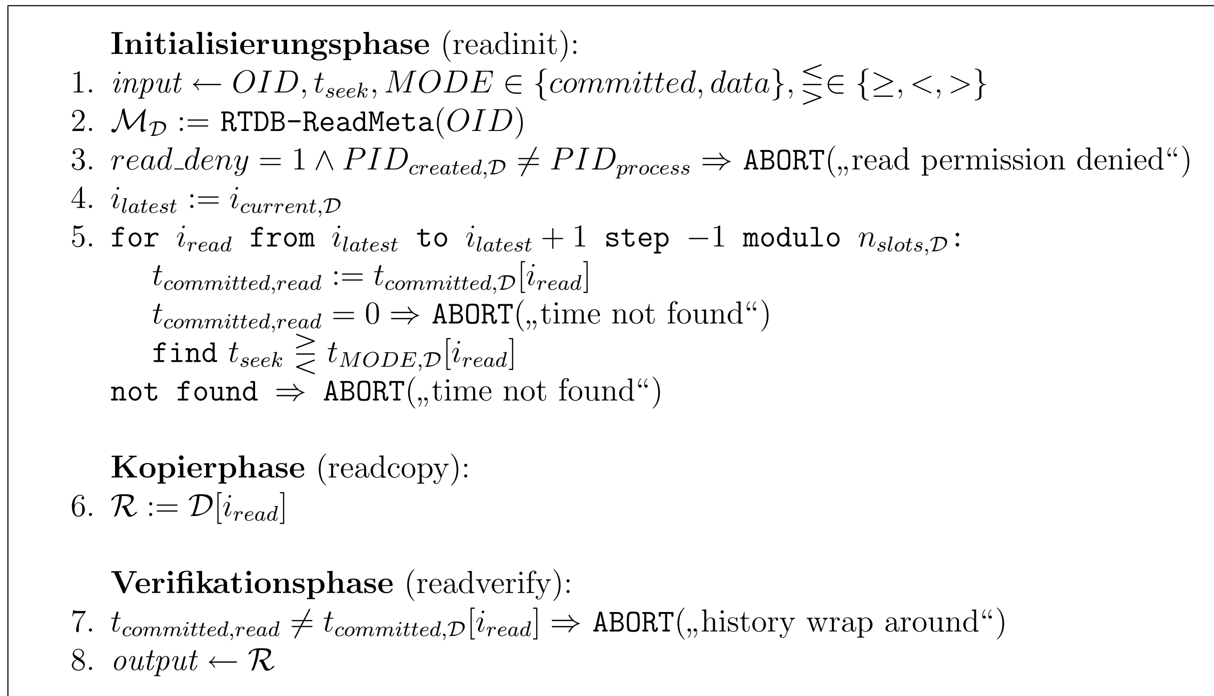
Abbildung 5.3: Algorithmus zum Schreiben der Daten eines RTDB-Objekts

tionen von $t_{committed}$ und $i_{current}$ sind atomar, bei 64 Bit Zeitstempeln auf einem 32 Bit System darf nur der höherwertige Teil zur Validitätsprüfung herangezogen werden.

Lesealgorithmus

Der dem entwickelten Protokoll entsprechende Lesealgorithmus der KogMo-RTDB für einen bestimmten Nutzdatensatz $\mathcal{R}$ ist in Abb. 5.4 gezeigt und ist in Anhang A.2 im Detail erklärt. Er besteht aus drei Phasen:

- In der *Initialisierungsphase* wird der durch OID, Zeitpunkt t_{seek} und Suchmodus spezifizierte Ringspeicherplatz gefunden.
- Während der *Kopierphase* wird eine lokale Kopie $\mathcal{R}$ der Nutzdaten generiert.
- Die *Verifikationsphase* dient dazu, die Konsistenz der Kopie anhand des gültigkeitsbestimmenden Zeitstempels zu prüfen und nur fehlerfreie Daten zurückzuliefern.



Initialisierungsphase (readinit):
1. $input \leftarrow OID, t_{seek}, MODE \in \{committed, data\}, \lesseqgtr \in \{\geq, <, >\}$
2. $\mathcal{M}_{\mathcal{D}} :=$ `RTDB-ReadMeta`(OID)
3. $read_deny = 1 \wedge PID_{created,\mathcal{D}} \neq PID_{process} \Rightarrow$ `ABORT`(„read permission denied“)
4. $i_{latest} := i_{current,\mathcal{D}}$
5. `for` i_{read} `from` i_{latest} `to` $i_{latest} + 1$ `step` -1 `modulo` $n_{slots,\mathcal{D}}$`:`
   $t_{committed,read} := t_{committed,\mathcal{D}}[i_{read}]$
   $t_{committed,read} = 0 \Rightarrow$ `ABORT`(„time not found“)
   `find` $t_{seek} \gtreqless t_{MODE,\mathcal{D}}[i_{read}]$
   `not found` $\Rightarrow$ `ABORT`(„time not found“)

Kopierphase (readcopy):
6. $\mathcal{R} := \mathcal{D}[i_{read}]$

Verifikationsphase (readverify):
7. $t_{committed,read} \neq t_{committed,\mathcal{D}}[i_{read}] \Rightarrow$ `ABORT`(„history wrap around“)
8. $output \leftarrow \mathcal{R}$

Abbildung 5.4: Algorithmus zum Lesen der Daten eines RTDB-Objekts

Kann der Lesealgorithmus Daten nicht mehr finden, oder werden diese während der Kopierphase überschrieben, tritt ein Fehler auf, auf den der lesende Prozess angemessen reagieren muss. Bei einer korrekten Dimensionierung von $T_{history}$ nach (5.8) und gleichzeitigem Einsatz der Schreibratenüberwachung (*cycle_watch*) wird dieses Problem vermieden.

Ein ähnliches nichtblockierendes Protokoll wird auch beim Lesen des Metadatenblocks $\mathcal{M}$ eines Objekts eingesetzt.

Einfluss von Schreibratenüberschreitung

Das präsentierte Schreib-/Leseprotokoll macht einen sehr effektiven Gebrauch von den zugrundeliegenden Ringspeicherstrukturen. Voraussetzung für ein Bestehen der Historie der Dauer $T_{history}$ ist jedoch, dass der schreibende Prozess die selbst definierte maximale Schreibrate $\frac{1}{t_{cycle,\mathcal{D}}}$ einhält oder dies von der KogMo-RTDB durch Setzen des *cycle_watch*-Parameters im Schreibalgorithmus aus Abb. 5.3 erzwingen lässt. Erfolgen hochfrequentere Aktualisierungen mit $\frac{1}{t_{cycle,writer}}$, sinkt die effektive Historienlänge $T_{history,eff}$ auf

$$T_{history,eff} = \frac{T_{history,\mathcal{D}} \cdot t_{cycle,min,\mathcal{D}}}{t_{cycle,writer}}$$

Da die Konfigurationsparameter wie *cycle_watch* bereits beim Anlegen von Objekten gesetzt werden und dann nicht mehr modifizierbar sind, ist diese Gefahr den lesenden

Prozessen a priori bekannt. So könnte der Betrieb eines Systems nicht freigegeben werden, wenn Module sich dieser Überwachung nicht unterwerfen. Von der KogMo-RTDB selbst sollte dieser Zwang aus konzeptionellen Gründen nicht auferlegt werden.

Auswirkungen der Lesedauer

Die Ausführungszeit einer Leseoperation setzt sich zusammen aus der Kopierdauer des Datenblocks sowie der Dauer der Initialisierungs- und Verifikationsphase des Algorithmus aus Abb. 5.4:

$$c_{read,\mathcal{D}} = c_{readinit} + c_{readcopy,\mathcal{D}} + c_{readverify}$$

Dabei ist nur $c_{readcopy,\mathcal{D}}$ von der Objektgröße $n_{bytes,\mathcal{D}}$ abhängig und benötigt, wenn $c_{copy,byte}$ die Kopierdauer eines einzelnen Bytes angibt:

$$c_{readcopy,\mathcal{D}} = c_{copy,byte} \cdot n_{bytes,\mathcal{D}}$$

Mit (5.1) lässt sich dies beschränken auf:

$$c_{readcopy,\mathcal{D}} \leq c_{copy,byte} \cdot n_{bytes,max,\mathcal{D}}$$

Somit ergibt sich für die *Worst-Case* Ausführungszeit einer RTDB-Leseoperation:

$$c_{read,\mathcal{D}} \leq c_{readinit} + c_{copy,byte} \cdot n_{bytes,max,\mathcal{D}} + c_{readverify} \tag{5.7}$$

Alle Datenbankoperationen werden im Kontext des aufrufenden Prozesses ausgeführt und laufen mit dessen Priorität. Daher ist es bei größeren Objekten wahrscheinlich, dass der Lesealgorithmus auf einem präemptiven Betriebssystem von höherprioren Prozessen unterbrochen wird. Aufgrund der zeitlichen Pufferung und des Protokolls hat dies keinen Einfluss auf die Konsistenz der Daten, lediglich die Ausführungsdauer des Algorithmus verlängert sich bei regelmäßiger Unterbrechung um einen Faktor $\tau_{preempt}$. Zu diesem Faktor können auch Hardwareeinflüsse von Prozessoren untereinander wie in [24] oder der Peripherie wie in [196] beitragen.

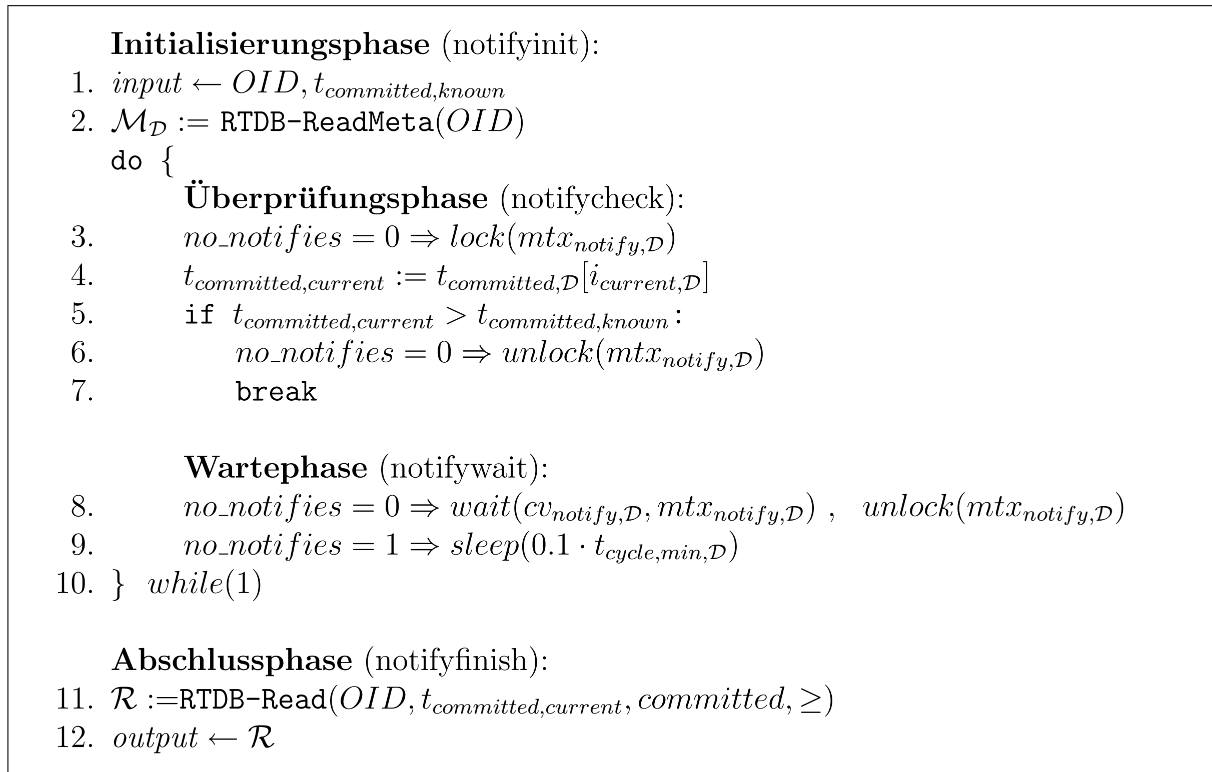
Damit der Lesealgorithmus aus der Historie ein $t_{age} + \Delta t$ (vgl. (5.5)) altes Objekt erfolgreich in den Kontext des aufrufenden Prozesses kopieren kann, muss folgende Bedingung erfüllt sein:

$$c_{read,\mathcal{D}} \cdot \tau_{preempt} \leq T_{history} - t_{age} - \Delta t$$

Zur Dimensionierung von $T_{history}$ folgt daraus für den *Worst Case*:

$$T_{history} \geq (c_{readinit} + c_{copy,byte} \cdot n_{bytes,max,\mathcal{D}} + c_{readverify}) \cdot \tau_{preempt} + t_{age} + \Delta t \tag{5.8}$$

Da für große Datenobjekte wie Videobilder $c_{copyread,\mathcal{D}}$ dominiert, wie in [74] und Kapitel 6.1.2 gemessen, bietet sich dort die Verwendung direkter Zeigerzugriffe auf den Speicher der KogMo-RTDB an, sodass der Kopieraufwand entfällt. Dabei ist jedoch nach jedem Verarbeitungsschritt die Gültigkeit der Ausgangsdaten manuell zu prüfen.



Initialisierungsphase (notifyinit):
1. $input \leftarrow OID, t_{committed,known}$
2. $\mathcal{M}_{\mathcal{D}} :=$ `RTDB-ReadMeta`(OID)

`do {`

Überprüfungsphase (notifycheck):

3. $no_notifies = 0 \Rightarrow lock(mtx_{notify,\mathcal{D}})$
4. $t_{committed,current} := t_{committed,\mathcal{D}}[i_{current,\mathcal{D}}]$
5. `if` $t_{committed,current} > t_{committed,known}$:
6. $no_notifies = 0 \Rightarrow unlock(mtx_{notify,\mathcal{D}})$
7. `break`

Wartephase (notifywait):

8. $no_notifies = 0 \Rightarrow wait(cv_{notify,\mathcal{D}}, mtx_{notify,\mathcal{D}})$, $unlock(mtx_{notify,\mathcal{D}})$
9. $no_notifies = 1 \Rightarrow sleep(0.1 \cdot t_{cycle,min,\mathcal{D}})$
10. `}` $while(1)$

Abschlussphase (notifyfinish):

11. $\mathcal{R} :=$`RTDB-Read`$(OID, t_{committed,current}, committed, \geq)$
12. $output \leftarrow \mathcal{R}$

Abbildung 5.5: Algorithmus zur Benachrichtigung bei neuen Daten eines RTDB-Objekts

5.3.6 Benachrichtigungsmethode

Der Benachrichtigungsalgorithmus ist ein elementarer Teil des entwickelten Protokolls und ergänzt es um ein blockierendes Warten auf neue Daten (`readdata_waitnext()`). Da hier Mutexe genutzt werden, hat sein Ablauf unmittelbare Auswirkungen auf das Zeitverhalten des Schreibalgorithmus und wird daher an dieser Stelle präsentiert.

Das Konzept der Benachrichtigung in der KogMo-RTDB besteht darin, dafür zu sorgen, dass ein Prozess stets mit den aktuellsten Daten arbeitet und niemals blockiert, wenn neuere Daten verfügbar sind. Wurden zwischen zwei Verarbeitungszyklen eines Prozesses mehrere Versionen eines Eingangsdatenobjekts geschrieben, sollte der Prozess nicht versuchen, alte Versionen aufzuarbeiten. Sonst besteht die Gefahr, dass er auf zunehmend älteren Daten arbeitet. Stattdessen sollte er über geeignete Algorithmen verfügen, um erforderliche Zeitsprünge beim Aufholen entsprechend zu berücksichtigen. Alte Daten lassen sich dennoch jederzeit aus der RTDB-Historie abfragen.

Den Algorithmus zur Benachrichtigung zeigt Abb. 5.5:

- Für ein gegebenes RTDB-Objekt OID wird durch den bekannten Aktualisierungszeitpunkt $t_{committed,known}$ eindeutig dessen zuletzt gelesene Version spezifiziert (1.). Damit werden Wettlaufsituationen (*race conditions*) zwischen Prozessen ausgeschlossen.

- In der *Überprüfungsphase* wird der aktuelle Schreibzeitpunkt $t_{committed,current}$ gelesen (4.). Ist dieser inzwischen jünger als der gegebene (5.), wird die Warteschleife abgebrochen (7. → 11.).
- In der *Wartephase* wird auf ein Signal eines anderen Prozesses Prozesses gewartet, das signalisiert, dass neue Daten verfügbar sind (vgl. 15. in Abb. 5.3). Wenn kurz nach dem Lesen des letzten Zeitpunkts (4.) eine Aktualisierung erfolgt, würde die nachfolgende Warteoperation auf eine weitere Aktualisierung warten und diese Version übergehen. Daher wird im Schreibalgorithmus für die Dauer der Objektaktualisierung (Inkrementierung von $i_{current,\mathcal{D}}$) ein Mutex $mtx_{notify,\mathcal{D}}$ gesperrt. Dieser wird während der Überprüfung ebenfalls belegt (3., 6.) und verhindert eine gleichzeitige Änderung. Der Systemaufruf zum Warten auf die Signalisierung der Zustandsvariable $cv_{notify,\mathcal{D}}$ (8.) gibt diesen Mutex, entsprechend der Mesa-Semantik [117], erst bei Wartebeginn atomar wieder frei. So ist auch die Wettlaufsituation bei der Zustandsprüfung ausgeschlossen.

  Die automatisch erfolgende Wiederbelegung des Mutex wird nicht benötigt, daher wird er sofort wieder freigegeben (8.). Die beiden Phasen werden kontinuierlich wiederholt (10.), bis der Abbruch (7.) aufgrund einer Objektänderung erfolgt.
- In der *Abschlussphase* werden die neuen Objektdaten mit dem bekannten Lesealgorithmus aus Abb. 5.4 vollständig gelesen (11.) und zurückgeliefert (12.).

Sind Benachrichtigungen unterbunden ($no_notifies = 1$), kann nicht auf ein Signal gewartet werden (8.) und es muss periodisch (9.) überprüft werden (*Polling*), ob sich der letzte Schreibzeitpunkt geändert hat. Die optimale Überprüfungsperiode t_{poll} ist vom jeweiligen Objekt abhängig, sinnvolle Werte sind z. B. $0.1 \ldots 0,5 \cdot t_{cycle,min}$. Bei kurzen Perioden steigt die Rechenlast, bei langsamen die maximale Latenzzeit t_{IPC}, bis eine Änderung festgestellt wird.

Ein ähnliches Benachrichtigungsprotokoll wird auch beim Warten auf neue Objekte mit vorgegebenen Metadaten eingesetzt.

5.3.7 Speicherverwaltung

Ein entscheidendes Designkriterium der KogMo-RTDB ist es, harten Echtzeitbedingungen zu genügen. Daher muss der Echtzeitnachweis nicht nur die Software, einschließlich des Betriebssystems, abdecken, sondern auch die benötigten Hardwarekomponenten. In [196] werden echtzeitfähige Methoden für den Zugriff auf Festplatten diskutiert. Die Zeit für einen Zugriff wird dabei im ungünstigsten Fall nicht nur durch das Konstruktionsprinzip der Festplatte mit einer rotierenden Scheibe und einem bewegten Schreib-Lesekopf, sondern zusätzlich durch die in [68] beschriebenen Bus-Verbindungen verschlechtert.

Um auch im ungünstigsten Fall (*Worst Case*) annehmbare Ausführungszeiten für alle Operationen zu garantieren, ist die KogMo-RTDB rein hauptspeicherbasiert und macht im Gegensatz zu klassischen Datenbanksystemen von sich aus keine Festplattenzugriffe.

Damit sind wie in Abschnitt 6.1 gezeigt, Aktualisierungsraten von Fahrzeugdaten im kHz-Bereich möglich, bei gleichzeitigem Transport von mehreren Kameradatenströmen in der Größenordnung von mehreren 10 MB/Sekunde. Wie in Kapitel 5.7 beschrieben, existieren jedoch Methoden, um Daten auf Massenspeicher mitzuprotokollieren und wieder einzuspielen.

Dynamische Speicherallokation

Die Beschränkung auf Hauptspeicherverwendung allein macht ein Datenbanksystem noch nicht echtzeitfähig. Im Regelfall wird der verfügbare Hauptspeicher (*Random Access Memory*, RAM) vom Betriebssystem verwaltet. Benötigt ein darauf laufendes Programm zusätzlichen Arbeitsspeicher, muss es diesen vom Betriebssystem anfordern. Dessen Methoden zur dynamischen Speicherallokation (*Dynamic Storage Allocation*, DSA) sind, wie in Kapitel 3.5.3 untersucht, vor allem auf einen hohen Durchsatz, und damit eine gute durchschnittliche Antwortzeit, sowie auf niedrige Speicherfragmentierung optimiert. Daher können keine maximalen Antwortzeiten für den ungünstigsten Fall garantiert werden. Zudem können bei der Speichervergabe unvorhersehbare Blockierungen durch andere, auf demselben System laufende und nicht echtzeitfähige Programme auftreten. In dieser Situation ist ein realistischer Echtzeitnachweis unmöglich.

Daher wurden für die Verwendung in der KogMo-RTDB eine eigene echtzeitfähige DSA-Strategie entwickelt. Diese besteht aus folgenden Bausteinen:

Initiale Speicherreservierung durch das Betriebssystem:
In der Initialisierungsphase der KogMo-RTDB wird ein ausreichend großer Speicherbereich vom Betriebssystem angefordert. Dabei besteht kein Bedarf an Echtzeitfähigkeit. Zur Laufzeit der KogMo-RTDB hat das Betriebssystem keinen Zugriff mehr auf diesen Speicherblock und somit können dessen Allokationsmechanismen für den Echtzeitnachweis ausgeklammert werden. Die Größe des Speicherblocks ist abhängig von der Anzahl und der Größe der benötigten Ringspeicher nach (5.10) für die Objekte aus der Gesamtmenge aller RTDB-Objekte $\mathcal{S}_{RTDB}$:

$$n_{bytes,RTDB} = s \cdot \sum_{\mathcal{D} \in \mathcal{S}_{RTDB}} n_{bytes,dataringbuffer,\mathcal{D}} \tag{5.9}$$

$n_{bytes,RTDB}$ kann zwischen $0.5 \cdot 10^6 Byte$ bis mehrere $10^9 Byte$ betragen. Wie in Abschnitt 3.5.2 erläutert, kann durch Speicherfragmentierung der effektiv nutzbare Speicher kleiner ausfallen. Dies muss durch einen Sicherheitsfaktor s berücksichtigt werden.

Echtzeitfähige Allokationsmechanismen für neue RTDB-Objekte zur Laufzeit:
Es darf nicht übersehen werden, dass das Design von Allokationsalgorithmen ein eigenes Forschungsgebiet ist. Dabei hat jeder Algorithmus seine Vor- und Nachteile, oftmals geht man Kompromisse ein. Kapitel 3.5.1 erläutert die Anforderungen an echtzeitfähige Allokationsalgorithmen. In der KogMo-RTDB wird ein zeitbeschränkter *good-fit* Allokationsalgorithmus namens *Two-Level Segregated Fit* [134]

(TLSF) eingesetzt. Dieser hat eine Komplexität von O(1) und ist ein Kompromiss bzgl. Fragmentierung (≠*best-fit*). Er ist frei im Quellcode erhältlich und wurde für die KogMo-RTDB angepasst. Nach [133] bewegt sich dessen Fragmentierung auch im Worst-Case unter 30%.

Anstelle von TLSF kann prinzipiell jeder beliebige Algorithmus eingesetzt werden. Eine realisierte Alternative ist die „Einmalvergabe“, in der Speicher linear vergeben und nicht mehr wiedervergeben wird. Dies dient zum einem als schneller „Referenzalgorithmus“ zum Zeitvergleich. Zum anderen kann dies in Produktivsystemen verwendet werden, in denen Softwaremodule ihre benötigten RTDB-Objekte nur in der Initialisierungsphase reservieren und zur Laufzeit keine neuen Objekte anlegen.

Überwachung der verfügbaren Speicherressourcen: Damit im kognitiven Automobil nicht in einer kritischen Situation der Speicher für relevante RTDB-Objekte ausgeht, enthält die KogMo-RTDB einen Managerprozess, der regelmäßig den freien RTDB-Speicher berechnet. Diese Information wird als RTDB-Objekt publiziert. Sie kann von einem Monitoring-Prozess, wie in Abschnitt 6.3.3 gezeigt, genutzt werden, um rechtzeitig eine sichere Notbremsung einzuleiten. Darüber hinaus wird die Speicherstatistik auch in jeder RTDB-Aufzeichnung mitprotokolliert. Mithilfe dieser Messdaten kann der benötigte Speicher für eine konkrete Fahrzeugkonfiguration ermittelt werden.

Periodische Datenaktualisierung: Die beabsichtigte echtzeitfähige Aktualisierung der Nutzdaten im kHz-Bereich ist zu hoch für eine dynamische Speicherallokation und -deallokation einzelner Ringspeicherplätze. Daher wird der gesamte benötigte Ringspeicher beim Erstellen eines Objektes in einem Block alloziert.

Die Größe n_{bytes} eines jeden Nutzdatenblocks (vgl. Kapitel 5.3.1) kann von Aktualisierung zu Aktualisierung unterschiedlich groß sein. Dies ist z.B. für serialisierte Daten wie XML-Daten oder das in Abschnitt 6.4.1 vorgestellte Annotationsobjekt notwendig, um keine Rechenzeit und Aufzeichnungsbandbreite zu verschwenden.

Um den Umgang mit RTDB-Objekten dennoch deterministisch zu machen, muss eine maximale Größe $n_{bytes,max}$ spezifiziert werden. Anhand dieser Größe wird der gesamte Ringspeicherplatz für ein neues RTDB-Objekt $\mathcal{D}$ unter Verwendung von (5.2) dimensioniert:

$$n_{bytes,dataringbuffer,\mathcal{D}} = n_{slots,\mathcal{D}} \cdot n_{bytes,max,\mathcal{D}} \tag{5.10}$$

Um dynamische Strukturen beliebiger Größe zu realisieren, können einem Objekt $\mathcal{D}$ zusätzliche untergeordnete Objekte als „Kinder“ angehängt werden, indem bei ihnen als Vaterobjekt $OID_{parent} := OID_{\mathcal{D}}$ gesetzt wird. Die Suchfunktionen der KogMo-RTDB verfügen über Methoden, um diese zu finden.

Deallokation von RTDB-Objekten: Die KogMo-RTDB hat unter anderem die Aufgabe, langsameren Prozessen eine konsistente Sicht auf die Datenbank zu offerieren. Das genannte Ringspeichersystem hält vergangene Daten für die spezifizierte Hi-

storienzeitspanne $T_{historiy}$ verfügbar. Wird ein Objekt gelöscht, dürfen weder seine Metadaten noch sein Ringspeicher sofort gelöscht werden, sondern erst nach Ablauf der Historienzeit. Dies muss auch für den Fall garantiert werden, dass ein Prozess aufgrund eines Softwarefehlers vorzeitig beendet wird. Hier müssen ebenfalls dessen RTDB-Objekte kurzzeitig erhalten bleiben, beispielsweise um die letzten verfügbaren Umfeldinformationen für ein sicheres Anhalten zu nutzen.

Daher ist das endgültige Entfernen gelöschter RTDB-Objekte nach Ablauf der Historienzeit Aufgabe des RTDB-Managerprozesses. Der damit beauftragte Thread muss selbstverständlich in einem hart echtzeitfähigen Kontext laufen und eine ausreichende Priorität besitzen. Das verzögerte Löschen nach der Historienzeit lässt sich durch die Markierung *immediately_delete* im Parameter $flags$ der Objektmetadaten deaktivieren und wird wird z.B. vom RTDB-Player verwendet, damit der Benutzer nach dem Ende einer Aufzeichnung nicht warten muss, bis der genutzte Speicher wieder verfügbar wird.

Für hochpriore Prozesse wie LIDAR-Objekterkenner, die in kurzer Abfolge Objekte eines bestimmten Types anlegen und wieder löschen, existiert zudem eine Ressourcen schonende Alternative: Diese können mit der Markierung *keep_alloc* angelegt werden. Dadurch wird der allozierte Speicherbereich nach Löschen des Objektes dauerhaft oder für eine bestimmte Zeit (z.B. 10 s) freigehalten, um ein neues identisches Objekt des Prozesses ohne erneute Speicherallokation aufzunehmen. Ein ähnliches Vorgehen findet sich auch im *Slab Cache* aus Kapitel 3.5.3. Diese Wiederverwendung vermeidet im eingeschwungenen Zustand unnötige Rechenzeit, darf aber nicht für eine *Worst Case*-Analyse verwendet werden.

5.3.8 Applikationsschnittstellen

Um das Ziel eines projektübergreifenden Integrationsframeworks zu erreichen, ist es notwendig, dass sich verschiedene Softwaremodule verbinden lassen. Dabei bringen Forschungsinstitute teils Software mit, die spezielle Bibliotheken und bestimmte Compilerversionen benötigen. Manchmal können aus rechtlichen Gründen nur Binärversionen von Modulen weitergeben werden. Erfahrungen haben gezeigt, dass hier große Integrationshindernisse entstehen können, insbesondere da die Portierung auf andere Versionen keinen wissenschaftlichen Mehrwert generiert und die Motivation für einen solchen Aufwand gering ist. Daher ist es wichtig, dass die Applikationsschnittstellen und die dazugehörigen Bibliotheken von allen Version verschiedener Compiler genutzt werden können.

Binäre Schnittstellenkompatibilität

Voraussetzung für das erfolgreiche Zusammenbinden (*link*en) von Software ist ein übereinstimmendes ABI (*Application Binary Interface*) (vgl. [124]). Ein Problem ist, dass es kein offizielles ABI für Objektdateien gibt, die z.B. aus C oder C++ Quellcode entstanden sind. Jedoch gibt es für jede Plattform einen Quasi-Standard, der durch die Schnittstellen des

Betriebssystems und dessen dominanten Compiler vorgegeben ist. Da beispielsweise Linux hauptsächlich in C geschrieben ist, ist das C-ABI relativ stabil. So lässt sich problemlos eine C-Bibliothek mit Programmen verbinden, die mit dem freien GNU C-Compiler in allen Versionen von 2.x, 3.x und 4.x übersetzt wurde.

Für C++ sieht die Situation anders aus. Hier verwenden die existierenden Compiler verschiedene C++-ABI. Dabei existieren deutlich mehr Freiheitsgrade, unter anderem im Klassenlayout mit insbesondere dem Aufbau der virtuellen Funktionstabellen und in der Namensgestaltung [51]. Da Funktionen überladen werden können, wird z.B. eine eindeutige Methode zur Erstellung der Symbole in der Objektdatei benötigt. Daher lassen sich oftmals Objektdateien, die von verschiedenen Compilern erstellt wurden, nicht zusammenbinden.

Es existieren Standardisierungsvorschläge des C++-ABIs, wie in [28]. Jedoch hat sich beispielsweise das C++-ABI des GNU C++-Compilers von 3.0 über 3.2 und 3.4 mehrmals geändert [62]. Für die Zukunft besteht durchaus das Ziel eines standardisierten C++-ABIs [63].

C-Schnittstelle

Aus den bereits genannten Gründen und um eine Integration möglichst einfach zu machen, bietet die KogMo-RTDB ihre primäre Schnittstelle in Form einer C-Bibliothek an. Dies ist gleichzeitig der kleinste gemeinsame Nenner für die Zusammenarbeit mit anderen Programmiersprachen, sodass sich damit C++, Ada, Java und andere einbinden lassen. Die Anbindung eines Java-Moduls unter Verwendung des SWIG-Toolkits [12] wird in [118] verwendet. Eine ausführliche Schnittstellenbeschreibung der KogMo-RTDB findet sich in [70].

C++-Schnittstelle

Eine C++-Schnittstelle bietet die KogMo-RTDB in Form einer sogenannten „Wrapper"-Klasse an. Diese wird vom jeweiligen C++-Compiler übersetzt und arbeitet dann mit anderen Softwaremodulen über die darunterliegende C-Schnittstelle teilprojektübergreifend zusammen. Für selbstdefinierte Nutzdatenstrukturen können daraus eigene Klassen abgeleitet werden und sollten in einem projektweiten Quellcodeverzeichnis veröffentlicht werden. Die in der KogMo-RTDB gespeicherten Daten der Objekte können auch von nicht-objektorientierten Programmiersprachen aus gelesen werden.

Abb. 5.6 zeigt ein kurzes Programmbeispiel in C++, das die Einfachheit der entwickelten Schnittstelle zeigt. Darin werden kontinuierlich Kamerabilder aus der KogMo-RTDB geholt, diese an einen Fahrspurverfolger weitergereicht und die erkannte Fahrspur zurück in die Datenbank geschrieben. Das `RTDBConn`-Objekt stellt eine Verbindung zur lokalen Datenbank `local:system` her. Auf einem Simulationssystem können so mehrere Instanzen der KogMo-RTDB gleichzeitig unter verschiedenen Namen laufen. `RTDBReadWaitNext()`

```
// Neue KogMo-RTDB-Verbindung aufbauen, der eigene Modulname ist a2_roadtracker,
// die eigene Zykluszeit beträgt 33 Millisekunden
RTDBConn DBC ( "a2_roadtracker", 0.033, "local:system" );

// Warten auf ein Objekt mit dem Videobild der Kamera,
// das vom Teilprojekt C3 bereitgestellt wird
C3_Image Video ( DBC );
Video.RTDBSearchWait ( "c3_camera_left" );

// Einfügen eines Fahrspur-Objekts, das die Spursegmente mit
// einem Klothoidenmodell beschreibt
A2_RoadKloth Road ( DBC, "a2_egolane" );
Road.RTDBInsert ()

for(;;) {
// Warten, bis das nächste Kamerabild verfügbar ist, anschließend Lesen diese Bildes
Video.RTDBReadWaitNext ();

// Ausführung eines Durchlaufs des Fahrspurverfolgers mit dem neuen Bild
Roadtracker.Run ( Video.getImage () );

// Kopie der resultierenden Spursegmente in das Fahrspur-Objekt
Road.setKloth ( Roadtracker.getKloth () );

// Übertragung des Datenzeitstempels
Road.setTimestamp ( Video.getTimestamp () );

// Aktualisierung des Fahrspur-Objekts in der KogMo-RTDB mit den neuen Daten,
// dadurch Wecken aller Prozesse, die auf dessen Aktualisierung warten
Road.RTDBWrite ();

// Signalisierung der eigenen Funktionsfähigkeit für Überwachung und Diagnose,
// gleichzeitig Synchronisationspunkt bei Modulen, die mehrere Objekte aktualisieren
DBC.CycleDone ();
}
```

Abbildung 5.6: Beispiel zum Einsatz des KogMo-RTDB C++-APIs

nutzt die Benachrichtigungsmethode aus Abschnitt 5.3.6 und vermeidet ein ineffizientes kontinuierliches Dauerabfragen (*Polling*) nach neuen Bildern. Der eigentliche Fahrspurverfolger ist beschrieben in [150]. Zuletzt muss der Datenzeitstempel t_{data}, wie in Abschnitt 5.3.3 beschrieben, weitergereicht werden, um die zeitliche Einordnung des Ergebnisses in späteren Verarbeitungsstufen zu ermöglichen (vgl. (5.3)).

5.4 Harte Echtzeitfähigkeit

Die bisher vorgestellten Funktionalitäten haben unter einem GPOS wie Linux bereits einen wahrnehmbaren Anwendungsnutzen und zeichnen sich wie in Kapitel 6.1 gezeigt durch gute durchschnittliche Antwortzeiten aus. Um jedoch harten Echtzeitanforderungen gerecht zu werden, müssen die Reaktionszeiten des *gesamten Systems* deterministisch und nach oben beschränkt sein. Zu diesem Zweck ist eine kurze Systembetrachtung notwendig, um dann wieder zu einer reinen Softwareuntersuchung zurückzukehren.

5.4.1 Systembetrachtung

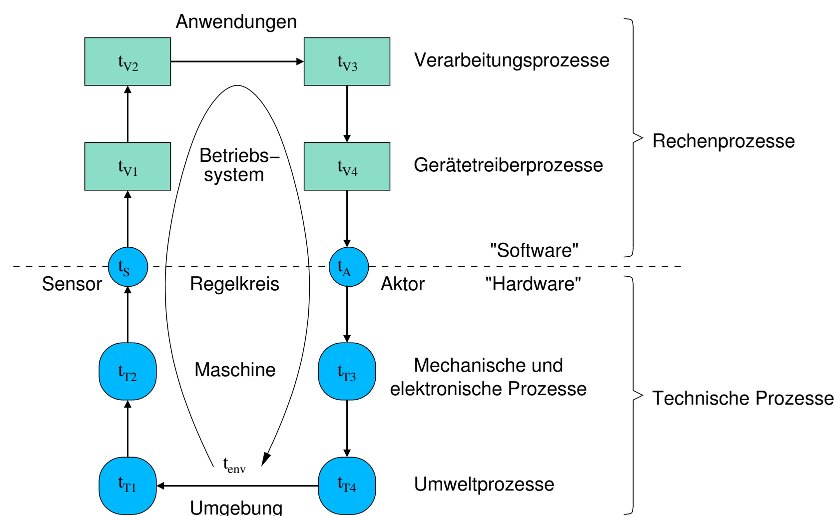


Abbildung 5.7: Einzelprozesse im Regelkreis eines technischen Systems

Abb. 5.7 zeigt eine vereinfachte Zerlegung des betrachteten Systems in repräsentative Einzelkomponenten, aufgeteilt in einen Hardware- und einen Softwareteil. Die zu realisierenden Reaktionszeiten werden durch die Umweltprozesse in der unmittelbaren Umgebung vorgegeben. Im Beispiel des fahrenden Automobils aus Kapitel 3.1 sind das andere Verkehrsteilnehmer, Hindernisse und der Verlauf der Straße in Kombination mit dem Bewegungszustand des eigenen Fahrzeugs.

Eine kritische Situation im Straßenverkehr benötigt beispielsweise eine Reaktion innerhalb einer Zeit d_{env} (*Deadline*). Dies kann eine Notbremsung oder der Beginn eines Ausweichmanövers sein. Die einzelnen Prozesse haben individuelle Totzeiten, die sich über den gesamten Regelkreis zu t_{sys} aufsummieren und für die gelten muss:

$$d_{env} \geq t_{sys,max} = \sum_i t_{T_i,max} + t_{S,max} + t_{A,max} + \sum_i t_{V_i,max} \tag{5.11}$$

Wenn die maximale Totzeit nur einer einzigen Komponente von t_{sys} gegen unendlich geht, lässt sich die maximale Reaktionszeit nicht nach oben beschränken:

$$\max t_{V_i} \rightarrow \infty \implies \nexists t_{sys,max} < \varepsilon$$

Daher ist es notwendig, jede involvierte Komponente zu identifizieren und ihr Zeitverhalten im ungünstigsten Fall zu analysieren:

Mechanische und elektronische Prozesse T_i: Schon innerhalb der technischen Prozesse können mechatronische Komponenten signifikante Totzeiten t_{T_i} aufweisen. Ein hydraulisches Bremssystem muss z.B. erst den notwendigen Bremsdruck aufbauen. Zudem können weitere eingebettete Systeme involviert sein, die eigene Rechenprozesse enthalten. Es wird hier vorausgesetzt, dass diese hart echtzeitfähig sind. Für die in Kapitel 6.4 beschriebenen autonomen Fahrzeuge trifft das z.B. für den Fahrzeugregler zu, der auf einer nicht ausgelasteten dSpace Autobox läuft.

Sensor S: Bei Sensor können prinzipbedingte Totzeiten hinzukommen. Ein mit 10 Hz rotierender Laserscanner, nimmt ein neu auftauchendes Objekt im ungünstigsten Fall erst nach einer Zeit $t_S = 100ms$ wahr. RADAR-Sensoren melden neue Objekte oft erst nach einer Validierungsphase über mehrere Messzyklen weiter.

In dieser Einteilung wird auch die Zeit, bis das Sensorsignal einem Rechenprozess zur Verfügung steht, der Sensortotzeit zugerechnet. Hierzu zählen auch, wie in Kapitel 3.3 ausgeführt, die rechnerinternen Bussysteme, die im Worst-Case deutlich höhere Blockierungszeiten als im Durchschnittsfall aufweisen können.

Aktor A: Unter der Aktortotzeit t_A werden hier alle Vorgänge subsummiert, die notwendig sind, damit das Ergebnis einer Berechnung wieder im technischen System spürbar ist. Die Grenze zu weiteren mechatronischen Prozessen soll hier nicht fest definiert werden, eine mögliche Grenze wäre das Magnetventil eines Bremssystems.

Die leicht zu übersehenden rechnerinternen Latenzzeiten wurden ebenfalls bereits in [68, 196] untersucht.

Rechenprozesse V_i: Abb. 5.7 zeigt eine Auswahl beteiligter Rechenprozesse. Gerade auf der Softwareebene können leicht wesentliche Prozesse vergessen werden, da sie im Gegensatz zu Hardware nicht „greifbar" sind und ihre Laufzeit im Normalfall nicht wahrnehmbar ist.

So müssen außer den algorithmischen Rechenprozessen auch alle Verbindungs- und Gerätetreiberprozesse, sowie das eingesetzte Betriebssystem betrachtet werden. Bei einem Standardbetriebssystem (GPOS) ist es üblich, dass alle Rechenprozesse auf dem Rechner nach einem Zeitanteilsverfahren ausgeführt werden, und so andere Prozesse die Ausführung kritischer Rechenprozesse verzögern können. Die in diesem Fall notwendige Analyse und Zeitbeschränkung sämtlicher Rechenprozesse einschließlich aller Programmpfade eines GPOS ist gewöhnlich nicht durchführbar, sodass hier echtzeitfähige Betriebssysteme wie aus Kapitel 3.4 eingesetzt werden, deren Latenzzeiten im Worst-Case bekannt sind.

Die Einordnung einzelner Prozesse in die genannten Kategorien kann abhängig vom Analyseschwerpunkt verschoben werden, solange keine Komponente übersehen wird. So könnte auch das gesamte Bremssystem als Aktor betrachtet werden oder der Fahrzeugregler in

die Softwareanalyse miteinbezogen werden, vorausgesetzt es wird keine wesentliche Komponente vernachlässigt.

5.4.2 Reaktionszeit des Rechnersystems

Echtzeitsysteme werden gerne unterschieden in ereignisgetriebene und zeitgetriebene Systeme. Die Zeitsteuerung in PC-basierten Systemen nutzt meist die verfügbaren Hardware-Zeitgeberbausteine, wie die prozessorinternen lokalen *APIC*s (*local Advanced Programmable Interrupt Controller*), deren Funktionsweise in [68] beleuchtet wird. Da die Zeitgeber periodisch einen Interrupt liefern, wird die Zeitsteuerung hier als ein Sonderfall der Ereignissteuerung betrachtet. Unter der Annahme, dass die eingesetzten Hardware-Zeitgeberbausteine eine hinreichende Genauigkeit besitzen, und der Tatsache, dass die rechnerinternen Buslaufzeiten für externe Interrupts länger sind als die der APICs, ist die Reaktion auf externe Ereignisse zeitkritischer als die Aktivierung periodischer Aufgaben.

Die Reaktionszeit des Rechnersystems t_{pcsys} auf ein Ereignis ist somit die Zeit zwischen einem eingehenden Hardwaresignal und der Reaktion mit einem ausgehenden Hardwaresignal. Wenn nicht auszuschließen ist, dass innerhalb der erlaubten Reaktionszeit ein weiteres gleichartiges Signal eingeht, muss durch Abschalten des Interrupts oder eine externe Hardwarevorrichtung zur Interruptkontrolle sichergestellt werden, dass die minimale Zwischenankunftszeit größer als die Reaktionszeit ist.

5.4.3 Komponenten der echtzeitfähigen Softwarearchitektur

Der Schwerpunkt dieser Arbeit ist es, die Echtzeitfähigkeit des Rechenbasissystems sicherzustellen. Dies umfasst die Schnittstellen am unmittelbaren Übergang zwischen Hard- und Softwareteil sowie die Basissoftware mit den Softwareschnittstellen zwischen Verarbeitungsmodulen. Komponenten der mechanischen und elektronischen Prozesse werden z. B. in [220, 218] untersucht, die Algorithmen von Softwaremodulen u. a. in [102].

Abb. 5.8 gibt einen Überblick über die hart echtzeitfähige Softwarearchitektur der Echtzeitdatenbank KogMo-RTDB. Auf oberster Ebene befinden sich die Softwaremodule, die die Algorithmen und Logik des kognitiven Systems enthalten. Diese sind als Standard-Linux Prozesse implementiert. Sie nutzen das API der KogMo-RTDB, das sich in der Bibliothek `libkogmo_rtdb_rt` befindet. Die KogMo-RTDB arbeitet oberhalb der Systembibliotheken. Auf der Echtzeitseite (links) nutzt sie das POSIX-Interface des Xenomai Echtzeit-Nucleus mit dem ADEOS Nanokernel. Für den echtzeitfähigen Hardwarezugriff werden Echtzeittreiber benutzt, die dem *Real-Time Driver Model* (RTDM) [106] entsprechen. Das Gesamtsystem kann aus einer beliebigen Mischung aus Echtzeitmodulen (links), nicht-Echtzeitmodulen (rechts) und Modulen, die im Betrieb zwischen hin- und herwechseln (mittig), bestehen. Diese Zuordnung entspricht auch der Farbgebung in Abb. 5.1. Im Folgenden werden die Kriterien für dieses Design näher erläutert.

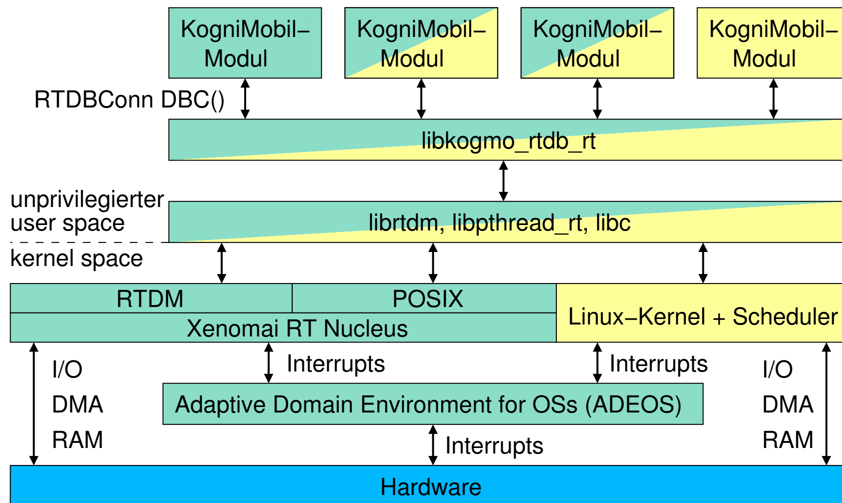


Abbildung 5.8: Hart echtzeitfähige Softwarearchitektur

5.4.4 Betriebssystem

Liegt der Schwerpunkt einer Architektur allein auf harten Echtzeiteigenschaften, ist der Einsatz eines einschlägigen RTOS, wie in Kapitel 3.4 präsentiert, zwingend. Das Ziel dieser Architektur ist jedoch zugleich, die nahtlose Integration von nicht-echtzeitfähigen Softwaremodulen zu unterstützen. Die Installation und Einarbeitung in ein neues Betriebssystem stellt eine deutliche Einstiegshürde dar. Daher wurde das unter den Entwicklern des Forschungsbereichs SFB/TR 28 verbreitete Standardbetriebssystem Linux als Basis genommen und darauf aufsetzende Echtzeiterweiterungen untersucht. Da der Linux-Kernel im Quellcode vorliegt, lassen sich echtzeitkritische Betriebssystembereiche im Detail untersuchen und notwendige Änderungen vornehmen.

Das Standardbetriebssystem Linux wurde jedoch nicht mit dem Designziel einer harten Echtzeitfähigkeit im Sinne der Festlegung aus Kapitel 3.2, sondern mit dem Ziel der bestmöglichen durchschnittlichen Verarbeitungszeit und eines hohen Durchsatzes entwickelt. Dabei werden, wie in Abschnitt 3.4.1 beschrieben, selten auftretende höhere Antwortzeiten in Kauf genommen. Das kann aber, abhängig vom einbettenden technischen System, katastrophale Auswirkungen haben.

Echtzeiterweiterung

Es existieren jedoch zahlreiche Ansätze, den Linux-Kernel um harte Echtzeitfähigkeiten zu ergänzen. Eine Auswahl wurde bereits in Kapitel 3.4.2 vorgestellt. Die meisten Lösungen implementieren dafür eine Zwei-Kerne-Technik, in der sie einen zweiten kleinen und äußerst deterministischen Echtzeitkern einführen, der streng vom Standard-Linux-Kern isoliert wird. Echtzeitfähige Softwaremodule müssen dort im ungeschützten Kontext des eigentlichen Betriebssystems im Kernadressraum (*Kernel Space*) ausgeführt werden, wo sie die bestmöglichen Interruptlatenzzeiten und direkten Hardwarezugriff bekommen. Die

dort laufenden Module haben jedoch keinen Speicherschutz, erschweren die Fehlersuche und erlauben nur begrenzten C++-Einsatz. Ein Softwarefehler auf Kernel-Ebene führt gewöhnlich zu einem Systemabsturz und erfordert einen Neustart. Unter diesen Bedingungen ist es nicht praktikabel, Softwaremodule auf Kernel-Ebene zu integrieren. Zudem kann es nicht allen Projektpartnern zugemutet werden, Kernel-Module zu programmieren.

Die vorgestellte Architektur basiert auf der bereits in Kapitel 3.4.2 vorgestellten Echtzeiterweiterung Xenomai [65]. Sie verwendet auch einen Zwei-Kern-Ansatz, ermöglicht jedoch die Ausführung von Echtzeitprozessen auf der geschützten Benutzerebene (*User Space*), auf der ihre Speicherbereiche durch das Betriebssystem voneinander geschützt sind und die eine bessere C++-Unterstützung bietet. Zudem verfolgt Xenomai das Ziel einer Integration mit dem Standard-Linux-Kern. Dadurch können für weniger zeitkritische Anforderungen zusätzlich weiche Echtzeiterweiterungen wie der *Real-Time Preemption Patch* [143] genutzt werden.

Nanokernel zur Ereignispriorisierung

Abb. 5.8 zeigt auf der untersten hardwarenächsten Ebene den Einsatz einer Schicht namens *Adaptive Domain Environment for Operating Systems* (ADEOS) [228]. Dies ist ein Nanokernel, der dazu dient, mehrere Betriebssysteme auf einem Rechner auszuführen. Im Gegensatz zu Virtualisierungslösungen wie Xen, VMWare, QEmu, usw. wird dabei nicht das Ziel einer strikten Isolation verfolgt, sondern das der Zusammenarbeit. Der Schwerpunkt liegt dabei auf der zeitlichen Steuerung. Daher wird die Verwaltung der Speicher- und Hardwareressourcen (RAM und E/A) dem Gastbetriebssystem Linux überlassen. So werden die durch ADEOS notwendigen Änderungen am Linux-Betriebssystem überschaubar und die Portierung auf zukünftige Betriebssystemversionen bleibt durchführbar. Zudem wird die gute Hardwareunterstützung des Linux-Kernels mitgenutzt, was den Entwicklungsaufwand weiter minimiert.

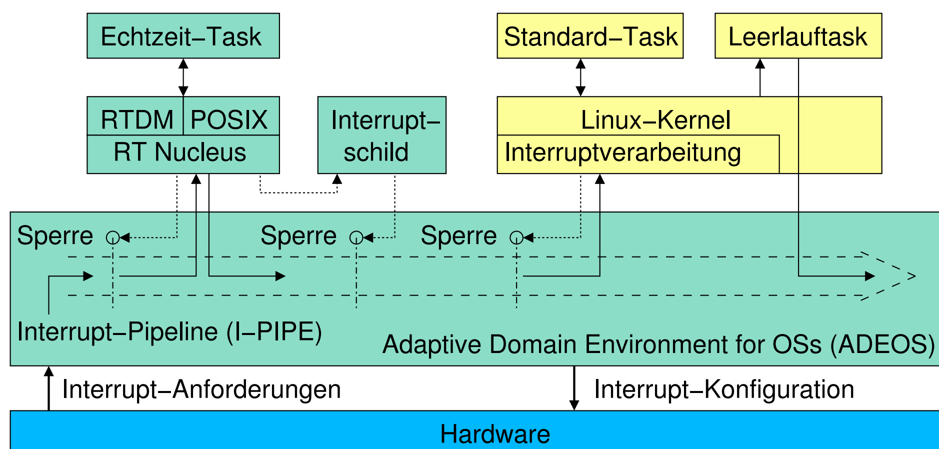


Abbildung 5.9: Zentrale Ereignisleitung durch den ADEOS Nanokernel

Alle Hard- und Softwareinterrupts werden in ADEOS, wie in Abb. 5.9 gezeigt, über eine zentrale Ereignisleitung (*Event/Interrupt-Pipeline*, *I-Pipe*) geführt und den beteiligten Betriebssystemen nacheinander zur Bearbeitung überlassen. Die Betriebssysteme werden hier als „Domänen“ bezeichnet und haben feste Prioritäten. Sie müssen minimal modifiziert werden, damit sie ihre Interruptbehandlung über ADEOS abwickeln und im Leerlauffall (als *Idle Task*) die Kontrolle an ADEOS zurückgeben, das dann die Kontrolle an Domänen niedrigerer Priorität abgibt und sie Interrupts verarbeiten lässt.

Insbesondere zum Sperren von Interrupts darf ein Glied der Ereignisleitung nicht mehr direkt auf die Hardware zugreifen, sondern muss dazu Methoden von ADEOS einsetzen. Dieses unterbricht dann die Ereignisleitung vor dem sperrenden Betriebssystem, sodass dieses und niederpriore Betriebssysteme keine Ereignisse mehr empfangen, höher priore jedoch weiterhin. Dadurch wird garantiert, dass die Interruptlatenzzeiten bei einem Echtzeitbetriebssystem mit höherer Priorität grundsätzlich und unabhängig von Linux deterministisch bleiben. Bei Bedarf können sich Treiber mit Bedarf an besonders kurzen Interruptlatenzzeiten als eigene Stufen an der Ereignisleitung registrieren [229].

Echtzeitnukleus mit POSIX-API

Als primäre Domäne registriert sich der generische Echtzeitkern (RT-Nucleus) von Xenomai an der Ereignisleitung. Da dieser nur die essentiellen Methoden zur Realisierung von Echtzeitfunktionalitäten enthält, ist auch an dieser Stelle der Echtzeitnachweis durch Quellcodeanalyse durchführbar.

Die verfügbaren Methoden dieses Echtzeitnukleus umfassen eine Task-, Interrupt- und Zeitverwaltung sowie Synchronisierungsprimitive. Aufbauend darauf sind verschiedene Emulationen (*Skins*) für Programmiermodelle bekannter Echtzeitsysteme realisiert. Die Softwarearchitektur der KogMo-RTDB setzte das standardisierte POSIX-API (*Portable Operating System Interface*) [97, 60] ein. Dieses ist auf den meisten UNIX-kompatiblen Betriebssystemen verfügbar, wie auch unter einem Standard-Linux.

Das gewählte Prozessmodell für zu integrierende Softwaremodule ist die Ausführung als Standard-Prozess. Dabei wird der von Xenomai unterstützte Start von Echtzeitapplikationen als Linux-Tasks auf Benutzerebene genutzt, die dann zu einem beliebigen Zeitpunkt durch entsprechende Systemaufrufe in einen hart echtzeitfähigen Kontext wechseln können, der nach der Nomenklatur von Xenomai *Primary*-Modus genannt wird. Es können jederzeit Systemfunktionen des GPOS aufgerufen werden, ohne die Stabilität des Systems durch Abstürze zu beeinträchtigen. Dabei wechseln die Tasks transparent in den sogenannten *Secondary*-Modus, in dem sie ihre harten Echtzeiteigenschaften verlieren, das GPOS durch einen Interrupt-Schild aber durch keine weiteren Interrupts gestört wird. Die Nutzung des *Secondary*-Modus ist insbesondere in der Funktionsentwicklungsphase sinnvoll, um beispielweise Bildschirmausgaben mittels `printf()` zu erzeugen, für den die Standard-E/A-Bibliothek den Systemaufruf `write(STDOUT,...)` nutzt. Auch in der Initialisierungsphase, in der noch keine harte Echtzeit notwendig ist, vermeidet die

Nutzbarkeit von Standard-Dateisystemoperationen aufwändige und unnötige Neuimplementierungen.

5.4.5 Interprozesskommunikation durch die KogMo-RTDB

Für die Kommunikation zwischen Echtzeitprozessen existieren zahlreiche Methoden. Die einfachste ist die Nutzung von Nachrichtenwarteschlangen (*Message Queues*) des RTOS. Unter den etablierten Linux-Echtzeiterweiterungen existieren dafür unterschiedliche Funktionen, je nachdem ob zwei Echtzeitprozesse, oder ein Echtzeitprozess mit einem nicht-Echtzeitprozess unter Linux kommunizieren. Durch die Verwendung dieser Methoden erfolgt jedoch eine frühe Festlegung auf das RTOS und die Aufteilung in Echtzeit- und nicht-Echtzeit-Prozesse.

Eine weitere Methode ist die Nutzung gemeinsamer Speicherbereiche (*Shared Memory*). Dabei besteht jedoch die Gefahr, dass durch den gleichzeitigen Zugriff mehrerer Prozesse inkonsistente Daten entstehen. Daher muss durch geeignete Synchronisierungsmethoden wie Semaphoren und Mutexes sichergestellt werden, dass immer nur ein Prozess jeweils ein Bündel an Änderungen vornimmt, und die betroffenen Datenstrukturen danach wieder in einem konsistenten Zustand hinterlässt. Das dazu notwendige Zugriffsprotokoll wird meist anwendungsspezifisch festgelegt und muss von allen beteiligten Softwaremodulen eingehalten werden. Weist nur ein einziges Modul einen Softwarefehler beim Datenzugriff auf, gefährdet es die Korrektheit des Gesamtsystems. Des weiteren sind die Synchronisierungsprimitive nur innerhalb der Gruppe der Echtzeitprozesse nutzbar, für die Synchronisierung zwischen Echtzeit- und nicht-Echtzeit-Prozessen müssen andere Methoden benutzt werden.

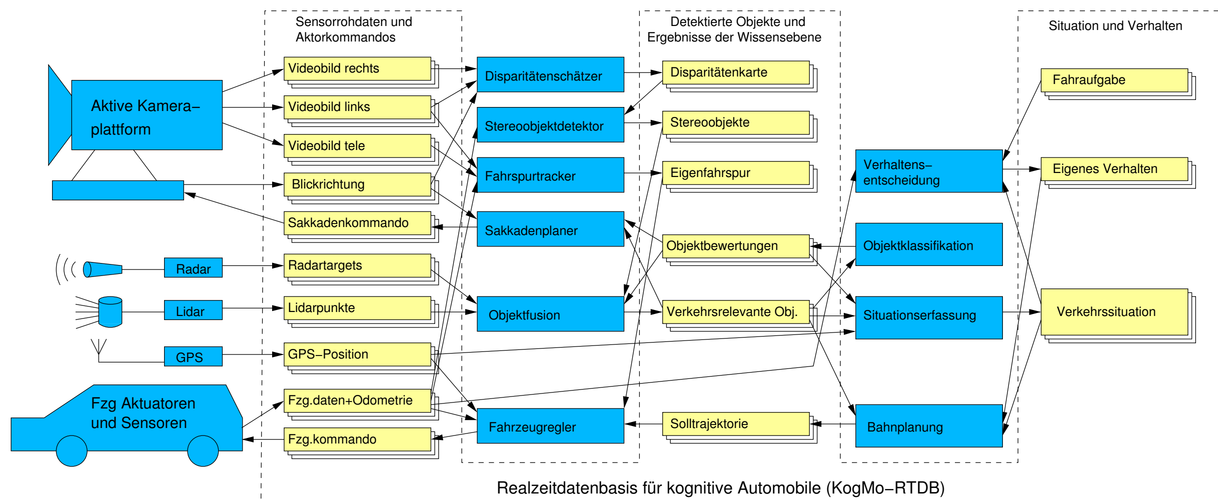


Abbildung 5.10: Kommunikation durch die KogMo-RTDB

Die in Kapitel 5.3 vorgestellte *Realzeitdatenbasis für kognitive Automobile* (KogMo-RTDB) bietet eine komfortablere Lösung zur Interprozesskommunikation. Abb. 5.10 verdeutlicht, wie die KogMo-RTDB auf allen Ebenen als Kommunikationsschicht zwischen

allen Modulen liegt, die untereinander Kommunikationsbeziehungen haben. Dadurch ist auch der Zugriff auf Informationen über Ebenen hinweg möglich.

Jeder Prozess kann nach Belieben Daten für alle anderen Prozesse veröffentlichen und deren Daten lesen. Die KogMo-RTDB stellt sicher, dass dabei keine inkonsistenten Datenobjekte weitergegeben werden. Wie in Abschnitt 5.3.5 dargelegt, wird nach Möglichkeit ein blockierungsfreies Schreib-/Leseprotokoll verwendet. Wird für ein Objekt ein öffentlicher Schreibzugriff gewünscht, werden automatisch die Synchronisationsprimitive des darunterliegenden Echtzeitbetriebssystems genutzt.

5.4.6 Hardwareschnittstellen

Auch bei der Auswahl der Schnittstellen zur Anbindung der Sensoren und Aktoren muss auf Determinismus geachtet werden. So sind parallele und serielle Punkt-zu-Punkt Verbindungen, solange sie z.B. über eine PCI-Schnittstellenkarte und nicht über einen USB-Konverter realisiert sind, aus Schnittstellensicht grundsätzlich hart echtzeitfähig. Für CAN gilt dies nur, wenn das maximale Aufkommen höherpriorer Nachrichten beschränkt ist.

Des weiteren findet man Schnittstellen, die in ihrer Standardform nicht echtzeitfähig sind, aufgrund ihrer weiten Verbreitung und des damit verbundenen geringen Preises für Echtzeitanwendungen interessant sind. So gilt ein normales Ethernet-Netz insbesondere aufgrund seiner kollisionsabhängigen zufallsbasierten Medienzugriffszeiten als nicht echtzeitfähig. Es existieren jedoch Ansätze, den Medienzugriff zu kontrollieren, um Echtzeitfähigkeit zu erreichen. Dies ist zum einen durch spezielle Hardware wie Realtime-Ethernet-Switches und Adapter möglich. Kostengünstiger sind jedoch reine Softwarelösungen, die Standardhardware verwenden [129]. Bei ausschließlicher Verwendung von Switches treten, abgesehen vom Überlastfall, keine Kollisionen auf. Bei Kenntnis deren Warteschlangenmechanismen und -längen [131] kann durch gezielte Senderatenbeschränkung (*Traffic Shaping*) in allen Stationen ebenfalls Echtzeitfähigkeit erreicht werden [130].

Möchte man ohne intelligente Komponenten wie Switches auskommen, bietet sich ein Zeitschlitzverfahren (*Time Domain Multiple Access*, TDMA) an. Für die genaue Zeitsteuerung ist eine genaue Uhrensynchronisation aller Stationen essentiell, RTnet [107, 109] setzt dafür ein Master-Slave-Verfahren ein. Alle Stationen müssen sich dabei an die zugewiesenen Zeitschlitze halten, der Medienzugriff wird durch die zusätzliche Schicht *RTmac* kontrolliert. Sonstiger Datenfluss muss durch den Echtzeit-Protokollstapel fließen [108]. Als Medium für RTnet kann auch IEEE 1394 *Firewire* dienen [233]. So lassen sich verteilte Echtzeitsysteme realisieren [226].

Schnittstellenanbindung

Nachdem das Signal eines Sensors über eine der genannten Schnittstellen eingetroffen ist, muss es, bevor die Rechenzeit der Software beginnt, über die rechnerinternen Bussysteme den Prozessor erreichen. Auch an dieser unscheinbaren Stelle können, wie in [200, 68, 197] gezeigt signifikante Latenzzeiten entstehen. Ein Beispiel für diese Bussysteme zeigt Abb. 3.2 in Kapitel 3.3.

Die zu berücksichtigenden Einflüsse sind die Zeit zum Auslösen einer Unterbrechungsanforderung (*Interrupt*), sowie die verlangsamte Übertragung einer Menge von Daten durch weitere quasi-gleichzeitige Übertragungen (*Timesharing*). In [196] wird diese Zugriffszeitverlängerung quantifiziert und Faktoren für den Worst-Case ermittelt.

In der entworfenen Architektur werden die Prozessoren untereinander mittels Hypertransport verbunden. Peripheriebusse werden über Hypertransport-Tunnel-Bausteine angebunden. Wie in Kapitel 4.3.1 gezeigt wird, werden die Schnittstellen so angebunden, dass die gegenseitigen Beeinflussungen minimal werden.

5.4.7 Schnittstellentreiber

Sobald das Hardwaresignal beispielsweise in Form einer Unterbrechungsanforderung beim Prozessor angekommen ist, ist es Aufgabe der Software, darauf angemessen zu reagieren. Die nachliegende Lösung, alle Daten direkt in der Unterbrechungsroutine (*Interrupt-Handler*) darauf zu verarbeiten, erreicht die niedrigsten Latenzzeiten, ist aber auch die unflexibelste Variante.

In modernen Betriebssystemen hat es sich durchgesetzt, Hardwaregerätetreiber und Applikationen durch standardisierte Schnittstellen voneinander zu trennen [203]. In einem solchen *Gerätetreibermodell* werden gleichartige Geräte in Klassen zusammengefasst, die gemeinsame Schnittstellen besitzen. So können Geräte gegeneinander ausgetauscht werden, ohne die Struktur der Applikationen modifizieren zu müssen.

Da ein Gerät immer nur von genau einem Gerätetreiber bedient wird, kann zudem der gleichzeitige Zugriff mehrerer Applikationen auf ein Gerät kontrolliert werden. Bei zeichenorientierten Geräten kann so ein exklusiver Zugriff gewährleistet werden, bei Netzwerkschnittstellen können die Datenpakete entsprechend verschachtelt (*Multiplex*) werden. Ohne zentrale Treiber wäre sonst ein eigenes Zugriffsprotokoll für jedes Gerät notwendig, an das sich alle nutzenden Prozesse halten müssen, damit keine Komplikationen entstehen.

Echtzeittreibermodell

In vielen Echtzeitsystemen werden echtzeitrelevante Schnittstellen immer noch von eigenständigen Treiber-Tasks bedient, die direkt auf die E/A-Peripherie zugreifen und ihre Daten z.B. mittels Warteschlangen (*Message Queues*) mit anderen Tasks austauschen.

Diese Lösung ist jedoch unportabel, da die Applikation sowohl an das konkrete Gerät als auch an das Hardwarelevel-API des RTOS gebunden ist.

Auf einem Zwei-Kern RTOS können die vorhandenen Treiber der GPOS nicht für zeitkritische Geräte verwendet werden, da hierbei die Peripheriezugriffe nicht mehr unter der Kontrolle des RTOS geschehen. Auch in Systemen wie Xenomai sind die Ausführungszeiten unter dem GPOS Linux nicht beschränkbar, wie in Kapitel 3.4.1 dargelegt wurde.

Einen Ausweg bietet ein Echtzeittreibermodell wie RTDM (*Real-Time Driver Model*) [106]. Durch die Organisation aller Echtzeittreiber in einer Treiberschicht mit einer einheitlichen Programmierschnittstelle werden Echtzeittreiber und -applikationen klar voneinander getrennt. Die dafür notwendige Portierung existierender Treiber wird durch die Anlehnung an das bewährte Linux-Treibermodell erleichtert.

Um eine unnötige Portierung zwischen den verschiedenen Echtzeiterweiterungen zu vermeiden, besteht das Ziel, die RTDM-Abstraktionsebene unter möglichst vielen Varianten, darunter auch Standard-Linux, verfügbar zu machen. RTDM-Treiber lassen sich bereits unter Xenomai und RTAI einsetzen.

In der vorliegenden Echtzeitarchitektur werden für Geräte, auf die eine deterministische Reaktion erfolgen soll, Echtzeittreiber eingesetzt, die das Echtzeittreibermodell des RTDM nutzen, darunter z.B. Treiber für die serielle und parallele Schnittstelle, sowie den CAN-Bus.

5.4.8 Echtzeitrechenprozesse

Auf oberster Ebene sind in Abb. 5.8 die eigentlichen Rechenprozesse angesiedelt. Ob ein Verarbeitungsmodul harten Echtzeitbedingungen genügt, hängt zum einen von den darin eingesetzten Algorithmen ab. In Kapitel 6.3.2 werden online- und offline-Methoden präsentiert, um die benötigte Gesamtrechenzeit (*Execution Time*) mithilfe der KogMo-RTDB messtechnisch zu ermitteln. Dabei ist jedoch nicht nicht garantiert, dass der *Worst-Case* in den resultierenden Messergebnissen enthalten ist. Daher sollte unterstützend eine Quellcodeanalyse durchgeführt werden, um den längsten Ausführungspfad zu identifizieren und dessen Laufzeit zu ermitteln. Ein graphenbasierter Ansatz zur statischen Analyse findet sich in [157], eine automatisierte Herangehensweise mit Laufzeitmessung zur dynamischen Analyse zeigt [159] auf. Eine aktuelle Übersicht über Methoden und Werkzeuge zur *Worst-Case Execution Time*-Analyse bietet [222].

Echtzeitgefährdende Operationen

Innerhalb der KogMo-RTDB Echtzeitarchitektur werden KogniMobil-Module als Tasks auf Benutzerebene ausgeführt. Sie werden dort als GPOS-Tasks gestartet und nach der Initialisierung mithilfe der Methoden von Xenomai auf die harte Echtzeitebene migriert und als RTOS-Tasks weitergeführt. Dabei bleiben sie auf Benutzerebene, eine Ausführung auf Betriebssystemebene ist nicht notwendig.

In dieser Umgebung haben die Module zur Laufzeit alle Freiheiten einer GPOS-Standard-Applikation. Sie können sie nutzen, um z.B. bei der Initialisierung nicht-echtzeitfähige Dateisystemzugriffe auf ihre Konfigurationsdaten machen. Im Echtzeitbetrieb darf jedoch nur noch eine Untermenge aller möglichen Operationen genutzt werden. Werden beispielsweise nicht-echtzeitfähige Systemaufrufe des GPOS genutzt, verliert das Modul seine harte Echtzeitfähigkeit und wird als GPOS-Task ausgeführt. Bei darauffolgenden RTDB- oder RTOS-Aufrufen wird es wieder zu einem RTOS-Task.

Eine absichtliche Nutzung von GPOS-Diensten liegt vor, wenn bei Stillstand des Fahrzeugs während der Fahrt gelernte Parameter in eine Datei gesichert werden. Die unbeabsichtigte Nutzung lässt sich durch die Migration zwischen den RTOS/GPOS-Domänen detektieren und darauf reagieren, beispielsweise mit einer Notbremsung des Fahrzeugs.

Im Folgenden wird ein Überblick über mögliche echtzeitschädliche Operationen gegeben. Softwaremodule sollten ihr Verhalten an den vorgeschlagenen Regeln ausrichten, um nicht ungewollt ihren Echtzeitkontext zu verlieren. Zu beachten ist, dass im Unterschied zu anderen Echtzeitlösungen die Missachtung keinen Systemabsturz nach sich zieht, sondern der Task lediglich als Standard-Task weitergeführt wird. Mit geeigneten Methoden kann dieser Übergang durch eine Behandlungsfunktion registriert werden.

Software-Ausnahmebehandlungen (Traps): Softwarefehler führen dazu, dass der aktuelle Rechenprozess unterbrochen wird und eine Routine zur Ausnahmebehandlung abgearbeitet wird. Mögliche Fehler sind eine Division durch Null oder eine Schutzverletzung (*Segmentation Fault*). Eine Schutzverletzung tritt insbesondere bei einem Zugriff auf eine ungültige virtuelle Speicheradresse auf. Um dabei auch nicht initialisierte Null-Zeiger zu detektieren, wird in den meisten GPOS dafür gesorgt, dass die Speicheradresse `Null` ebenfalls ungültig ist.

Da bei Softwarefehlern die Richtigkeit des Rechenergebnisses nicht gewährleistet ist, wird die gesetzte Deadline des fehlerhaften Rechenprozesses hinfällig. Stattdessen muss eine echtzeitfähige Fehlerbehandlung einspringen. An dieser Stelle ist zu bemerken, dass Softwarefehler im Kernel-Kontext zu schweren Systemabstürzen führen. Die Abfangbarkeit ist allein der Ausführung auf Benutzerebene mit dem damit verbundenen Speicherschutz geschuldet.

Systemaufrufe: Durch einen GPOS-Systemaufruf verlässt jeder Echtzeittask unmittelbar seinen Echtzeitkontext. Daher dürfen Systemaufrufe im echtzeitkritischen Hauptteil eines Rechenprozesses nicht benutzt werden. In der Initialisierungsphase und während eines Pausenzustands ist es dennoch sehr nützlich, Systemaufrufe z.B. für Dateizugriffe oder Benutzerinteraktion einzusetzen. Unter anderen Echtzeitbetriebssystemen, die nicht die hier vorstellte Architektur nutzen, ist dies oftmals nicht so einfach zu realisieren und benötigt zusätzliche Vermittlungsmodule.

Dynamische Speicherverwaltung: Die Speicherverwaltung besitzt einige Mechanismen, die potentiell realzeitschädlich sein können. Dies ist zum einen der eingesetzte Allokationsalgorithmus an sich. Wie schon in Kapitel 3.5.1 und in [224, 161] gezeigt, sind viele Algorithmen für bestmögliche durchschnittliche Zuteilungszeiten opti-

miert. Dabei werden sporadisch auftretende maximale Zeiten toleriert. Für Echtzeitsysteme kommen daher nur spezielle Echtzeitallokatoren wie TLSF [134] in Frage. Die KogMo-RTDB alloziert sich daher in der Initialisierungsphase beim GPOS nicht-echtzeitfähig einen bestimmbaren Speicherblock, der dann im Betrieb durch den TLFS-Algorithmus verwaltet wird. Daher ist das Anlegen neuer Objekte in der RTDB mittels `RTDBInsert()` hart echtzeitfähig.

Das Allozieren von zusätzlichem Speicher unter Verwendung der Funktion `malloc()` der C-Laufzeitbibliothek (*libc*) von GNU/Linux, ist, wie in Kapitel 3.5.3 gezeigt, nicht echtzeitfähig. Zum einen basiert der Allokator auf dem *dlmalloc*-Algorithmus [120], der insbesondere durch sein Aufschieben der Wiedervereinigung freier Speicherblöcke (*Deferred Coalescing*), lange Spitzenlaufzeiten verursachen kann. Zum anderen wird der Haldenspeicher, aus dem der Speicher vergeben wird, bei Bedarf durch den Linux-Systemaufruf `brk()` vergrößert. Dabei wird der Echtzeitkontext grundsätzlich verlassen.

Der Standardoperator `new[]` der C++ Laufzeitbibliothek benutzt in der aktuellen Version (libstdc++-v3/gcc-3.4.6) ebenfalls die Funktion `malloc()` der C-Laufzeitbibliothek. Werden die genannten Funktionalitäten benötigt, muss `malloc()` ersetzt bzw. `new[]` überladen werden. Der maximal benötigte Speicher muss dann bei Programmstart alloziert und zur Laufzeit durch einen echtzeitfähigen Allokator vergeben werden.

Speicherseitenmanagement: Linux besitzt wie die meisten modernen Betriebssysteme ein bedarfsgesteuertes Speicherseitenmanagement zur Verwaltung des logischen und virtuellen Speichers ein. Sobald ein Prozess auf eine Speicherseite zugreift, die sich nicht im physikalischen Speicher befindet, oder für die er die falschen Rechte hat, wird der Prozess unterbrochen, ein Seitenfehler ausgelöst, die Speicherseite verfügbar gemacht und der Prozess wieder fortgesetzt. Wenn der physikalische Speicher knapp wird, werden selten benutzte Speicherseiten auf die Festplatte ausgelagert.

Kern dieses Mechanismus ist die Bearbeitungsroutine für Seitenfehler (*Page Fault*) des GPOS. Daher führt das Auslösen eines Seitenfehlers unmittelbar zum Verlust der Echtzeiteigenschaften. So wird empfohlen, in Echtzeitsystemen keine Auslagerungsdatei (*Swap Space*) zu verwenden, da das Wiedereinlagern einer Speicherseite einen langwierigen Festplattenzugriff erfordert. Im gemischten Betrieb kann es dennoch Gründe dafür geben, z.B. nicht-Echtzeitprozesse zur Datenanalyse.

Sogar ohne Auslagerungsdatei kann es zu Seitenfehlern kommen. Zum einen, weil ein GPOS bei Programmstart nur einen Teil des ausführbaren Codes und der Bibliotheken in den physikalischen Speicher lädt. Erreicht der Programmablauf ein nicht vorhandenes Codesegment, wird dieses durch den Speicherseitenmanager automatisch nachgeladen. Zum anderen wird physikalischer Speicher einem Prozess erst zugewiesen, wenn der allozierte logische Speicher auch beschrieben wird. Dazu zeigen die Adressen des allozierten aber noch unbenutzten Speichers auf eine spezielle Seite (`ZERO_PAGE`), die Nullen enthält und mit Kopieren-bei-Schreibzugriff (*copy-on-write*, COW) markiert ist. Bei dem ersten Schreibzugriff auf eine solche

Speicherseite wird sie im physikalischen Speicher alloziert und die Null-Seite dupliziert. Daher genügt es nicht, nur die Allokation in die Initialisierung zu verschieben, es sind auch Schreibzugriffe auf alle allozierten Speicherbereiche notwendig.

Um unvorhersehbare Unterbrechungen durch das automatische Speicherseitenmanagement auszuschließen, bieten viele Betriebssysteme eine Funktion, um Speicherseiten gegen Auslagen zu sperren. Unter Linux und POSIX-kompatiblen Systemen dient dazu der Systemaufruf `mlockall()`. Dabei werden zudem alle noch nicht eingelagerten Text- und Datensegmente von Festplatte gelesen und für alle allozierten Speicherbereiche physikalischer Speicher bereitgestellt. Die KogMo-RTDB führt für jeden sie nutzenden Prozess die notwendigen Schritte zum Sperren des Speichers automatisch in der Initialisierungsphase durch.

Bibliotheksfunktionen: Besondere Vorsicht ist bei der Nutzung fremder Bibliotheken geboten. Um die harte Echtzeit eines Prozesses nicht zu gefährden, muss jede genutzte Bibliotheksfunktion den hier aufgestellten Regeln genügen. So muss die längst mögliche Ausführungszeit berücksichtigt werden, die insbesondere bei mathematischen Bibliotheken in Spezialfällen deutlich über der Durchschnittszeit liegen kann.

Eingesetzte Bibliotheken dürfen keine Systemaufrufe vornehmen, wie z.B. eine Bildschirmausgabe im Fehlerfall. Oft geschieht ein Systemaufruf indirekt durch eine Speicherallokation. In den folgenden Beispielen wurde die C-Laufzeitbibliothek (*glibc-2.6.1*) von GNU/Linux untersucht:

- `strncmp()` vergleicht zwei Zeichenketten. Die Laufzeit ist von der Länge der kürzeren Zeichenkette abhängig und zusätzlich durch einen Aufrufparameter `n` beschränkbar. Diese Funktion gefährdet damit eine harte Echtzeitfähigkeit nicht.
- `strdup()` dupliziert eine Zeichenkette. Dafür wird Speicher mittels `malloc()` alloziert und so die Echtzeitfähigkeit gefährdet. Für eine echtzeitfähige Lösung müsste eine zweite ausreichend große Zeichenkette statisch alloziert und mittels `strncpy()` kopiert werden. Die in diesem Fall notwendige Festlegung einer Maximalgröße trägt zusätzlich zu einer deterministischen Laufzeit bei.
- `getenv()` liefert einen Zeiger auf eine Umgebungsvariable. Dazu wird wiederholt `strncmp()` genutzt. Wenn die maximale Anzahl von Umgebungsvariablen bekannt ist, gefährdet die Funktion die harte Echtzeitfähigkeit nicht.
- `setenv()` setzt eine Umgebungsvariable. Da dazu `realloc()` eingesetzt wird, um den Umgebungsspeicher zu vergrößern, darf die Funktion nicht in harten Echtzeitprozessen genutzt werden.

Diese Beispiele zeigen, dass eine mögliche Echtzeitgefährdung von Funktion zu Funktion verschieden ist und jeder Einzelfall geprüft werden muss. Da für jede neue Version einer Bibliothek diese Prüfung zu wiederholen ist, wird in Abschnitt 5.6 ein *Referenzsystem* empfohlen, das alle Softwareversionen standardisiert.

5.5 Kooperation verschieden harter Echtzeitprozesse

Viele technische Systeme verlangen nur von einem Teil der Rechenprozesse unbedingt harte Echtzeitfähigkeit. Wie im vorangegangenen Kapitel gezeigt, erfordert ein hart echtzeitfähiges Design einen erheblichen Zusatzaufwand, verglichen mit „nur“ weicher oder keiner Echtzeit. Eine oft gewählte Lösung ist die Partitionierung in ein kritisches, hart echtzeitfähiges und ein unkritisches Teilsystem. Um dabei möglichst wenige Komponenten in den Echtzeitnachweis einbeziehen zu müssen, wird die Trennung sowohl auf Hard- und Softwareebene durchgeführt.

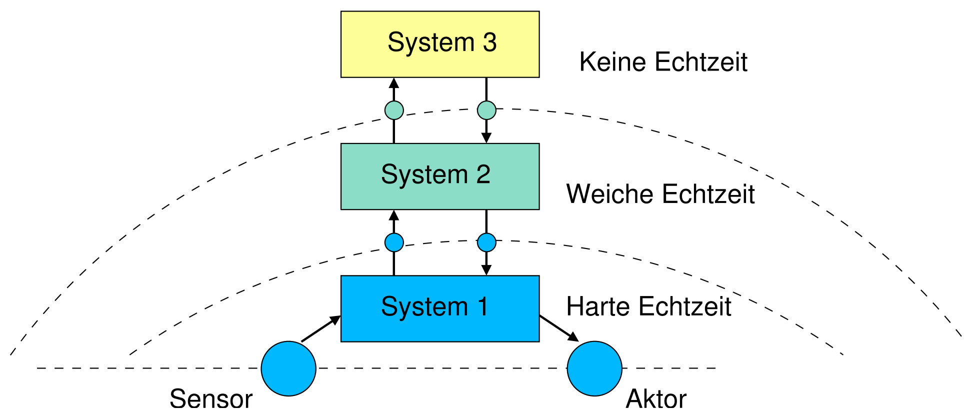


Abbildung 5.11: Datenfluss zwischen unterschiedlich harten Echtzeitsystemen

Den Datenfluss in einer solchen Aufteilung, wie sie gerne in Systemen ohne KogMo-RTDB gewählt wird, zeigt Abb. 5.11. Die kaskadierten Funktionen oder Regelkreise haben von harter Echtzeit unten, über weiche Echtzeit bis keine Echtzeit nach oben hin abnehmende Echtzeitanforderungen. Oft gewählte Realisierungen sehen folgendermaßen aus:

- Für den hart (ggf. auch weich) echtzeitfähigen Teil wird ein kleines eingebettetes System verwendet. Ein Prozessor ohne Caches und Pipelines bietet gerade ausreichend Rechenleistung für einfache Berechnungen, liefert diese aber mit deterministischen Laufzeiten. Für die Kommunikation werden CAN und serielle Schnittstellen eingesetzt, die sich durch kurze Latenzzeiten auszeichnen, jedoch eine geringe Bandbreite aufweisen.

- Für den nicht echtzeitfähigen Teil wird ein Standard-PC-System eingesetzt. Dieses bietet eine hohe durchschnittliche Rechenleistung bei sporadisch auftretenden Blockierungen beispielsweise durch andere Hintergrundprozesse oder Betriebssystemtreiber. Zur Kommunikation wird Standard-Ethernet eingesetzt, das zwar eine hohe Bandbreite bietet, aufgrund des in Kapitel 5.4.6 beschriebenen Medienzugriffs aber keine maximalen Übertragungszeiten garantieren kann.

Probleme bei dieser Vorgehensweise entstehen gewöhnlich an der Schnittstelle zwischen den Teilsystemen. Die verbindenden Schnittstellen müssen in ihrer Echtzeitfähigkeit dem echtzeitfähigeren System entsprechen, denn wie schon in Kapitel 5.4 dargelegt, verliert ei-

ne Kette von Funktionen ihre Echtzeitfähigkeit, wenn sie ein nicht-echtzeitfähiges „schwaches“ Glied enthält.

Die obige Aufteilung findet sich zumeist in der Aufgabenverteilung an einzelne Entwickler oder ganze Abteilungen wieder, was die Trennung der Teilsysteme vertieft. Aus Sicht des Echtzeitdesigners wird die Kommunikation mit weniger echtzeitfähigen Teilsystemen oftmals als unwichtig oder sogar lästig erachtet. Zudem kostet den unterlagerten Regelkreis das explizite Aus- und Einkoppeln von Daten zusätzliche Rechenzeit. So werden über die Schnittstellen nur die nötigsten Informationen weitergegeben, was spätere Erweiterungen erschwert. Dadurch wird ferner die Kooperation der einzelnen Softwaremodule auf das absolute Minimum beschränkt.

5.5.1 Harte Echtzeit und Kooperation im kognitiven Automobil

In einem kognitiven Automobil besteht sowohl der Bedarf an harter Echtzeit als auch an enger Kooperation zwischen den Modulen. Dies erzeugt die gegensätzliche Anforderung, die Teilsysteme zugunsten der Echtzeit zu entkoppeln und zugleich zugunsten der Kooperation eng miteinander zu verbinden. Der funktionalen Architektur aus Kapitel 3.6.3 ist zu entnehmen, dass der Regelkreis der Wahrnehmung auf mehreren Ebenen geschlossen wird.

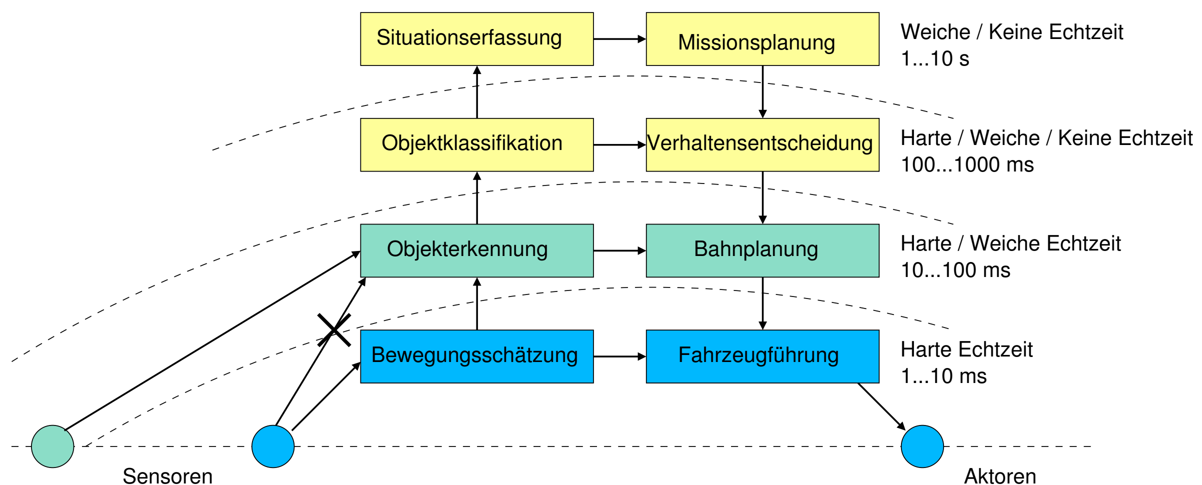


Abbildung 5.12: Unflexible Partitionierung der Echtzeitebenen im Kognitiven Automobil

Abb. 5.12 zeigt die nach oben abnehmenden Echtzeitanforderungen mit gleichzeitig größer werdenden Zykluszeiten der beteiligten Rechenprozesse. So ist auf unterster Ebene die Fahrzeugführung angesiedelt, die für ihre unmittelbare Quer- und Längsführung harte Echtzeitanforderungen stellt, und durch kurze Zykluszeiten im Bereich von beispielsweise $1\ldots10$ ms charakterisiert ist. Bei einer Verletzung der Zeitbedingungen könnte das Fahrzeug leicht von der Fahrbahn abkommen. In Gegensatz dazu ist auf der Planungsebene mit Zeiten von $1\ldots10$ s und mehr eine Deadline-Verletzung, die z.B. zum Verpassen einer

Autobahnausfahrt führt, ärgerlich, aber i.d.R. nicht lebensgefährlich (außer für einen Notfallpatienten in einem Krankenwagen). Die Einteilung zeigt die Mindestanforderungen an Echtzeitfähigkeit. Selbstverständlich sind Fahrzeuge realisierbar, deren Echtzeitfähigkeit bis zur Planungsebene reicht.

In jedem Fall muss ein sicheres Verhalten auch dann gewährleistet sein, wenn höhere Entscheidungsebenen nicht oder nicht rechtzeitig reagieren. Jedoch steht bei der Entwicklung in einem Forschungsprojekt nicht bei allen kognitiven Funktionen von Anfang an deren Echtzeitfähigkeit im Vordergrund. In der Erprobung kognitiver Wahrnehmungs- und Entscheidungsfunktionen auf oberster Ebene ist an erster Stelle das Ergebnis interessant. Dafür wiederum sind möglichst viele Eingangsdaten hilfreich. Da die Anzahl an verfügbaren Sensoren meist beschränkt ist, möchte man auch auf die Daten von Sensoren zugreifen, die von den unterlagerten hart echtzeitfähigen Sicherheitsfunktionen benutzt werden.

Die Messdaten eines abstandsgebenden Sensors wie RADAR oder LIDAR braucht beispielsweise sowohl eine automatische Notbremsfunktion, die einen Auffahrunfall verhindern muss, als auch eine Verifikationseinheit von videobasierter Fremdfahrzeugerkennung. Eine direkte Kommunikation einer nicht-hart echtzeitfähigen Funktion mit einem Sensor, der von einer hart echtzeitfähigen Funktion beansprucht wird, ist, wie in Abb. 5.12 angedeutet, nicht erlaubt. Eine zugegebenermaßen unflexible Lösung wäre es, die Sensordaten einfach bis zur betreffenden Ebene weiterzureichen.

Als weiteres Problem kommt hinzu, dass bei der Einteilung in Teilsysteme für harte, weiche und keine Echtzeit, zu einem sehr frühen Designzeitpunkt Partitionierungsentscheidungen gefällt werden müssen, die später nur mehr schwer geändert werden können. Ein Verschieben von Prozessen zwischen Teilsystemen bedeutet meistens eine Portierung von Software und ist mit zumutbarem Aufwand oft nicht mehr möglich.

5.5.2 KogMo-RTDB als Bindeglied zwischen Echtzeitebenen

Die KogMo-RTDB bietet die Methoden für eine flexiblere Lösung des gegebenen Problems. In einer KogMo-RTDB-basierten Architektur fungiert die KogMo-RTDB als Schnittstelle zwischen allen Rechenprozessen. Alle Prozesse werden auf einem Rechnersystem ausgeführt, dessen Hard- und Softwarebasis echtzeitfähig ist. Die KogMo-RTDB mit ihren Schnittstellen ist ebenfalls hart echtzeitfähig. Die Echtzeitfähigkeit einzelner Rechenprozesse hängt damit nur noch vom Determinismus der eingesetzten Algorithmen ab.

Somit ist in einer KogMo-RTDB-basierten Architektur keine frühe Festlegung in unterschiedlich echtzeitfähige Systeme notwendig. Sobald alle an einer Funktion beteiligten Prozesse echtzeitfähig sind, gilt dies für die gesamte Funktion. Es ist kein Verschieben von Prozessen zwischen Teilsystemen und kein Redesign notwendig.

Abb. 5.13 zeigt, wie die Rechenprozesse eines kognitiven Automobils über die KogMo-RTDB gekoppelt werden. Die KogMo-RTDB ist hart echtzeitfähig, für einige Rechenprozesse ist deren Echtzeitebene noch nicht festgelegt, wie dies zu Entwicklungsbeginn

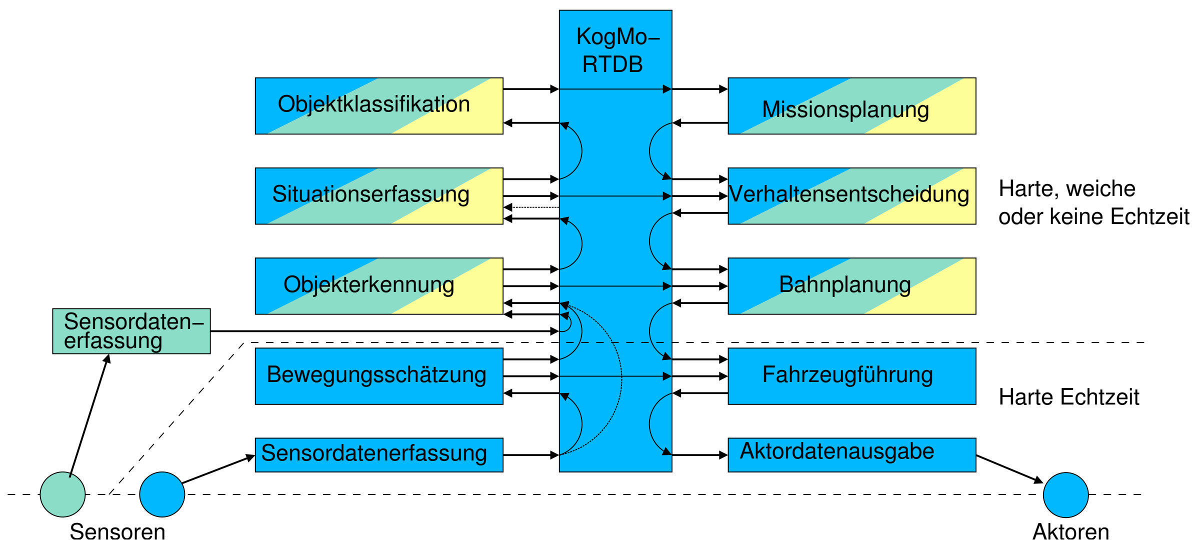


Abbildung 5.13: Koppelung von Rechenprozessen mit unterschiedlichen Echtzeitanforderungen über die KogMo-RTDB

denkbar ist. Echtzeit wird hier zu einer reinen Frage der Software auf Anwendungsebene. Die Grundlagen an echtzeitfähiger Hardware und Basissoftware sind durch die hier vorgestellte Architektur geschaffen.

Bei einer konsequenten Verwendung der KogMo-RTDB zur Interprozesskommunikation lassen sich Informationen aus einer hart echtzeitfähigen Regelschleife auskoppeln, ohne dessen Zeitverhalten zu verändern. Da eine strikte Trennung der Rechenprozesse auf verschiedene Rechnersysteme nicht mehr notwendig ist, wird die enge Vernetzung der Prozesse gefördert. So können insbesondere harte Brüche zwischen Wahrnehmungs- und Handlungsebenen vermieden werden.

5.5.3 Anforderungen an echtzeitfähigen Datenaustausch

Alle mit der KogMo-RTDB verbundenen Prozesse verwenden dieselbe Programmierschnittstelle, die auch für alle verschieden harten Echtzeitprozesse identisch ist. Die beteiligten Prozesse müssen beim Austausch von Daten keine Regeln einhalten und keine Rücksicht auf hart echtzeitfähige Prozesse nehmen. Das eingesetzte Echtzeitbetriebssystem setzt nur die Einhaltung von Prioritäten durch. Das reibungslose Zusammenwirken zu ermöglichen ist allein Aufgabe der KogMo-RTDB.

Daher ist es essentiell, sicherzustellen, dass es zu keinen unbeabsichtigten Blockierungen und einer damit einhergehenden Prioritätsinversion kommen kann. Für den gleichzeitigen Schreib- und Lesezugriff auf dasselbe RTDB-Objekt wurde in Kapitel 5.3.5 bereits ein blockierungsfreies Schreib-/Leseprotokoll vorgestellt und dessen Zeitverhalten untersucht. An Stellen notwendiger Blockierungen müssen deren Auswirkungen genau unter-

sucht und begrenzt werden. Nur so ist die Echtzeitfähigkeit aller beteiligten Prozesse jederzeit gewährleistet.

5.5.4 Erlaubte deterministische Blockierungen

Es existieren jedoch Anwendungsfälle, in denen gleichzeitig dieselben Daten modifizierbar sein müssen. Dafür wären zwar ebenfalls blockierungsfreie Schreib-Schreibprotokolle ähnlich den in Kapitel 2.4 vorgestellten Lösungen denkbar. Diese erfordern jedoch einen erheblichen Mehraufwand an Ressourcen und limitieren zudem die Maximalanzahl möglicher Schreibprozesse. Der bisherige Speicheraufwand des entwickelten blockierungsfreien Schreib-Leseprotokolls kann, wie in Kapitel 5.3.4 gezeigt, durch seinen Zusatznutzen als Historienspeicher gerechtfertigt werden.

Mögliche Blockierungen innerhalb eines Programms stellen eine Gefahr für dessen harte Echtzeitfähigkeit dar. Sie machen sie genau dann zunichte, wenn die längste Blockierungsdauer nicht auf die maximal zulässige Zeit beschränkt werden kann. Wie in den Kapiteln 3.4 und 5.4.4 dargelegt, kann diese Zusicherung nur für unter einem Realzeitbetriebssystem laufende Tasks gegeben werden, deren Algorithmen deterministisch sind.

In der KogMo-RTDB werden daher nur unter bestimmten Voraussetzungen Blockierungen zugelassen. Eine Blockierung tritt beispielsweise auf, wenn ein Prozess ein Objekt schreiben will, das gerade von einem anderen, evtl. sogar niederprioren Prozess geschrieben wird, und dauert bis zur Beendigung des konkurrierenden Zugriffs. Zur Garantie der Einmaligkeit einer OID wird ebenfalls eine blockierende Sperre eingesetzt.

Für die Zulässigkeit einer Blockierung müssen folgende Bedingungen erfüllt sein:

1. Der Task muss sich während der gesamten Blockierungsdauer durchgehend im Echtzeitkontext befinden, der dem *Primary Mode* von Xenomai in der implementierten Architektur entspricht. Es muss sichergestellt sein, dass dieser auch im Fehlerfall bis zur Aufhebung der Blockierung nicht verlassen wird. Dies verbietet insbesondere eine Fehlerausgabe (`printf()`), die durch den damit ausgelösten Linux-Systemaufruf, wie in Kapitel 3.4.2 und 5.4.8 verdeutlicht, einen Wechsel in den GPOS Kontext (*Secondary Mode*) nach sich ziehen würde. Die Echtzeitmethoden der KogMo-RTDB halten diese Bedingung stets ein.

2. Für Bestimmung der WCET ist der längste Programmpfad während einer Blockierung maßgeblich. Dies setzt voraus, dass dieser bekannt ist. Innerhalb der KogMo-RTDB-Operationen ist dies gegeben, auch die maximale Anzahl von Schleifendurchläufen z.B. bei einer Kopieroperation ist durch die a priori anzugebende maximale Objektgröße $n_{bytes,max}$ für jedes Objekt beschränkt.

3. Für die Dauer einer Datenbankoperation, die bestimmte andere Operationen blockiert, darf der Programmpfad den Programmcode der KogMo-RTDB nicht verlassen. Nur dann ist sichergestellt, das die maximalen Ausführungszeiten in je-

dem Fall nur von der KogMo-RTDB bestimmt werden, und nicht vom Verhalten der nutzenden Anwendungsprozesse abhängen.

Die genannten Blockierungen sind damit zulässig. Ob sie wirklich auftreten, ist durch die Konfiguration jedes einzelnen KogMo-RTDB-Objekts bestimmbar und wird in den folgenden Kapiteln genauer untersucht.

5.5.5 Unzulässige blockierende Funktionalitäten

Sind für die Dauer einer Blockierung die oben genannten Bedingungen nicht erfüllt, ist diese im harten Echtzeitbetrieb nicht zulässig. Dies führt dazu, dass die KogMo-RTDB manche, aus Anwendungssicht wünschenswerte, Operationen nicht anbieten kann:

Sperren von Objekten: Eine denkbare Funktion wäre das zeitweise Sperren einer Auswahl von Objekten. Dazu würde eine Applikation eine Menge von Objekten (*lock set*) $\mathcal{S}_{lock} = \{\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_n\}$ durch eine DB-Methode atomar sperren und später wieder freigeben. Dies ist notwendig, wenn eine Kombination von Objekten $\mathcal{S}_{inconsitent} = \{\mathcal{D}_1(t_1), \mathcal{D}_2(t_2), ...\}$ zu zwei verschiedenen Zeitpunkten t_1 und t_2 möglich ist, die einen inkonsistenten Zustand darstellt. Durch eine Sperrung von $\mathcal{S}_{lock}$ gegen fremdes Lesen und Schreiben könnte dieser inkonsistente Zustand während der Aktualisierung vermieden werden. Würde eine solche Sperre zugelassen, ist es wahrscheinlich, dass der sperrende Prozess für die Dauer der Sperre weitere Abfragen macht und eigene Berechnungen durchführt.

Durch die genannte Sperrung werden jedoch Schreib- und sogar Lesezugriffe anderer Prozesse blockiert. Deren Dauer wäre nun nicht mehr nur von der KogMo-RTDB abhängig, sondern von der Ausführungszeit des Algorithmus eines anderen Prozesses.

Daraus folgt, dass nun das Verhalten *aller* mit der KogMo-RTDB verbundener Prozesse in den Echtzeitnachweis einbezogen werden müsste. Hier entsteht ein starker Interessenkonflikt, da dies beispielsweise im kognitiven Automobil für viele wissensverarbeitende Prozesse einen unzumutbaren Aufwand bedeuten würde. Zudem lässt sich ohne eine vollständige Programmanalyse nicht ausschließen, dass ein blockierender Prozess echtzeitgefährdende Operationen wie in Kapitel 5.4.8 benutzt, und damit die Echtzeitfähigkeit des Gesamtsystems in Gefahr bringt.

Eine Lösung dieses Problems ist es, aus den obigen Objekten ein neues Objekt $\mathcal{D}_{sum} = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \mathcal{D}_n$ zu bilden, das mit den gegebenen Methoden der KogMo-RTDB atomar und unter harter Echtzeit aktualisiert werden kann.

Ein weiterer Ansatz ist es, die Aktualisierungsreihenfolge der Objekte $\mathcal{D}_{first}, ...,$ $\mathcal{D}_i, ..., \mathcal{D}_{last}$ festzulegen. Aus Sicht der KogMo-RTDB ist dies eine Konvention, die aber durch eine entsprechende Zusatzbibliothek transparent realisiert werden kann. Lesende Prozesse können auf die Aktualisierung von $\mathcal{D}_{last}$ warten und die restlichen

Objekte $\mathcal{D}_i$ anhand deren Aktualisierungszeitstempel $\mathcal{D}_i(t_{committed,i} = t_{committed,last})$ aus der Historie auslesen.

Objekte beliebiger Größe: Praktisch wären Objekte, die in ihrer Größe beliebig und ohne obere Grenze wachsen können. Dies würde für alle lesenden Prozesse eine unbegrenzte dynamische Speicheranforderung zur Laufzeit bedeuten, die zum einen, wie in Kapitel 5.4.8 gezeigt, eine Gefahr für harte Realzeit darstellt, zum anderen durch eine fehlende obere Grenze keine Beschränkung der Laufzeit z.B. einer Kopierroutine zulässt.

In der KogMo-RTDB muss daher für jedes Objekt beim Eintragen eine maximale Grösse $n_{bytes,max}$ in den Objektmetadaten (siehe Kapitel 5.3.1) spezifiziert werden. Der benötigte Speicher wird von der Datenbank mit den Methoden aus Kapitel 5.3.7 hart echtzeitfähig alloziert.

Bei jeder Aktualisierung wird die aktuelle Größe durch den Parameter $n_{bytes,committed}$ ($0 < n_{bytes,committed} \leq n_{bytes,max}$) angegeben. Dadurch wird die Dauer von Kopieroperationen nicht unnötig verlängert und beim Aufzeichnen (siehe Kapitel 5.7.2) kein Speicherplatz verschwendet. Für WCET-Berechnungen ist jedoch stets nur $n_{bytes,max}$ relevant.

Auslösung von automatischen Aktionen: Viele SQL-Datenbanken [36] bieten Applikationen die Möglichkeit, Anweisungsfolgen innerhalb der Datenbanken abzulegen (*stored procedures*) und diese dann automatisch beim Eintreten vordefinierter Bedingungen ausführen zu lassen (*Trigger*). Auch aktive Realzeitdatenbanken (*Active Real-Time Databases*, *ARTDBs*) bieten diese Möglichkeit [146].

Für Systeme aus konkurrierenden Prozessen mit unterschiedlich harten Echtzeitanforderungen stellt sich jedoch die Frage, in welchem Kontext solche automatischen Prozeduren ausgeführt werden sollen. Um sichere Aussagen über das Zeitverhalten eines Prozesses machen zu können, darf es nicht erlaubt werden, dass durch fremde Software zusätzlicher Programmcode in dessen Kontext ausgeführt wird. Um ihre Funktion zu erfüllen, müssten die automatischen Anweisungen jedoch mit ausreichend hoher Priorität unmittelbar ausgeführt werden.

Um unnötige Komplexität zu vermeiden, bietet die KogMo-RTDB keine Methoden, um innerhalb der Datenbank *stored procedures* zu bestimmten *trigger*-Punkten auszuführen, sie verhält sich damit rein passiv. Es besteht jedoch die in Abschnitt 6.3 beschriebene Möglichkeit, ähnliche Funktionalität über normale Applikationen nachzubilden, die auf die entsprechenden *trigger*-Bedingungen warten und bei Eintreten ihre Anweisungen ausführen. Deren Priorisierung kann so individuell über den Scheduler des Betriebssystems geschehen. Aus der Priorität ergibt sich unmittelbar, ob deren Laufzeit in den Echtzeitnachweis eines anderen Moduls miteinbezogen werden muss.

Die obigen Beispiele zeigen, warum es in der Architektur der KogMo-RTDB nicht zulässig ist, die Kontrolle an ein Anwendungsprogramm abzugeben, während bestimmte Elemente

der Datenbank gesperrt sind. Nur so kann sichergestellt werden, dass alle Methoden stets deterministisch sind.

5.5.6 Synchronisierungsmethoden

In allen Daten verwaltenden Systemen, so auch in Datenbanken, besteht die Gefahr von inkonsistenten Daten. Diese entstehen, sobald es möglich ist, eine weitere Operation aufzurufen, solange eine vorhergehende noch nicht abgeschlossen ist. In modernen präemptiven Multitaskingbetriebssystemen ist dies schwer zu verhindern, sobald mindestens zwei Tasks auf dieselben Daten zugreifen. Kommen Mehrkern-Multiprozessorarchitekturen zum Einsatz, besteht zudem die Gefahr des simultanen Zugriffs auf Datenbestände.

Um solche Inkonsistenzen zu verhindern, existieren verschiedene Ansätze. Für die vorgestellte Systemarchitektur ist insbesondere die folgende Auswahl relevant:

Deaktivierung von Interrupts: Dies ist auf Einkern-Einprozessorsystemen die einfachste Art, zu vermeiden, dass ein kritischer Pfad unterbrochen wird. Jedoch verlängert sich die maximale Interruptlatenz aller Tasks sowie die Ausführungszeit aller höherprioren Tasks um die maximale Dauer der Interruptdeaktivierung.

Auf Mehrkern- oder Mehrprozessorsystemen muss zusätzlich verhindert werden, dass andere Recheneinheiten den kritischen Programmabschnitt betreten können, der einen gemeinsamen Datenbereich modifiziert. Dazu dienen beispielsweise aktive Warteschleifen (*Spinlocks*) [19]. Für die Dauer der Blockierung sind im ungünstigsten Fall alle anderen Recheneinheiten blockiert, sodass die effektiv nutzbare Rechenleistung sinkt.

Die Deaktivierung von Interrupts skaliert sehr schlecht mit der Anzahl von Recheneinheiten und muss in einem größeren System, wie dem Rechensystem des kognitiven Automobils aus Kapitel 4.3, vermieden werden. Zudem ist die Interruptkontrolle nicht auf Benutzerebene erlaubt, sondern benötigt die Privilegien der Betriebssystemebene.

Atomare Operationen: Darunter versteht man aus Softwaresicht Operationen, die von der Hardware ununterbrechbar und in Mehrkernsystemen auch ohne den Einfluss fremder Recheneinheiten durchgeführt werden. Zu dieser Klasse gehören in der IA32-Architektur [98] die Assemblerbefehle `decl` (Dekrementieren einer Variablen) und `cmpxchg` (Vergleich und bedingter Wertetausch), denen dafür das `lock`-Prefix vorangestellt werden muss. So wird durch geeignete Cache-Konfiguration und Signalisierung zwischen den Prozessoren verhindert, dass andere Prozessoren währenddessen die betroffene Speicherstelle modifizieren können. Auf Opteron-Prozessoren ist dies ebenfalls implementiert [1], dort wird über den Hypertransport-Bus ein *Atomic Read-Modify-Write Request* gemeldet [209].

Im Kern von Betriebssystemen wie Linux [126] werden mithilfe dieser atomaren Assemblerbefehle *Spinlocks* und *Mutexes* realisiert. Ist garantiert, dass nur ein Prozess

eine Variable modifiziert und hat die Variable nur die Größe eines prozessoreigenen Elementartyps (z.B. Ganzzahl), ist die Modifikation dieser Variablen ebenfalls atomar.

Mutexes: Mutexes und Semaphoren sind ein von Dijkstra [45] vorgeschlagener Mechanismus zur Prozesssynchronisation. Eine Implementierung nutzt beispielsweise obige atomare Operationen, um auf eine gemeinsame Zählervariable zuzugreifen. Mutexes besitzen einen Eigentümer und eine zugehörige Warteliste, in der alle Prozesse verzeichnet sind, die auf das Freiwerden warten. Während dieser passiven Wartephase können andere niederpriore Prozesse rechnen.

Der Nachteil von Mutexes ist in der Praxis, dass für jeden Test ein Betriebssystemaufruf notwendig ist, dessen Wechsel in den geschützten Kernel-Modus zusätzliche Laufzeit verursacht. Die nicht-Echtzeitversion der KogMo-RTDB verwendet jedoch Operationen, die von aktuellen POSIX-Bibliotheken unter dem GPOS Linux durch *Futex*-Operationen realisiert werden. Ein *Futex* (*fast userspace mutex*) [52] benötigt im wartefreien Fall keinen Betriebssystemaufruf und ist so in diesem häufig auftretenden Fall deutlich schneller. In der Echtzeitversion ist die Laufzeitverlängerung durch die obligatorischen Echtzeitbetriebssystemaufrufe der Preis für harte Realzeit.

Zustandsvariablen (*condition variables*): Diese dienen nach [91] dem Warten auf das Eintreten eines Ereignisses. In der verwendeten Implementierung wird die Mesa-Semantik [117] verwendet und jede Zustandsvariable mit einem Mutex verbunden, damit bei der Zustandsprüfung keine Wettlaufsituation (*race condition*) entstehen kann.

Für Echtzeitsysteme sind atomare Operationen die günstigste Option. Wenn dies nicht realisierbar ist, sollten Mutexes eingesetzt werden. Die Deaktivierung von Interrupts sollte unbedingt vermieden werden. Diese Überlegungen begründen das Design der KogMo-RTDB-Methoden.

5.5.7 Synchronisation von KogMo-RTDB Transaktionen

Im Folgenden wird nun der Einsatz der genannten Synchronisierungsmethoden und deren Auswirkungen auf die Laufzeit von Transaktionen der KogMo-RTDB untersucht. Dabei wird ein Prozess betrachtet, der ein Objekt in der KogMo-RTDB angelegt hat, somit der Besitzer dieses Objekts ist, und nun unter harten Echtzeitbedingungen Lese- und Schreibzugriffe auf dieses Objekt machen möchte. Nur wenn die Priorität des betrachteten Prozesses höher ist, als die der konkurrierenden Prozesse, ist die durch die KogMo-RTDB betrachtete Blockierungszeit maßgeblich.

Das Design der KogMo-RTDB ist darauf ausgelegt, die Interferenzen zwischen zwei Transaktionen zu minimieren. Da jedes einzelne Objekt dazu eigene Mutexes und Zustandsvariablen besitzt, ist der zu betrachtende ungünstigste Fall der *gleichzeitige* Zugriff auf *dasselbe* Objekt. Tabelle 5.2 gibt einen Überblock über die dabei auftretenden maximalen Blockierungen des Echtzeitprozesses durch einen anderen Prozess. In Abhängigkeit der

Tabelle 5.2: Maximal mögliche Blockierungen einer Transaktion

Transaktion des ersten Prozesses (Besitzer des Objekts)	**Konkurrierende Transaktion eines zweiten Prozesses im ungünstigsten Fall**		
	Lesen	Schreiben	Benachrichtigung
Lesen	keine Blockierung (1.)	keine Blockierung (1.)	keine Blockierung (1.)
Schreiben	keine Blockierung (aber Mutex-Test) (3.)	Kopierdauer Objektdaten (2.)	Lesedauer Objektbasisdaten (4.)

genutzten KogMo-RTDB Methoden aus Kapitel 5.3.2 müssen die folgenden Fälle unterschieden werden:

1. In der KogMo-RTDB kommen zum Erhalt der Datenkonsistenz im blockierungsfreien Schreib-/Leseprotokoll aus Kapitel 5.3.5 das atomare Umsetzen eines Index und einer Validitätsvariable zur Anwendung. Ein Lesezugriff ist damit grundsätzlich blockierungsfrei.

2. In der KogMo-RTDB wird für jedes Objekt ein Mutex als Sperre verwendet, das eine gleichzeitige Modifikation durch mehrere Prozesse verhindert. Eine Wartephase tritt daher nur auf, wenn zwei Schreiboperationen auf das gleiche Objekt gleichzeitig durchgeführt werden. Die maximale Dauer ist, wie in Kapitel 6.1.2 vermessen, für große $n_{bytes} > 10 \cdot 10^4$ maßgeblich bestimmt durch $t_{copy,\mathcal{D}}$ aus Kapitel 5.3.5. $t_{copy,\mathcal{D}}$ ist beschränkt durch $n_{bytes} \leq n_{bytes,max}$, das zur Lebensdauer eines Objekts unveränderbar ist. Die maximale Wartezeit ist damit deterministisch.

3. Auch wenn keine weitere Schreiboperation stattfindet, muss vor der Schreiboperation der zugehörige Mutex geprüft und gesperrt werden. Dies erfordert einen Systemaufruf, der zwar in einem Realzeitbetriebssystem deterministisch ist, aber zusätzliche Zeit benötigt.

4. Zur Vermeidung einer Wettlaufsituation wird im Benachrichtigungsalgorithmus aus Abschnitt 5.3.6 für die kurze Dauer der Überprüfungsphase ein Mutex gesperrt, der ebenfalls im Schreibalgorithmus eingesetzt wird. Aus Sicht des Schreibprozesses entsteht dadurch eine weitere mögliche Blockierungssituation. Deren Dauer $c_{notifycheck}$ ist maßgeblich bestimmt durch die Lesedauer des aktuellen Historienspeicherplatzes des Objektes. Dazu muss jedoch nicht das vollständige Objekt gelesen werden, sondern lediglich die Objektbasisdaten. Somit ist diese Blockierungsdauer nicht von der individuellen Objektgröße abhängig, sondern für jedes Objekt konstant.

Sind die genannten Blockierungen für eine Anwendung nicht tolerierbar, können sie beim Einfügen jedes Objektes individuell deaktiviert werden. Dazu dienen folgende Attribute des internen Konfigurationsparameters $flags$ im statischen Metadatenblock eines jeden Objekts aus Tabelle 5.1:

write_allow: Durch Aktivieren dieser Eigenschaft wird das Objekt zum öffentlichen Schreiben durch andere Prozesse freigegeben. Da dies gerade für größere Objekte die Blockierungszeiten signifikant erhöhen kann und zudem in der Regel unerwünscht ist, ist dieses Attribut standardmäßig nicht gesetzt.

no_notifies: Ein Setzen dieses Attributs verhindert eine aktive Benachrichtigung von Prozessen, die auf eine Änderung dieses Objekts warten. Da die zeitliche Auswirkung für den Schreibenden minimal ist, wird dies in der Standardkonfiguration erlaubt. Ohne Benachrichtigungssignale müssen Änderungen durch periodische Überprüfung (*Polling*) festgestellt werden (vgl. Abschnitt 5.3.6). *Polling* kann auch beim Einsatz dieser Architektur zur Kommunikation zwischen virtuellen Maschinen eingesetzt werden, wenn keine Synchronisierungsprimitive zwischen diesen verfügbar sind.

Bei der Wahl einer restriktiven Objektkonfiguration mit $(write_allow, no_notifies) = (0, 1)$ wird der Schreibalgorithmus frei von ggf. blockierenden Systemaufrufen und es ergibt sich das in Tabelle 5.3 gezeigte blockierungsfreie Zugriffsverhalten. Der durchlaufene Programmpfad beschränkt sich auf die KogMo-RTDB und ist dort kurz und überschaubar. Zudem erreicht der Algorithmus seine kürzeste WCET. Ob diese Konfiguration gewählt werden kann, ist davon abhängig, ob die Anwendungen auf die genannten Eigenschaften verzichten können und wie wichtig ihnen das Zeitverhalten ist.

Tabelle 5.3: Mögliche Blockierungen bei restriktiver Objektkonfiguration

Blockierung des ersten Prozesses (Besitzer des Objekts)	**Aktion eines zweiten Prozesses**		
	Lesen	Schreiben	Benachrichtigung
Lesen	keine Blockierung	nicht erlaubt	nicht erlaubt
Schreiben	keine Blockierung	nicht erlaubt	nicht erlaubt

5.5.8 Prioritätserhalt bei konkurrierenden Transaktionen

Für die Einhaltung der harten Echtzeitfähigkeit durch die KogMo-RTDB sind zwei Punkte unverzichtbar:

- Die Ausführungszeit jeder RTDB-Transaktion ist deterministisch.
- Die Priorität eines Prozesses bleibt erhalten, während Methoden der RTDB-Bibliothek im Kontext des aufrufenden Prozesses ausgeführt werden.

Die maximalen Ausführungszeiten werden durch statische Analyse in Abschnitt 5.5.9 beschränkt, indem der längste Programmpfad bestimmt und die maximale Anzahl der Iterationen jeder Programmschleife begrenzt wird. Dabei werden auch die möglichen Blockierungen durch andere Prozesse berücksichtigt. Dies wird in Kapitel 6.1 durch Messungen verifiziert. Zum Prioritätserhalt wird im Folgenden dargelegt, wie eine Prioritätsinversion bei der Ausführung von Transaktionen verhindert wird.

Für die folgende Untersuchung wird ein kognitives System angenommen, das aus einer Menge $\mathcal{T}$ von m echtzeitfähigen Rechenprozessen (in der Echtzeit-Literatur meist Task genannt) T_{RT} und n nicht-echtzeitfähigen Tasks T_{NRT} besteht:

$$\mathcal{T} = \{T_{RT,1}, ..., T_{RT,m}, T_{NRT,1}, ..., T_{NRT,n}\} \tag{5.12}$$

Bei der Untersuchung des Zeitverhaltens der KogMo-RTDB ist der einfache Fall statischer Prioritäten ausreichend, der auch in der Praxis eine dominante Rolle spielt. Der Task $T_{RT,1}$ hat dabei die höchste Priorität, der Task $T_{NRT,n}$ die niedrigste. Die Priorität p eines Tasks T_{RT} ist grundsätzlich höher als die eines T_{NRT}. Somit gilt:

$$p(T_{RT,1}) > ... > p(T_{RT,m}) > p(T_{NRT,1}) > ... > p(T_{NRT,n}) \tag{5.13}$$

Der Scheduler des Betriebssystems sorgt dafür, dass stets genau der Task den nächsten freiwerdenden Prozessor zugeteilt bekommt, der rechenbereit ist und die höchste Priorität besitzt. In einem Echtzeitbetriebssystem umfasst die Priorisierung auch sämtliche Systemaufrufe und betriebssysteminterne Funktionen. Die in Kapitel 5.4.4 vorgestellte Lösung hält die vorgegebenen Prioritäten auch beim gemischten Einsatz von Echtzeit- und Nicht-Echtzeittasks ein.

Konkurrierende Echtzeit- und nicht-Echtzeittasks

Das obige Problem aus $\mathcal{T}$ lässt sich reduzieren auf das Teilproblem der Taskmenge $\mathcal{T}_{red} = \{T_{RT,1}, T_{NRT,n}\} \subseteq \mathcal{T}$, bestehend aus zwei repräsentativen Rechenprozessen:

Echtzeitfähiger E/A-Schnittstellenprozess höchster Priorität ($T_{RT,1}$): Dieser zeitkritische Prozess wartet auf ein rechnerexternes Ereignis, das eine Unterbrechungsanforderung (*Interrupt*) auslöst, die ihm selbstverständlich über einen echtzeitfähigen Gerätetreiber wie aus Kapitel 5.4.7 signalisiert wird. Nach dessen Eintreffen muss der Prozess ein KogMo-RTDB-Objekt schreiben, das jedoch auch von anderen Prozessen niedrigerer Priorität geschrieben werden darf.

Danach führt er Berechnungen durch und gibt ggf. ein Notaus-Signal aus. In einem ausführlicheren Beispiel könnte das Notaus-Signal von einem weiteren Prozess generiert werden, der das eben geschriebene KogMo-RTDB-Objekt auswertet.

Entscheidend ist, dass für den gesamten Programmpfad, in dem sich der Schreibzugriff befindet, harte Echtzeitanforderungen bestehen.

Nicht echtzeitfähiger Benutzer-Dialogprozess niedrigster Priorität ($T_{NRT,n}$): Dieser zeitunkritische Prozess dient dem Dialog mit einem Benutzer. Es werden Eingaben entgegengenommen und diese in Objekte geschrieben, die z.B. Konfigurationsparameter aufnehmen.

Im Lauf des Programmablaufs wird auch das oben genannte Objekt von $T_{RT,1}$ geschrieben.

Das hier skizzierte Problem dient dazu, den Worst-Case zu untersuchen. Ein Anwendungsfall für ein öffentlich beschreibbares Objekt ist die Protokollierung von Ereignissen und Textmeldungen, um einen einheitlichen echtzeitfähigen Mechanismus ähnlich dem Systemprotokoll von UNIX (*syslog*) zu realisieren. Es wird angenommen, dass die Maximalgröße des Objekts großzügig dimensioniert wird, sodass dessen Kopierzeit im Worst-Case für die Berechnung der Schreibdauer maßgeblich ist.

Echtzeitfähiger Ablauf konkurrierender Transaktionen

Nun soll im Detail der Ablauf konkurrierender Transaktionen am Beispiel der zwei exemplarischen Tasks untersucht werden. Das Diagramm 5.14 zeigt den zeitlichen Ablauf der maßgeblichen Transaktionen. Es wird unterschieden, ob die Programmausführung im Kontext von $T_{RT,1}$ (links) oder $T_{NRT,n}$ (rechts) geschieht. Es wird weiter differenziert, welcher Systemteil die Kontrolle hat:

- **Linux:** Es werden Systemaufrufe des nicht-echtzeitfähigen Standardbetriebssystems Linux genutzt.
- **Xenomai:** Die Echtzeiterweiterung Xenomai bzw. der Adeos-Nanokernel übernehmen die Kontrolle.
- **Echtzeitprozess (1):** Die Kontrolle liegt bei $T_{RT,1}$. Es wird vorausgesetzt, dass dort nur echzeitfähige Algorithmen implementiert sind.
- **Nicht-Echtzeitprozess (2):** Es wird $T_{NRT,n}$ ausgeführt. Dessen Verhalten und Algorithmen unterliegen keinerlei Einschränkungen. Es muss davon ausgegangen werden, dass er sich „maximal echtzeitschädlich“ verhält.
- **KogMo-RTDB:** Es wird eine Funktion der KogMo-RTDB abgearbeitet. Diese befindet sich in der dynamischen Bibliothek `libkogmo_rtdb_rt` (*shared object*, Dateiendung „.so“ unter UNIX), die zu allen Programmen dazugebunden wird. Sie läuft auch im Programmkontext des nutzenden Programms.

Anhand Abb. 5.14 wird nun verdeutlicht, wie in der entworfenen Echtzeitarchitektur konkurrierende Transaktionen behandelt werden und eine Prioritätsinversion verhindert wird:

1. Der Prozess $T_{RT,1}$ wird gestartet. Zu Beginn läuft er in einem nicht-echtzeitfähigen Linux-Kontext.
2. Er alloziert Speicher für eine spätere Verwendung und liest eine Datei mit Konfigurationsparametern ein. Dabei verwendet er gefahrlos die Funktionen der C-Standardbibliothek. Würde der Prozess als ein Kernel-Modul laufen, wie dies bei anderen Echtzeitlösungen notwendig ist, stünde ihm diese Bibliothek nicht direkt zur Verfügung und es wäre dafür ein größerer Softwareaufwand nötig.
3. Der Prozess meldet sich bei der KogMo-RTDB an, indem er die Bibliotheksfunktion `kogmo_rtdb_connect()` aufruft.
4. Die KogMo-RTDB meldet den Prozess beim Echtzeitscheduler von Xenomai an. Der Prozess wechselt in den primären Modus (vgl. Kapitel 5.4.4) und kann so nicht mehr vom Linux-Kernel oder von Prozessen niedrigerer Priorität unterbrochen werden.
   Der Programmfluss unter harter Echtzeit wird im Folgenden durch eine Doppellinie symbolisiert.
5. Es wird auf externe Daten von einer Schnittstelle gewartet, z.B. von einem Sensor über CAN. Dabei wird ein Echtzeittreiber verwendet, der nur zur Initialisierung auf

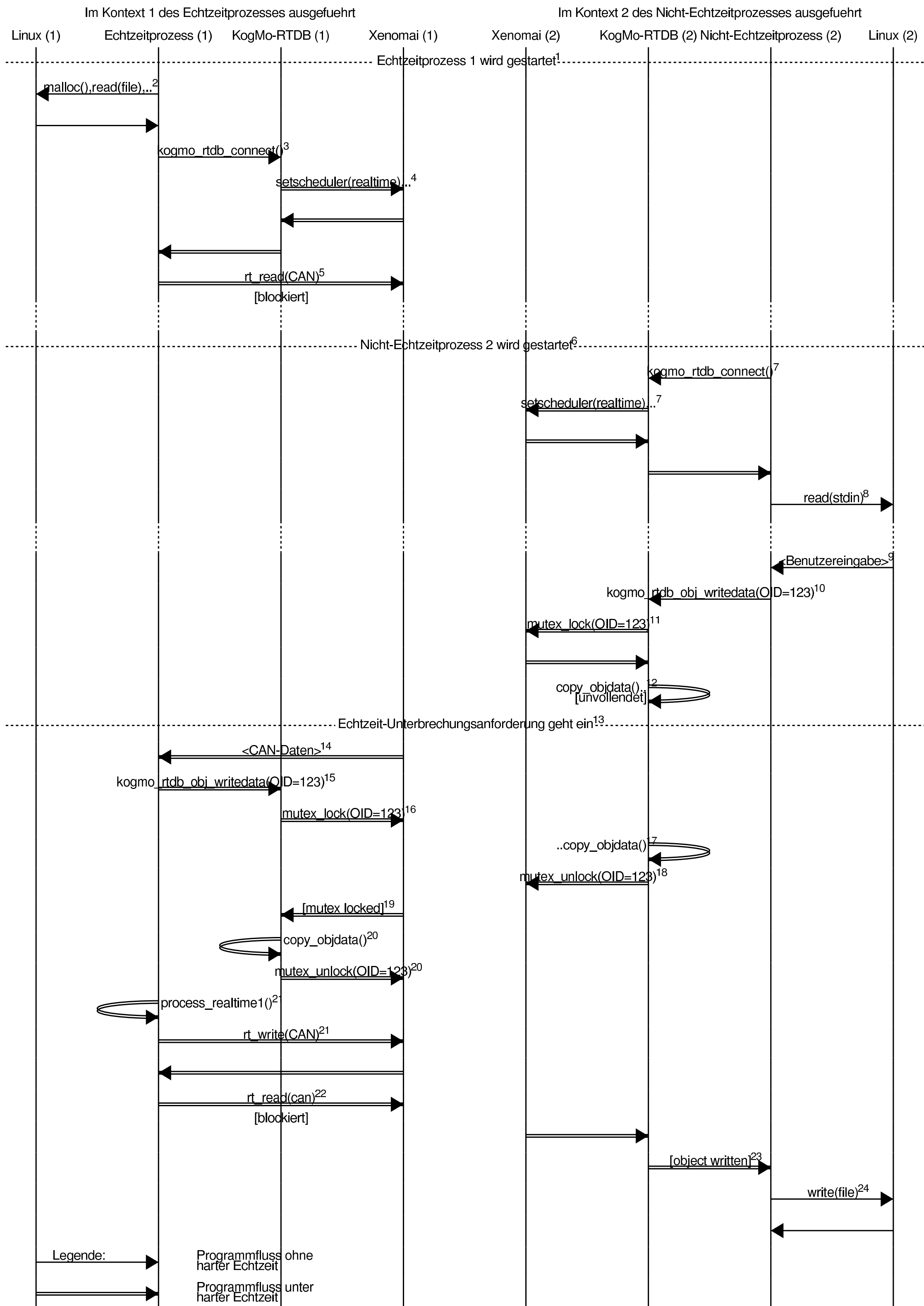


Abbildung 5.14: Programmfluss bei konkurrierendem Schreibzugriff im Worst-Case

Linux angewiesen ist und zur Laufzeit ausschließlich die echtzeitfähigen Schnittstellen von Xenomai nutzt.

6. Der Prozess $T_{NRT,n}$ wird vom Benutzer gestartet.
7. Der Prozess meldet sich bei der KogMo-RTDB an und wird von dieser auch bei Xenomai als Echtzeitprozess registriert, jedoch mit einer sehr niedrigen Priorität. So kann die KogMo-RTDB später beliebig in den Echtzeitmodus wechseln und die Funktionen des Echtzeitbetriebssystems nutzen.
8. Der Prozess wartet auf eine Benutzereingabe. Durch den erforderlichen Linux-Systemaufruf wechselt er in den nicht-echtzeitfähigen sekundären Modus von Xenomai und steht wieder unter der Kontrolle des Linux-Schedulers.
9. Der Benutzer hat eine Eingabe getätigt.
10. Aufgrund der Eingabe möchte $T_{NRT,n}$ das KogMo-RTDB-Objekt mit der $OID = 123$ mit neuen Daten aktualisieren.
11. Da das gewünschte Objekt auch von anderen Prozessen beschreibbar ist, muss der Schreibalgorithmus der KogMo-RTDB das Objekt vor weiteren Schreibzugriffen schützen (vgl. Kapitel 5.5.7). Dazu wird ein vom Echtzeitbetriebssystem kontrollierter Mutex gesperrt und zuvor in den Echtzeitmodus gewechselt.
12. Ebenfalls im Echtzeitmodus wird nun das Kopieren der neuen Objektdaten begonnen. Dies kann von jedem Echtzeitprozess höherer Priorität unterbrochen werden, nicht jedoch von Nicht-Echtzeitprozessen oder dem Linux-Kernel.
13. Nachdem über die CAN-Schnittstelle neue Daten empfangen wurden, wird eine Unterbrechungsanforderung (*Interrupt*) ausgelöst.
14. In der in Abb. 5.8 gezeigten Architektur wird der Interrupt vom ADEOS-Nanokernel aus Kapitel 5.4.4 abgefangen, noch bevor die Kontrolle einen nicht-echtzeitfähigen Linux-Treiber erreicht hat. Da der CAN-Echtzeittreiber seinen Interrupt bei der Initialisierung registriert hat, läuft der Interrupt in der ADEOS-Ereignisleitung aus Abb. 5.9 nur bis zum Echtzeitkern von Xenomai, der die Kontrolle umgehend an $T_{RT,1}$ abgibt.
15. $T_{RT,1}$ möchte nun ebenfalls das KogMo-RTDB-Objekt mit der $OID = 123$ schreiben.
16. Der Schreibalgorithmus der KogMo-RTDB versucht nun ebenfalls den Mutex zum Schreibschutz des Objekts zu sperren.
    Der Mutex ist bereits im Besitz von $T_{NRT,n}$ und kann nicht gesperrt werden. Da sie jedoch von der KogMo-RTDB mit dem Attribut zur Prioritätsvererbung (*priority inheritance*) versehen ist, setzt das Echtzeitbetriebssystem nun entsprechend dem Prioritätsvererbungsprotokoll (PIP) [182] die niedrige Priorität von $T_{NRT,n}$ auf die höhere Priorität $p(T_{RT,1})$.
17. Da $T_{RT,1}$ nun bis zum Freiwerden des Mutex blockiert, gibt der Echtzeitscheduler nun die Kontrolle an $T_{NRT,n}$ ab und der Kopiervorgang wird fortgesetzt.

An dieser kritischen Stelle ist es essentiell, dass die Kontrolle auf keinen Fall an das Anwenderprogramm zurückgegeben wird. Außerdem darf der Algorithmus der KogMo-RTDB keine Systemaufrufe enthalten. Wäre dies nicht berücksichtigt, könnte an dieser Stelle eine *Prioritätsinversion* bis auf die niedrigste Prioritätsebene des Nicht-Echtzeitbetriebssystems auftreten. Die entstehenden maximalen Ausführungszeiten wären unbeschränkbar.

18. Nach Abschluss der KogMo-RTDB-Transaktion zum Aktualisieren des Objekts wird der Mutex wieder freigegeben. Dadurch verliert $T_{NRT,n}$ wieder seine „geerbte“ hohe Priorität.

19. Der Echtzeitscheduler von Xenomai gibt die Kontrolle an die KogMo-RTDB im Kontext des Echtzeitprozesses zurück und sperrt den Schreibmutex nun mit dem neuen Besitzer.

20. Nun kann der Schreibvorgang für den Echtzeitprozess durchgeführt werden und dann der Mutex wieder freigegeben werden.

21. $T_{RT,1}$ führt weitere Berechnungen durch und verschickt als Reaktion eine CAN-Botschaft über den Echtzeittreiber. Dies könnte z. B. eine Notstopp-Nachricht sein, wie sie in den Anwendungsfällen aus Kapitel 6.4 eingesetzt wird.

22. $T_{RT,1}$ wartet auf die nächste eingehende Botschaft und blockiert.

23. Nachdem nun kein Prozess höherer Priorität mehr lauffähig ist, geht die Kontrolle wieder an $T_{NRT,n}$ über, dessen KogMo-RTDB Schreiboperation nun abgeschlossen ist.

24. $T_{NRT,n}$ schreibt Daten in eine Datei und verliert durch den Linux-Systemaufruf seinen Echtzeitkontext wieder.

Die Analyse des Ablaufs der zwei Transaktionen zeigt, wie mit der Priorität des höherprioren Prozesses eine konkurrierende Transaktion zu Ende geführt wird. Die dabei entstehenden maximalen Ausführungszeiten sind durch die Architektur der KogMo-RTDB a priori bestimmbar und in den Echtzeitnachweis, wie in Anschnitt 5.5.9 durchgeführt, einbeziehbar.

Der kritische Abschnitt des obigen Ablaufs sind die Punkte 13 bis 21, die die Zeit vom eingehenden externen Hardwaresignal bis zur Reaktion mit einem ausgehenden Signal umfassen. In Abschnitt 6.1 werden die dabei auftretenden Laufzeiten unter verschiedenen Bedingungen quantitativ durch Messungen bestimmt.

Die gezeigte Beispielanwendung dient vor allem der obigen Untersuchung. In einer realistischen Anwendung würde man erst die CAN-Botschaft schicken und dann das Objekt schreiben. In den Messungen in Kapitel 6.1.3 wird für die Ausgabe des Signals ein weiterer Prozess eingesetzt, sodass der gesamte Kommunikationskanal über die KogMo-RTDB vermessen wird.

5.5.9 Echtzeitnachweis

Anhand der Ergebnisse aus den vorangegangenen Kapiteln kann nun für die Kommunikation in einem System, das die entworfene Architektur verwendet, der Echtzeitnachweis erbracht werden.

Taskmodell

Betrachtet wird ein Task T_i aus einer Menge von n Tasks $\mathcal{T} = \{T_1, ..., T_i, ..., T_n\}$. Jeder Task T hat seine eigene statische Priorität p, wobei T_1 die höchste und T_n die niedrigste Priorität besitzt:

$$p(T_1) > ... > p(T_i) > ... > p(T_n) \tag{5.14}$$

Dabei wird nun entgegen (5.12) und (5.13) nicht mehr explizit in Echtzeit und Nicht-Echtzeit unterschieden, da bereits in Abschnitt 5.5.8 gezeigt wurde, dass dies für RTDB-Transaktionen keinen Unterschied darstellt. Es wird vorausgesetzt, dass T_i und alle höherprioren Tasks $T \in \{T_1, ..., T_{i-1}\}$ hart echtzeitfähig sind und die Regeln aus Abschnitt 5.4.8 befolgen.

Ein Task T_i wird hier durch folgendes mathematisches Modell beschrieben:

- c_i gibt die Rechenzeit eines Tasks im *Worst Case* mit allen notwendigen Korrekturfaktoren für hardwarebedingte Laufzeitverlängerungen an. Die Ermittlung von t_{WCET} in der ungünstigsten Konstellation aus *Caches* und *Translation Lookaside Buffers*(TLB) wird in [24] beschrieben, es gilt: $c_i = t_{WCET}$.
- t_i spezifiziert die Periode eines Tasks. Wie in Abschnitt 5.3.2 ausgeführt, muss jeder RTDB-Prozess seine Zykluszeit spezifizieren. Daher kann $t_i = t_{cycle,min}$ aus dessen öffentlichem Prozessobjekt nach Tabelle 5.1 entnommen werden.
- d_i: Notwendige Reaktionszeit eines Tasks (*Deadline*), es wird $d_i \leq t_i$ vorausgesetzt.

Systemmodell

Die folgende Analyse geht von einem Systemmodell mit folgenden Eigenschaften aus:

Scheduling: Es kommt ein prioritätsbasierter präemptiver echtzeitfähiger Scheduler mit statischen Prioritäten (*fixed priority scheduling*) zum Einsatz. Dieser einfachste allgemeine Fall ist gerechtfertigt, da er aufgrund seiner niedrigen Komplexität noch immer in vielen Anwendungen, wie auch dem kognitiven Automobil, eingesetzt wird. Die erzielten Ergebnisse lassen sich auch mit anderen Schedulingverfahren nutzen. Jedoch steht die Optimierung des Schedulings nicht im Fokus dieser Arbeit. Das in Abschnitt 5.4.4 vorgestellte Betriebssystem Linux mit der Realzeiterweiterung Xenomai und dem ADEOS Nanokernel erfühlt die gestellte Anforderung.

Prioritätsvergabe: Für die Verteilung der fixen Prioritäten bietet sich *Rate Monotonic* nach [121] an, für das zusätzlich zu (5.14) gelten muss:

$$t_1 \leq ... \leq t_i \leq ... \leq t_n \tag{5.15}$$

Da jedes Modul einer Verarbeitungskette jeweils auf ein Trigger-Ereignis durch neue Daten seines Vorgängers wartet (vgl. Kapitel 6.3), sind die niedrigsten Latenzzeiten durch steigende Prioritäten entlang der Kette von den Sensoren zu den Aktoren zu erreichen. Dadurch werden ergebnisunrelevante Organisationstätigkeiten eines jeden Zyklus erst nach der Ausgabe des Aktorkommandos erledigt.

Taskauslösung: Im kognitiven Fahrzeug kommen, wie in Abb. 5.1 gezeigt, vor allem zwei Arten von Echtzeit-Tasks vor: Schnittstellenmodule und Wahrnehmungsmodule. Ein Task eines Schnittstellenmoduls wird durch einen Echtzeit-Interrupt aktiviert (siehe Abschnitt 5.4.6), der bei neuen Daten eines Peripheriegerätes ausgelöst wird. Da die meisten Peripheriegeräte wie Kameras, LIDAR-Sensoren, GPS, usw. zyklisch Daten mit t_{sensor} liefern, kann für die Schnittstellentasks mit der bekannten Periode gerechnet werden: $t_i = t_{sensor}$.

Die meisten Wahrnehmungsmodule, insbesondere die unterer Ebenen, sind dadurch charakterisiert, dass sie jeweils durch neue Eingangsdaten ausgelöst werden, auf die sie mit `readdata_waitnext(`$t_{committed,known}$`)` aus Abschnitt 5.3.2 warten. So ergibt sich eine dem Datenfluss entgegen gerichtete Kette von Abhängigkeiten, die sich bis zu einem Schnittstellentask fortsetzt. Alle Tasks dieser Kette haben dann dieselbe Periode, für die Phase ϕ gilt: $\phi_i = \phi_{i-1} + c_{i-1}$. Wissensbasierte Wahrnehmungsmodule höherer Ebenen und Überwachungsmodule sind oft eigenständig und analysieren ausgewählte Objekte in der KogMo-RTDB unabhängig (ϕ_i beliebig) mit ihrer eigenen Periode t_i.

Da nach dieser Betrachtung der Anwendung alle ereignisgesteuerten (*event-triggered*) Tasks auf eine Zeitsteuerung (*time-triggered*) zurückgeführt werden können, werden im Weiteren nur periodische Tasks betrachtet. Je nach Anwendung muss sonst z.B. ein Ereignisstrom nach [81] betrachtet werden.

Nachweisverfahren

Der Nachweis für einen Task T_i wird nach [127] für die kritische Situation (*critical instant*) vorgenommen, dass zusammen mit T_i alle höherprioren Tasks $T_1, ..., T_{i-1}$ gleichzeitig aktiviert werden. Die Annahme der gleichen Phasenlage $\phi_1 = ... = \phi_i$ führt zu einer Überabschätzung, verzichtet aber auf den anderenfalls erforderlichen Datenabhängigkeitsgraphen der Tasks. Ein Task T_i ist in einen existierenden Schedule aus $T_1, ..., T_{i-1}$ einreihbar, wenn gilt (vgl. z.B. [128]):

$$d_i \geq t = c_i + \sum_{j=1}^{i-1} \left\lceil \frac{t}{t_j} \right\rceil \cdot c_j \tag{5.16}$$

Sind alle t Vielfache voneinander, $\frac{c}{t}$ klein oder im Schedule noch viel Leerlauf, kann die triviale Lösung $t = d_i$ schnell zum Ergebnis führen. Sonsten muss nach einem passenden t gesucht werden, beispielsweise iterativ nach [9]. Existiert ein Zeitpunkt $t \leq d_i$, ist bewiesen, dass T_i seine Deadline d_i in jedem Fall einhalten kann. Kann kein t gefunden werden, ist eine schritthaltende Verarbeitung von T_i nicht möglich.

Erweiterung um RTDB-Einflüsse

Die Rechenzeit c_i eines Tasks T_i wird im Folgenden zerlegt in die Zeit zur Ausführung von RTDB-Transaktionen und die verbleibende Zeit z.B. für Berechnungen von taskspezifischen Algorithmen, deren Echtzeitfähigkeit als gegeben betrachtet wird:

$$c_i = c_{RTDB} + c_{non-RTDB} \tag{5.17}$$

Aus der Menge möglicher RTDB-Transaktionen werden hier zur Vereinfachung nur je ein Nutzdatenzugriff auf die von einem Task verwendete Menge an Leseobjekten $\mathcal{S}_{read}$ und Schreibobjekten $\mathcal{S}_{write}$ betrachtet. Diese Annahme ist zulässig, da der Task diese Daten intern zwischenspeichern kann. Ein blockierendes Warten auf neue Daten sollte, wie in Kapitel 6.3 gezeigt, nur einmal pro Periode zur Taskaktivierung eingesetzt werden, und wird daher genauso wie Metadatenoperationen, die sich auf die Initialisierungsphase beschränken lassen, zur Vereinfachung vernachlässigt:

$$c_{RTDB} \approx \sum_{\mathcal{D} \in \mathcal{S}_{read}} c_{read,\mathcal{D}} + \sum_{\mathcal{D} \in \mathcal{S}_{write}} c_{write,\mathcal{D}} \tag{5.18}$$

Die WCET von $c_{read,\mathcal{D}}$ wurde bereits in (5.7) berechnet, und ist dank des blockierungsfreien Leseprotokolls aus Kapitel 5.3.5 unabhängig vom Verhalten anderer Prozesse. Für Schreiboperationen ist wie in Abschnitt 5.5.7 untersucht, die individuelle Konfiguration jedes Objekts, ausgedrückt durch $flags = (write_allow, no_notifies, \dots)$ in den Objektmetadaten $\mathcal{M} = (flags, \dots)$ relevant. Sie bestimmt, ob kurzzeitige Blockierungen durch konkurrierende Transaktionen niederpriorer Tasks zu Synchronisierungszwecken zulässig sind. Für eine mit $(write_allow, no_notifies) = (0, 1)$ ungestörte Schreiboperationen nach dem Algorithmus aus Abb. 5.3 gilt:

$$c_{write,nonblocking,\mathcal{D}} = c_{writeinit} + c_{writeinvalidate} + c_{writecopy} + c_{writevalidate} + c_{writenotify} \tag{5.19}$$

Da die KogMo-RTDB Objekte pro Schreibvorgang variabler Größe $n_{bytes,write} \leq n_{bytes,max,\mathcal{D}}$ erlaubt und die aktuelle Größe als n_{bytes} im Kopf des Nutzdatenblocks auf jedem Ringspeicherplatz (*Historyslot*) festgehalten wird, kann der Kopiervorgang nach $n_{bytes,write}$ abgebrochen werden:

$$c_{writecopy} = c_{copy,byte} \cdot n_{bytes,write} \tag{5.20}$$

Die maximalen Ausführungszeiten der anderen Phasen des Algorithmus sind unabhängig von den Objektkonfigurationen und a priori bestimmbar.

Wird ein Objekt nur mit einem öffentlichen Schreibrecht $(write_allow, no_notifies) = (1,1)$ angelegt, entsteht die längst mögliche Laufzeit, wenn der höherpriore Task den Schreibmutex des Objekts in dem Moment belegen will, in dem sie gerade der niederpriore Task belegt hat. Dabei kann es zwar, wie in Abschnitt 5.5.8 dargelegt, durch die erstellte Architektur zu keiner Prioritätsinversion kommen, doch der niederpriore Task muss erst seine Phasen *invalidate-copy-validate* durchlaufen, um keine Inkonsistenzen entstehen zu lassen. Für eine maximale Laufzeit muss in diesem Fall damit gerechnet werden, dass der niederpriore Task einen Datenblock maximaler Größe $n_{bytes,max}$ schreibt und es gilt:

$$\begin{aligned} c_{write,writeblocking,\mathcal{D}} =& c_{writeinit} + 2 \cdot c_{writeinvalidate} + c_{copy,byte} \cdot (n_{bytes,write} + n_{bytes,max}) + \\ &+ 2 \cdot c_{writevalidate} + c_{writenotify} \end{aligned} \tag{5.21}$$

Sind bei einem Objekt die Benachrichtigungen $(write_allow, no_notifies) = (0,0)$ erlaubt, kann der Schreibvorgang um die Zeit $c_{notifycheck}$ aus Abschnitt 5.3.6 verlängert werden, die der wartende Prozess benötigt, um mit belegtem Benachrichtigungsmutex seine Wartebedingung zu prüfen:

$$\begin{aligned} c_{write,waitblocking,\mathcal{D}} =& c_{writeinit} + c_{writeinvalidate} + c_{copy,byte} \cdot n_{bytes,write} + \\ &+ c_{writevalidate} + c_{notifycheck} + c_{writenotify} \end{aligned} \tag{5.22}$$

Für die WCET einer Schreiboperation ergibt sich durch Zusammenfassung von (5.19), (5.21) und (5.22) allgemein:

$$\begin{aligned} c_{write,\mathcal{D}} =& c_{writeinit} + c_{writeinvalidate} + c_{copy,byte} \cdot n_{bytes,write} + c_{writevalidate} + c_{writenotify} + \\ &+ \begin{cases} c_{writeinvalidate} + c_{copy,byte} \cdot n_{bytes,max} + c_{writevalidate} & \text{für } write_allow = 1 \\ 0 & \text{sonst} \end{cases} \\ &+ \begin{cases} c_{notifycheck} & \text{für } no_notifies = 0 \\ 0 & \text{sonst} \end{cases} \end{aligned} \tag{5.23}$$

Um den Nachweis durchzuführen, muss (5.7) und (5.23) mit (5.18) in (5.17) eingesetzt werden, um die gesamte Rechenzeit c_i eines jeden T_i zu ermitteln, und (5.16) für alle Tasks erfüllt sein. Sind Parameter einzelner Tasks nicht bekannt, können diese mit den Methoden aus Kapitel 6.3.2 gemessen werden.

Hinzunahme weiterer Tasks zur Laufzeit

Ist für eine Menge $\mathcal{T}$ von Tasks der Echtzeitnachweis erbracht und sind diese auf der entwickelten Architektur implementiert, ist es sogar möglich, zur Laufzeit neue Tasks hinzuzunehmen, ohne dass diese das bisherige System beeinträchtigen.

Eine Voraussetzung ist, dass sich der neue Task T_{new} in den bestehenden Schedule einfügen lässt. Informationen über T_{new} sind gerade bei einem Entwicklungssystem oft nicht verfügbar oder die WCET von T_{new} ist noch nicht berechnet. Daher wird dem neuen Task eine Priorität zugeordnet, die niedriger ist als alle bestehenden in (5.14):

$$p(T_n) > p(T_{new}) \tag{5.24}$$

Dies ist ein häufiger Fall: Das bestehende System ist z. B. ein Sicherheitssystem wie aus Kapitel 6.4.2. Der neue Task ist z. B. eine unkritische Zusatzfunktion, eine Funktion in Erprobung oder ein Visualisierungsprozess (vgl. Kapitel 6.3). Kommuniziert T_{new} nun mit den bestehenden Tasks direkt (*Request/Response*-Prinzip), würde er dadurch dessen Laufzeiten verlängern. Verwendet er dafür die KogMo-RTDB, ist durch deren Methoden sichergestellt, dass die Laufzeiten deren RTDB-Operationen wie in (5.23) beschränkt bleiben.

Der neu hinzukommende Task muss selbst entscheiden, ob er in der bestehenden Objektkonstellation lauffähig ist, d. h. ob alle benötigten Objekte vorhanden sind und die dort spezifizierten Historienzeiten für ihn ausreichen sind (vgl. Kapitel 5.3.5). Denn der neue Task kennt am ehesten seine Laufzeitanforderungen. Ist ein Task für das System wichtig, darf ein hochpriorer Überwachungsprozess, wie in Kapitel 6.3.3 realisiert, das System erst freigeben, wenn dieser ordnungsgemäß läuft und seine Bereitschaft signalisiert.

In obiger Überlegung wird zusätzlich davon ausgegangen, dass sich T_{new} nicht „feindlich“ verhält und z. B. anderen Prozessen Abbruchsignale schickt. Ist dies zu befürchten, sind solche Prozesse mittels Virtualisierung in eigenen virtuellen Maschinen (*virtual machines*) zu isolieren. Die Kommunikationsmethoden dieser Arbeit lassen sich dafür zur sicheren Kommunikation zwischen virtuellen Maschinen ausbauen.

5.5.10 Echtzeitfähige Einfüge- und Löschoperationen

Bisher wurde vor allem das Zeitverhalten von Lese- und Schreiboperationen untersucht, die für die meisten Anwendungen entscheidend sind. Es wurde davon ausgegangen, dass die zugehörigen KogMo-RTDB-Objekte bei der Initialisierung und vor dem Eintritt in eine zeitkritische Hauptschleife angelegt wurden.

Es existieren jedoch Anwendungen, die es erfordern, Objekte echtzeitfähig anzulegen und zu löschen. Ein Beispiel ist eine Objekterkennung, die neu erkannte Objekte anlegen muss und für die dann eine Entscheidung zur Notbremsung getroffen werden muss.

Wie bereits in Kapitel 5.3.7 ausgeführt, verfügt die KogMo-RTDB über eine eigene Speicherverwaltung. Der anfangs zu startende KogMo-RTDB-Manager-Prozess alloziert in seiner Initialisierungsphase nicht-echtzeitfähig beim Standardbetriebssystem Linux einen ausreichend großen Speicherblock und sorgt dafür, dass sofort physikalischer Speicher bereitgestellt wird und nicht erst bei einem späteren durch *copy-on-write* ausgelösten Seitenfehler, wie in Abschnitt 5.4.8 erläutert. Der Speicher wird zudem gegen Auslagerung schützt.

Verbinden sich andere Programme mit der KogMo-RTDB vor Abschluss der Initialisierung, blockiert die Funktion `kogmo_rtdb_connect()` bis zum Verbindungsaufbau. In ihrer Startphase sollten alle Programme blockierende Funktionen zur initialen Objektsuche (`searchinfo_wait()`) und dem erstmaligen Lesen (`readdata_waitnext()`) einsetzen. Diese Maßnahmen erleichtern den Systemstart, da sich die richtige Startreihenfolge automatisch einstellt.

Für das Anlegen- und Löschen von Objekten wurde eine ähnliche Methode wie in Abb. 5.14 erstellt. Für das Objektmanagement werden ebenfalls Mutexes eingesetzt, sodass sichergestellt ist, dass bei konkurrierenden Transaktionen die Prioritäten analog vererbt werden und die benötigten Zeiten deterministisch bleiben.

5.6 Nicht-echtzeitfähiges Funktionsentwicklungssystem

Der Schwerpunkt der in dieser Arbeit vorgestellten Architektur ist die harte Echtzeitfähigkeit. Um die in Kapitel 5.5.9 bewiesenen und in Kapitel 6.1 gemessenen Zeiten einzuhalten, sind zwingend alle in Kapitel 5.4 beschriebenen echtzeitfähigen Systemkomponenten notwendig. Wie in Abschnitt 5.5 gezeigt, können zudem auch nicht-echtzeitfähige Softwaremodule eingebunden werden.

In Projekten ohne harte Echtzeitanforderungen und zur Funktionsentwicklung ist es nützlich, die KogMo-RTDB genauso auf Standard-Systemen ohne Echtzeiterweiterung zu nutzen. Dort kann sie als leichtgewichtiges aber leistungsfähiges Framework zur Integration verschiedener Softwaremodule eingesetzt werden.

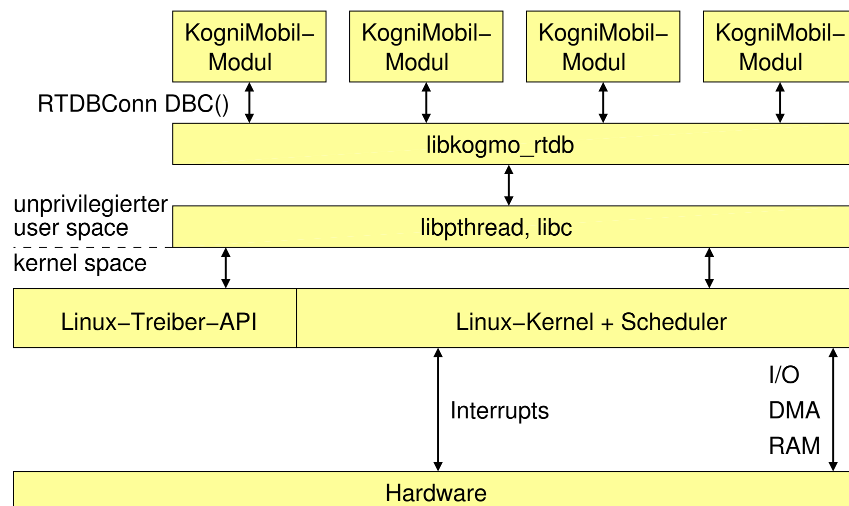


Abbildung 5.15: Version der Softwarearchitektur für Standard-Linux

Aus diesem Grund wurde die KogMo-RTDB so ausgelegt, dass sie mit einer minimalen Anpassung auch auf dem Standardbetriebssystem Linux ohne Erweiterungen lauffähig ist. Abb. 5.15 zeigt diese Architektur. Im Vergleich mit der hart-echtzeitfähigen Version

aus Abb. 5.8 ist zu erkennen, dass ein unmodifiziertes Standard-Linux eingesetzt wird und die dynamische Bibliothek `libkogmo_rtdb` anstelle `libkogmo_rtdb_rt` eingesetzt wird. Die Applikationsschnittstellen sind identisch und sogar binärkompatibel wie in Abschnitt 5.3.8 ausgeführt.

Dieses Design ermöglicht es, eine Applikation auf einem Simulationsrechner gegen `libkogmo_rtdb` zu linken und dann auf einem binärkompatiblen Fahrzeugrechner mit `libkogmo_rtdb_rt` auszuführen. Hält diese die Regeln aus Abschnitt 5.4.8 ein, kann sie sogar Teil einer hart echtzeitfähigen Regelschleife sein. Visualisierungsmodule werden dadurch zwar nicht echtzeitfähig, doch entfällt ein u.U. zeitintensives Neuübersetzen. Wie in Kapitel 5.5.8 erläutert, wird unter solchen Softwaremodulen für die Dauer von RTDB-Funktionsaufrufen in einen hart echtzeitfähigen Kontext geschaltet, um keine Prioritätsinversion zu riskieren oder Echtzeitbedingungen zu verletzen.

Um die erwünschte Austauschbarkeit von Softwaremodulen zwischen verschiedenen Simulations- und Echtzeitsystemen im Feld sicherzustellen, wurde für den SFB/TR 28 ein *Referenzsystem* spezifiziert [78]. Diese zentrale Referenzinstallation legt alle Programm- und Bibliotheksversionen zur Vermeidung von Versionskonflikten bei der Integration fest. Es ist allen beteiligten Forschern über das Internet zugänglich und dient als gemeinsames Integrations- und Testsystem. Es wird zudem für gemeinschaftliche Simulationen, oft auch mit dem Einsatz mehrerer KogMo-RTDBs für jedes einzelne simulierte Fahrzeug [57], sowie als zentrales Archiv für Aufzeichnungen verwendet.

5.7 Datenaufzeichnung und Simulation

Die zentrale Datenhaltung der in Abb. 5.1 gezeigten Softwarearchitektur der KogMo-RTDB eröffnet die Möglichkeit einer einheitlichen Protokollierung aller Daten. Zur KogMo-RTDB gehören daher ein Aufzeichnungs- und ein Wiedergabeprogramm (*RTDB-Recorder* und *Player*). Alle sichtbaren Änderungen in der KogMo-RTDB können vom RTDB-Recorder einschließlich des genauen Zeitpunkts in eine Datei mitprotokolliert werden. Dies umfasst alle Einfüge-, Schreib- und Löschoperationen, nicht jedoch die Leseoperationen, da diese an den Daten keine Veränderungen verursachen.

Der aufgezeichnete Datenstrom kann später in eine beliebige KogMo-RTDB wieder abgespielt werden, entweder in Echtzeit oder mit einer wählbaren Geschwindigkeit. Verbundene Prozesse sehen alle aufgezeichneten Ereignisse dort wieder ablaufen. Ohne explizit in der KogMo-RTDB nach Hinweisen einer Aufzeichnung zu suchen, finden die Prozesse in den Objekten keinen Unterschied zwischen der Wiedergabe einer Aufzeichnung im Labor und dem Betrieb im kognitiven Automobil. Dies ermöglicht es, ein Programm ohne Neuübersetzung vor dem realen Einsatz mit aufgezeichneten oder vollständig simulierten Daten [215] zu testen.

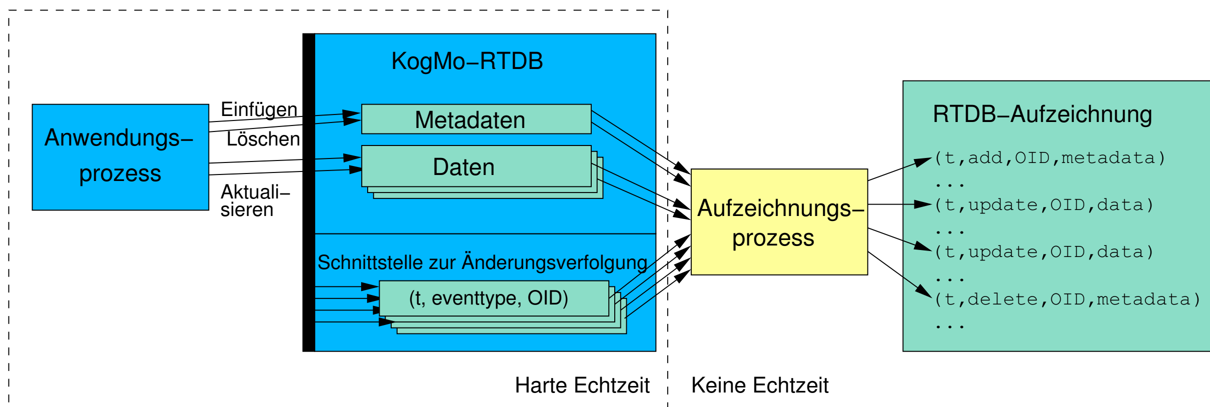


Abbildung 5.16: Aufzeichnung mittels Änderungsverfolgung

Tabelle 5.4: Belegung des Ereignistupels EV

Ereignis	**Betroffende Daten**	t_{event}	*eventtype*	*OID*
Objekt erstellt	Metadaten	$t_{created}$	*CREATED*	neue *OID*
Objekt aktualisiert	Objektdaten	$t_{committed}$	*UPDATED*	aktuelle *OID*
Objekt gelöscht	Metadaten	$t_{deleted}$	*DELETED*	alte *OID*

5.7.1 Aufzeichnungsmethode

Um dem Konzept der KogMo-RTDB zu entsprechen, darf eine Aufzeichnung unter keinen Umständen die Echtzeitfähigkeit der Prozesse, deren Transaktionen protokolliert werden, stören. Dies schließt die einfache Lösung aus, die Protokollierung im Kontext des verursachenden Prozesses durchzuführen. Stattdessen ist eine schlanke Schnittstelle zur Änderungsverfolgung notwendig, sodass die zur Protokollierung notwendigen echtzeitschädlichen blockierenden Festplattenzugriffe im Kontext eines dedizierten Aufzeichnungsprozesses durchgeführt werden, der außerhalb der kritischen Regelkreise angesiedelt ist.

Um diese Prämissen zu erfüllen, wurde die in Abb. 5.16 gezeigte Schnittstelle zur Änderungsverfolgung (*trace tap*) entworfen. Durch konsequente Nutzung der in Kapitel 5.3.4 vorgestellten Historie minimiert sie den Rechenaufwand der protokollierten Prozesse. Bei der Nutzung der KogMo-RTDB-Methoden wird lediglich das die jeweilige Transaktion identifizierende Ereignistupel $EV = (t_{event}, eventtype, OID)$ in einen Ereignispuffer ausgekoppelt. Die Art des Ereignisses bestimmt die Belegung des Ereignistupels nach Tabelle 5.4. Durch die Beschränkung auf diesen Ereignistupel EV ist die Erzeugungsdauer der Änderungsnachricht unabhängig von der individuellen Objektgröße n_{bytes}.

In Abhängigkeit von *eventtype* holt sich der Aufzeichnungsprozess die zum Zeitpunkt t_{event} aktuellen Daten bzw. Metadaten des Objektes OID mit den Standardmethoden der KogMo-RTDB aus deren Historienspeicher.

Der Vorteil dieses Vorgehens ist die Reduzierung unnötigen Kopieraufwands für Objektdaten. Der Nachteil ist, dass dadurch Daten verloren gehen können, wenn die Zeitspanne zwischen der Protokollierung des Ereignisses t_{event} und dem Abrufen durch den Aufzeichnungsprozess $t_{read,recorder}$ größer als die gültige Historienlänge $T_{history}$ ist. Daher sollte $T_{history}$ nicht zu kurz (z.B. ≥ 3 s) gewählt werden.

Da in den meisten Anwendungen die Aktualisierung von Objekten periodisch geschieht, bedeutet ein Datenverlust durch $t_{read,recorder} - t_{event} > T_{history}$ bei der Aufzeichnung, dass die Bandbreite des verwendeten Festplattensystems zu gering dimensioniert ist. Langfristig ist daher eine verlustlose Aufzeichnung auch durch längere Pufferung nicht möglich. Spätestens, wenn die Festplattenschreibpuffer vollgelaufen sind, blockiert das Betriebssystem den Aufzeichnungsprozess.

Wichtiger ist es daher, a posteriori ein Maß für die Fehlerhaftigkeit einer Aufzeichnung angeben zu können. Nur so lassen sich fehlerfreie Aufzeichnungen zur Qualitätssicherung automatisch aus einem Pool herausfiltern. Der KogMo-RTDB-Aufzeichnungsprozess schreibt daher pro verlorenem Ereignis eine kurze Fehlerkennung in die Aufzeichnung. Dies erhöht zwar das Datenaufkommen geringfügig, doch selbst wenn eine Aufzeichnung nur noch aus Fehlerkennungen besteht, würde auch ein Herausfiltern der kurzen Kennungen nicht die nötige Bandbreite für die eigentlichen Datenprotokollierungen freisetzen.

In der aktuellen Implementierung werden für die interne Aufzeichnungsschnittstelle Nachrichtenwarteschlangen (*POSIX Message-Queues*) des Realzeitbetriebssystems verwendet, die Verwendung eines RTDB-Objektes ist ebenfalls möglich. Innerhalb des RTOS sind bei gleichzeitigem Zugriff zwar kurze deterministische Blockierungen möglich, die jedoch ebenfalls durch Prioritätsvererbung korrekt gelöst werden. Zum Abruf der Nachrichten wechselt der Aufzeichnungsprozess wie in Kapitel 5.5.8 gezeigt in den primären Modus, beim Systemaufruf für den Dateizugriff wieder in den sekundären Modus. Um den Fehler einer überlaufenden Nachrichtenwarteschlange zuverlässig zu detektieren, wird dieser durch eine Fehlernachricht signalisiert, die durch eine höhere Priorität an den Anfang der Warteschlange gesetzt wird. In einer dauerhaften Überlaufsituation ist nicht mehr sichergestellt, dass jede Fehlermeldung protokolliert wird. Jedoch lassen sich derartige Aufzeichnungen gehäufter Fehler durch die gezeigten Fehlerkennungen identifizieren und leicht aussortieren.

5.7.2 Aufzeichnungsformat

Ein aufgezeichneter Datenstrom benötigt eine wohldefinierte Struktur, die flexibel und erweiterbar sein muss. Sie darf sich aber nach ihrer Etablierung nicht mehr ändern, sonst werden alte Aufzeichnungen wertlos. Um die Akzeptanz des KogMo-RTDB-Aufzeichnungsformats zu fördern, wurde das bekannte *AVI*-Multimediaformat (*audio video interleave*) so erweitert, dass andere AVI-Wiedergabeprogramme die im KogMo-RTDB-Datenstrom enthaltenen Videobildobjekte als Videofilm ohne vorherige Konvertierung abspielen können.

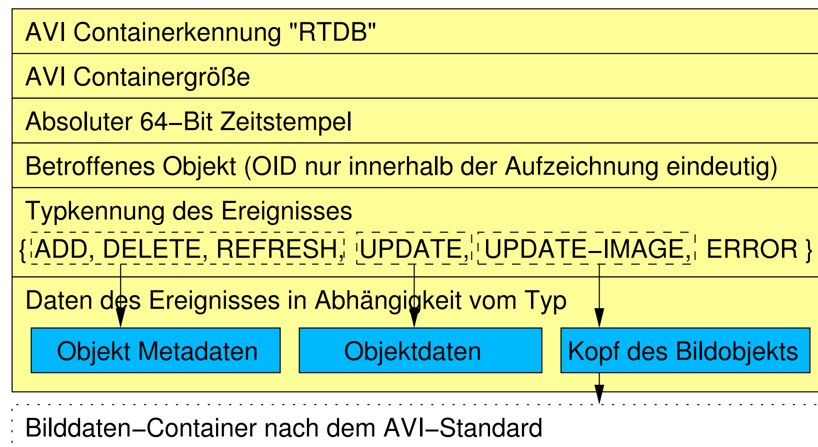


Abbildung 5.17: AVI-kompatible Struktur eines aufgezeichneten RTDB-Ereignisses

Da gängige Videocodecs eine verlustbehaftete Kompression einsetzen und dies die Ergebnisse der meisten Bildverarbeitungsalgorithmen beeinträchtigt, werden alle Bilder unkomprimiert aufgezeichnet. Um bei der Aufzeichnung den Prozessor nicht unnötig zu belasten, wurde ein schneller Rohformat-Kodierer entwickelt, der die Anzahl notwendiger Kopieroperationen minimiert. So wird die Summe der Video- und sonstiger Objektdatenströme, die gleichzeitig aufgezeichnet werden können, auf dem Fahrzeugrechner nur durch den Durchsatz der Festplatten limitiert.

Wenn eine Aufzeichnung mindestens ein Videoobjekt enthält, wird ein Dateivorspann geschrieben, der bis auf die Dateigröße dem AVI-Standard entspricht. Die Dateigröße kann erst am Ende einer Aufzeichnung berechnet werden. Dann muss zurück zum Vorspann gesprungen werden und die korrekte Größe eingetragen werden. Da eine Aufzeichnung als kontinuierlicher Strom betrachtet wird, der für einen noch höheren Durchsatz auch direkt auf RAID-Festplatten ohne Dateisystem geschrieben werden kann, ist ein nachträgliches Ändern einmal geschriebener Daten nicht in jedem Fall möglich.

Ist ein AVI-Wiedergabeprogramm auf die korrekte Größe angewiesen, kann diese im nachhinein durch ein KogMo-RTDB-AVI-Werkzeug berechnet und eingetragen werden. Dabei wird zugleich ein passender AVI-Index erstellt, der Standard-AVI-Software die Navigation vereinfacht.

Ein aufzuzeichnendes Ereignis der KogMo-RTDB wird in einen AVI-kompatiblen Container verpackt. Die gewählte Kennung (*four character code*, *fcc*) „RTDB“ wird von der meisten AVI-Videoabspielsoftware wie beabsichtigt ignoriert. Um volle AVI-Kompatibilität herzustellen, kann man dies auch als „JUNK“ kennzeichnen. Abb. 5.17 zeigt die Struktur dieses Containers.

Jedes Element enthält den in Kapitel 5.3.3 eingeführten absoluten 64 Bit Zeitstempel, der je nach verwendeter Rechnerhardware eine Genauigkeit bis zu einer Nanosekunde abbilden kann, die so für eine spätere Auswertung erhalten bleibt.

Der verbleibende Inhalt des Containers enthält die Typenkennung des aufgezeichneten Ereignisses und davon abhängig die folgenden Daten:

- Wenn ein neues Objekt angelegt wurde, werden die neuen Objekt-Metadaten angehängt.
- Wenn ein bestehendes Objekt gelöscht wurde, werden erneut die Objekt-Metadaten angehängt. Diese enthalten nun die Kennung des löschenden Prozesses $PID_{deleted}$ und den Löschzeitpunkt $t_{deleted}$. Die Lebensdauer des Objekts in der KogMo-RTDB wird nach dem Löschzeitpunkt um die spezifizierte Historienlänge $T_{history}$ verlängert. Durch die konsequente Aufzeichnung der Metadaten verhält sich ein Objekt in der Wiedergabe wie bei der Aufzeichnung.
- Es kann bei der Aufzeichnung zwischenzeitlich ein erneutes Schreiben aller Metadaten- und Datenobjekte ausgelöst werden. So können mögliche Schnittpunkte gesetzt werden, nach denen der aktuelle RTDB-Zustand festgehalten ist und die Aufzeichnung somit unabhängig von der Vorgeschichte ist. Zu Beginn jeder Aufzeichnung wird automatisch ein Schnittpunkt gesetzt und der Anfangszustand festgehalten.
- Wenn ein Objekt aktualisiert wurde, wird der aktualisierte Datenblock mit der angegebenen Größe $n_{bytes,committed} \leq n_{bytes,max}$ geschrieben. Da dieser, wie aus Tabelle 5.1 ersichtlich, die Objekt-ID OID nicht enthält, wird die betroffene OID im Kopf des Containers protokolliert.
- Ein Sonderfall ist die Aufzeichnung eines Videobildes. Von normaler AVI-Software muss ein Bild des n-ten Videostroms als „*nn*db“-Container geschrieben werden. Daher wird für Videobilder nur der Kopf des Bildobjekts im „RTDB“-Container protokolliert, auf den dann die eigentlichen Bilddaten im „*nn*db“-Container folgen.

Sollten bei der Aufzeichnung Fehler auftreten, z.B. der Verlust einzelner Objekte oder Bilder, weil sie in ihrer Historie bereits überschrieben sind, wird eine Fehlermarkierung in den Aufzeichnungsstrom eingefügt. Diese hat die Form eines RTDB-Containers mit der Typenkennung „ERROR“. Wenn bekannt wird festgehalten, bei welchem Objekt der Verlust auftrat.

5.7.3 Einspielen einer Aufzeichnung

Der *RTDB-Player* kann eine Aufzeichnung wieder in eine laufende KogMo-RTDB einspielen. Dabei lässt sich durch ein *Playercontrol*-Objekt die Wiedergabe anhalten, die Geschwindigkeit regeln, eine Position anspringen und eine Endlosschleife setzen. Der aktuelle Zeitpunkt der wiedergegebenen Daten wird in einem Objekt *Playerstatus* veröffentlicht. Wurde die KogMo-RTDB mit einer Simulationsfreigabe gestartet, gleicht der RTDB-Player die RTDB-Zeit der laufenden Aufzeichnung an. Welche Objekte vom Player beigesteuert werden, ist anhand $PID_{created} = PID_{Player}$ ersichtlich.

Das Festplattensystem des wiedergebenden Rechners muss mindestens so leistungsfähig sein, wie bei der Aufnahme, andernfalls ist eine Wiedergabe des Original-Protokolls nur

verlangsamt möglich. Die KogMo-RTDB stellt ein Werkzeug zur Verfügung, um eine Aufnahme zu schneiden. Es können beliebig Objekte herausgefiltert werden und die Aufzeichnung zeitlich gekürzt werden. Eine dermaßen bandbreitenreduzierte Aufnahme lässt sich auch auf schwächeren Systemen in Echtzeit abspielen.

Eine Aufzeichnung enthält alle Objekte, die sich während der Erstellung in der KogMo-RTDB befanden, wenn sie nicht explizit auf Wunsch des Benutzers herausgefiltert wurden, um Bandbreite und Speicherplatz zu sparen. Für eine spätere Nachvollziehbarkeit ist es nützlich, eine Aufnahme so vollständig wie möglich zu machen. So ist sie unabhängig von weiteren Konfigurationsdateien, die verloren gehen oder verwechselt werden können. Daher wird empfohlen, stets folgende Objekte mitzuführen, damit die entsprechenden Informationen in jeder Aufzeichnung enthalten sind:

GPS Position: Die genaue Position ermöglich es, die Bewegungen eines Fahrzeugs auf einer Karte nachzuvollziehen und durch eine geographische Suche ähnliche Aufzeichnungen derselben Straße oder Gegend zu finden. Anhand der ebenfalls enthaltenen Zeit können geospezifische Umgebungsbedingungen wie Jahreszeit oder Sonnenstand berechnet werden, z.B. um Beleuchtungsbedingungen in der Bildverarbeitung zu berücksichtigen.

Einbauorte und Konfiguration von Sensoren: Das Basispaket der KogMo-RTDB enthält ein 6-DoF (*six degrees of freedom*) Objekt, das benutzt werden sollte, um den genauen Einbauort einschließlich der Orientierung eines jeden Sensors zu dokumentieren. So lässt sich die genaue Geometrie später beim Abspielen nachvollziehen und ohne separate Konfiguration von neuen Algorithmen nutzen. Verschiedene Einbauorte können bewertet werden. Es ist empfehlenswert, alle verfügbaren Parameter eines Sensors in ein eigenes Konfigurationsobjekt zu schreiben, für einen Videosensor sind das z.B. die Kalibration und die Belichtungsdauer [163].

Laufende Prozesse: Für jeden mit der KogMo-RTDB verbundenen Prozess wird automatisch ein Prozess-Objekt erstellt, das mit jedem Zyklus des Prozesses aktualisiert wird. Wenn es nicht absichtlich herausgefiltert wird, enthält so jede Aufzeichnung Informationen über die Aktivität der Prozesse. Diese Daten können z.B. zur nachträglichen Analyse von Ausführungszeiten genutzt werden, wie es in Kapitel 6.3.2 gezeigt wird.

Eine Sonderbehandlung erfährt die Identifikationsnummer OID eines jeden Objekts bei der Wiedergabe: Da sie stets eindeutig sein muss und eine OID bei einer bereits laufenden KogMo-RTDB schon vergeben sein kann, werden sie beim Einspielen einer Aufzeichnung neu vergeben. Die aufgezeichneten OIDs werden anhand einer dynamischen Tabelle übersetzt, sodass bestehende Eltern-Kind-Beziehungen im Szenenbaum beibehalten werden.

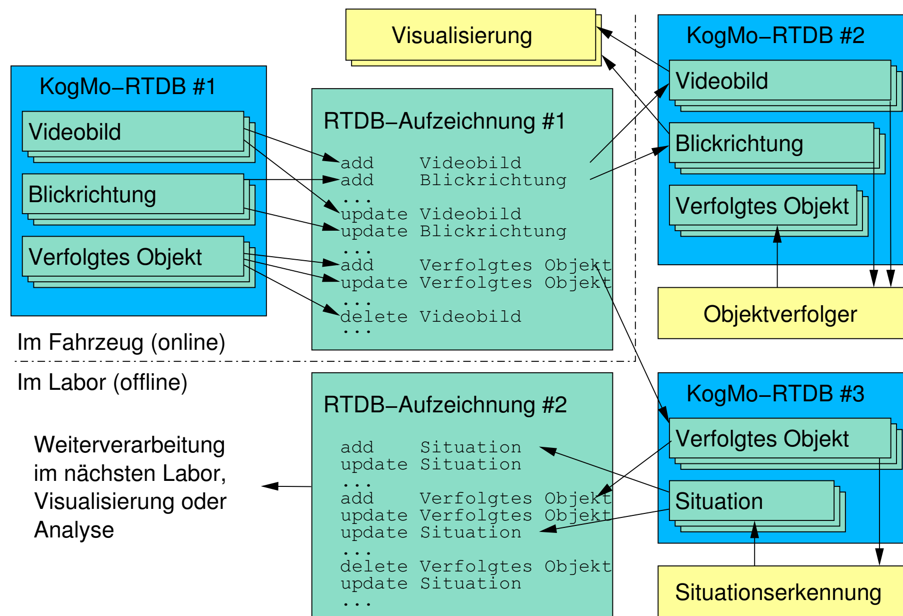


Abbildung 5.18: Aufzeichnungen und Wiedergabe von RTDB-Protokollen

5.7.4 Einsatz von Aufzeichnungen zur Simulation

Aufzeichnungen können zum einen zur Dokumentation von Testfahren eingesetzt werden. Zum anderen dienen Sie zur Generierung von Testdatensätzen, anhand deren im Labor neue Algorithmen getestet und bewertet werden. Abb. 5.18 zeigt eine erste KogMo-RTDB (#1), die auf dem Fahrzeugrechner eines kognitiven Automobils läuft. Aus ihr werden z.B. die Rohdaten aller Videokameras einschließlich der dazugehörigen aktuellen Kalibrierdaten der aktiven Kameraplattform aufgezeichnet. Die erkannten Objekte eines mitlaufenden Objektverfolgers werden ebenfalls protokolliert.

Später im Labor wird die Aufzeichnung auf einem anderen Rechner in eine zweite RTDB #2 abgespielt. Dabei werden jedoch erkannte Objekte herausgefiltert und stattdessen die Bildrohdaten von einer verbesserten Software analysiert. Das Ergebnis kann leicht mit den aufgezeichneten Resultaten verglichen werden. Eine automatische Evaluierung mit verschiedenen Parametern ist ebenfalls durchführbar.

Abweichende Anforderungen hat beispielsweise eine Forschungseinrichtung, die sich mit Situationserkennung beschäftigt. Abb. 5.18 zeigt, wie dort in RTDB #3 nur die für sie interessanten Objekte, wie z.B. die verfolgten Fahrzeuge wiedergegeben werden. Ein Algorithmus zur Situationserkennung wird offline anhand eingespielter Situationen trainiert. Dessen Ergebnisse werden nun wiederum in einer zweiten Aufzeichnung festgehalten, die nun an die nächste Forschungsgruppe weitergeben wird.

Damit solche KogMo-RTDB Aufzeichnung ihren maximalen Nutzen entfalten können, sollten folgende Designempfehlungen berücksichtigt werden:

Abtrennung der Visualisierungsmodule: Wie bereits in Abschnitt 5.4.8 zur Abtrennung nicht-echtzeitfähiger Prozesse empfohlen, sollten sämtliche Visualisierungselemente als eigenständige Module realisiert werden, die ihre Daten ausschließlich über die KogMo-RTDB beziehen. Um die in einer bestehenden Aufzeichnung enthaltenen Daten zu zeigen, muss diese lediglich in eine laufende KogMo-RTDB wiedergegeben werden, mit der die existierenden Visualisierungsmodule verbunden sind, wie in Abb. 5.18 angedeutet. Diese müssen dafür nicht modifiziert werden.

Austausch von Aufzeichnungen über zentrales Datenverzeichnis: Es bietet sich an, nützliche und gute Aufzeichnungen zum Austausch unter Projektpartnern an zentraler Stelle abzulegen. Im SFB/TR 28 dient hierzu das Referenzsystem aus Kapitel 5.6. Es ist geplant, diese Austauschplattform für andere Forschergruppen zu öffnen. Unter Zuhilfenahme des KogMo-RTDB-Schneidewerkzeugs können aus einer Aufzeichnung die relevanten Teile vor dem Herunterladen herausgeschnitten werden.

Zentrale Hinterlegung der Objektdefinitionen: Die Inhalte der einzelnen Objekte sind unter den Rahmenbedingungen von Kapitel 5.3.1 beliebig vom Benutzer festlegbar. Um Einheitlichkeit zu erreichen, hat jede Objektdefinition eine zugehörige Typenkennung TID, die nicht doppelt vergeben werden darf. Idealerweise hat jedes Projekt einen eigenen Wertebereich, über den es nach Belieben verfügen darf. Alle Definitionen sollten in einem gemeinsamen Quellcodeverzeichnis abgelegt werden, das unter einer Versionsverwaltung wie z.B. Subversion [160] steht. Die Versionsnummer der verwendeten Objektdefinitionen wird zur besseren Nachvollziehbarkeit in jeder Aufzeichnung notiert.

Gerade die gesamtheitliche Aufzeichnungsfunktion hat sich für viele Projekte auch über den Automobilbereich hinaus [217, 122] als nützlich erwiesen.

6 Messergebnisse und Anwendungen

In diesem Kapitel werden zum einen Messergebnisse der vorgestellten Hard- und Softwarearchitektur geliefert und ausgewertet. Anhand der Ergebnisse werden dann die Echtzeiteigenschaften quantitativ untersucht. Zum anderen werden ausgewählte Anwendungen präsentiert, die den Einsatz in verschiedenen Fahrzeugen demonstrieren. Abschließend wird die Übertragbarkeit in weitere Forschungsbereiche gezeigt.

6.1 Ausführungszeiten von KogMo-RTDB-Operationen

Die folgenden Messungen wurden auf dem Fahrzeugrechner [69] eines Versuchsträgers des SFB/TR 28, einem mit Sensorik und Aktorik ausgerüsteten Audi Q7 [71], durchgeführt. Der Rechner besitzt zwei Zweikern-Prozessoren AMD Opteron 275HE mit je 2.2 GHz und vier GB Hauptspeicher (DDR-400 RAM) auf einem Tyan Thunder K8WE2 (S2895) Mainboard. Eine ausführliche Spezifikation findet sich in [69].

Um die Echtzeiteigenschaften der entworfenen Architektur zu bewerten, wurden die Ausführungszeiten der wichtigsten KogMo-RTDB-Operationen vermessen. Dabei wurde die KogMo-RTDB mit einem Speicherblock von $1.4 \cdot 10^7$ Bytes und maximal 1000 Objekten konfiguriert. Wenn nicht abweichend angegeben, hat das Test-Objekt eine Größe von 152 Bytes.

Die durchgeführten Messungen enthalten auch Laufzeitverlängerungen durch Störungen durch das verwendete Betriebssystem sowie andere Prozesse und Hardwareeinflüsse. Zum Vergleich wurden alle Messungen auch an dem nicht-echtzeitfähigen Funktionsentwicklungssystem aus Abschnitt 5.6 vorgenommen. Um die Wirksamkeit der entworfenen Echtzeitarchitektur zu zeigen, wurden die Versuche zudem in zwei Konfigurationen durchgeführt:

„Unbelastet“: In der unbelasteten Konfiguration wird nur das zu vermessende Softwaremodul gestartet. Jedoch wird keine Anstrengung unternommen, dieses zu isolieren. Daher laufen gleichzeitig weitere Programme wie Systemdienste und eine graphische Benutzeroberfläche.

„Stark ausgelastet“: In der stark ausgelasteten Konfiguration laufen auf dem System eine Auswahl von Programmen, die dazu dienen, eine starke E/A-, Speicher- und Prozessorlast durch Standard-Tasks zu erzeugen. Dies ließe sich bis zur Unbenutzbarkeit durch Standard-Tasks und damit Unbedienbarkeit des Systems steigern.

Tabelle 6.1: Ausführungszeiten (μs) in nicht-Echtzeit-Konfiguration

KogMo-RTDB-Operation	**Unbelastet**			**Stark ausgelastet**		
	Mittel.	Min.	Max.	Mittel.	Min.	Max.
Objekt lesen	**4.6**	4	16	17.0	4	10721
Objekt schreiben	**8.3**	5	51	22.6	5	181681
Objekt einfügen	**60.5**	53	154	122.5	39	93109
Objekt löschen	**5.2**	4	28	18.5	4	41250
Benachrichtigung	**29.6**	23	241	169.8	20	36129

Tabelle 6.2: Ausführungszeiten (μs) in Echtzeit-Konfiguration

KogMo-RTDB-Operation	**Unbelastet**			**Stark ausgelastet**		
	Mittel.	Min.	Max.	Mittel.	Min.	Max.
Objekt lesen	5.2	4	18	16.8	4	**62**
Objekt schreiben	10.3	8	28	25.6	6	**134**
Objekt einfügen	45.3	38	77	75.6	38	**273**
Objekt löschen	9.0	8	20	18.5	8	**131**
Benachrichtigung	31.6	27	96	66.5	21	**208**

Tabelle 6.1 zeigt die Messergebnisse des nicht-echtzeitfähigen Funktionsentwicklungssystems (NRT) mit der Architektur aus Abb. 5.15, Tabelle 6.2 die des echtzeitfähigen Fahrzeugsystems (RT) aus Abb. 5.8, beides auf derselben Rechnerhardware. Es sind jeweils das Minimum, das Maximum und der Mittelwert der Messungen angegeben.

Es ist zu erkennen, dass die gemessene maximale Ausführungszeit für eine Leseoperation bei NRT unter starker Last um mehr als einen Faktor 10^3 ansteigt, wohingegen bei RT der Faktor unter 10 liegt. Da für Standard-Tasks die Laufzeiten beliebig lang werden können, zeigt sich hier die Notwendigkeit der entworfenen Echtzeitarchitektur.

Der Einfluss der im Zeitscheibenscheduling gleichzeitig laufenden Standardtasks bewirkt, dass sich die Zeiten auch in der Echtzeitarchitektur etwas verschlechtern. Diese verursachen Cacheverdrängungen und beanspruchen Speicherbandbreite. Sind hier bessere Zeiten gewünscht, muss man wie in [68, 199, 93] durchgeführt, einzelne Prozessoren exklusiv für Echtzeittasks reservieren.

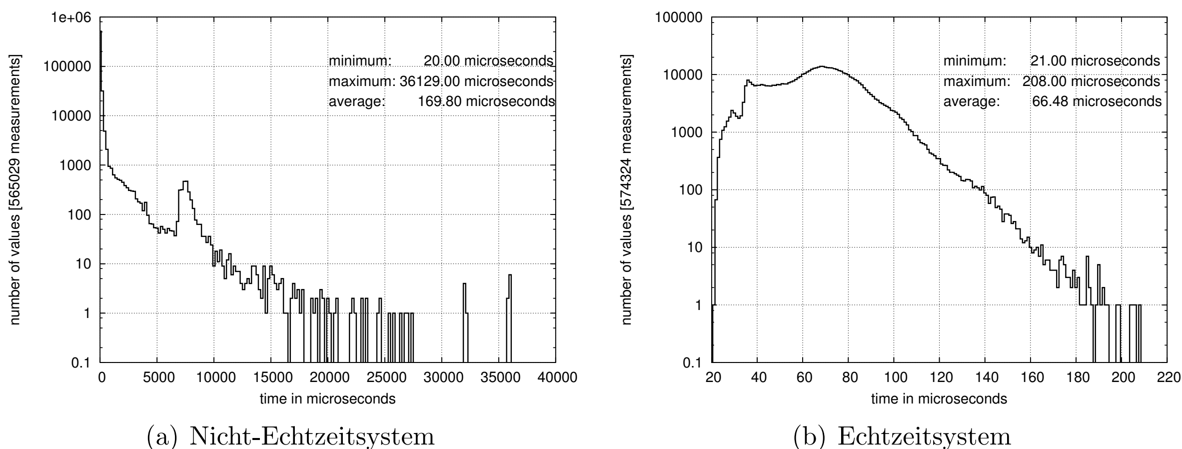
Die minimalen Ausführungszeiten in der unbelasteten Konfiguration vermitteln einen Eindruck vom niedrigen Overhead der effizienten Implementierung. Mit dem Kehrwert der durchschnittlichen Ausführungszeit errechnet man im unbelasteten Fall bei NRT eine Leistungsfähigkeit von über $1.2 \cdot 10^5$ Aktualisierungs- und $2.1 \cdot 10^5$ Abfrageoperationen pro Sekunde.

Es fällt außerdem auf, dass bis auf die Lesezeiten, die bestmöglichen minimalen Zeiten bei NRT kleiner sind. Dies liegt an der Realisierung der Mutexoperationen. Wie in Abschnitt 5.5.6 erläutert, ist bei RT jedes Mal ein Systemaufruf notwendig, wohingegen bei NRT durch die Verwendungen von *Futexes* (*fast userspace mutex*) [52] im wartefreien Fall der Betriebssystemaufruf entfällt. Lesezugriffe sind durch den Algorithmus aus Kapi-

tel 5.3.5 grundsätzlich frei von Systemaufrufen und möglichen Blockierungen, daher sind dort die minimalen Zeiten identisch.

6.1.1 Latenzzeit der Interprozesskommunikation

Des Weiteren wurde die Benachrichtigungslatenzzeit gemessen. Es ist die Zeit zwischen dem Aktualisieren eines Objekts durch ein erstes Softwaremodul und der Benachrichtigung eines wartenden zweiten Moduls einschließlich des Lesens der Objektdaten. Dies stellt gleichzeitig die Latenzzeit t_{IPC} beim Einsatz der KogMo-RTDB zur Interprozesskommunikation (*Inter-Process Communication*, IPC) dar.



(a) Nicht-Echtzeitsystem

(b) Echtzeitsystem

Abbildung 6.1: Benachrichtigungslatenzzeiten (IPC) bei starker Systemauslastung

Die Benachrichtigungszeiten beim Nicht-Echtzeitsystem (RT) können der letzten Zeile von Tabelle 6.1 entnommen werden, die genaue Verteilung zeigt Abb. 6.1(a). Die Latenzzeiten des Echtzeitsystems (RT) finden sich in Tabelle 6.2 und Abb. 6.1(b). Die Verteilungen zeigen deutlich den Kontrast zwischen NRT und RT. Es ist zu erkennen, dass die entwickelte Echtzeitarchitektur auch bei hoher Last durch Standardtasks die Kommunikation zwischen zeitkritischen Echtzeittasks weiterhin sicherstellt. Zudem wird erneut der niedrige Overhead durch die effiziente Implementierung sichtbar.

6.1.2 Abhängigkeit von der Objektgröße

Die Standardtransaktionen der KogMo-RTDB kopieren beim Lesen und Schreiben die Objekte von bzw. in den Speicher des aufrufenden Prozesses. Beim Lesen kann so in der Verifikationsphase aus Abb. 5.4 die Datenkonsistenz gewährleistet werden. Beim Schreiben wird während der Kopierphasen mit Schreibsperre aus Abb. 5.3 eine Prioritätsinversion durch undeterministischen Programmfluss verhindert. Daraus ergibt sich nach (5.7) und (5.23) eine Abhängigkeit der Ausführungszeiten von der Objektgröße.

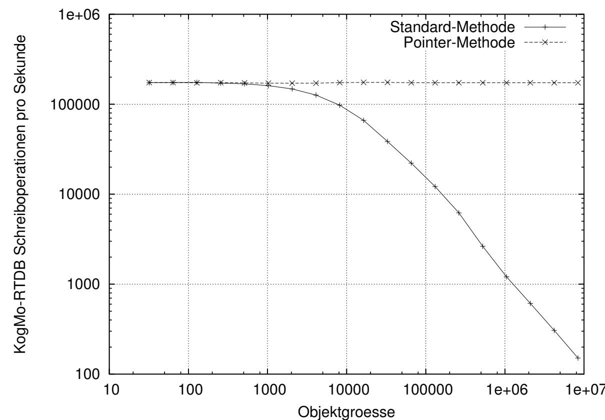


Abbildung 6.2: Ausführungsrate einer KogMo-RTDB Schreiboperation

Da die einzelne Messung, wie in Abb. 6.1 zu sehen, stark streut und wenig über den erreichbaren Datendurchsatz aussagt, wurden die Auswirkungen der Objektgröße anhand der Ausführungsrate von KogMo-RTDB Methoden ermittelt. Abbildung 6.2 zeigt das Messergebnis für Schreiboperationen bei Variation der Objektgröße. Es ist zu sehen, dass bis zu einem Umfang von 10^3 Bytes die Datenorganisation maßgeblich die Ausführungszeit bestimmt, und erst ab ca. 10^4 Bytes die Kopierzeit $c_{copy,\mathcal{D}}$ dominant wird.

Für solche „großen" Datenobjekte stellt die KogMo-RTDB Pointer-Methoden zum direkten Zugriff auf den Objektdatenspeicher bereit. So entfällt die Zeit für das Kopieren. Beim Lesen entsteht allerdings der Nachteil, dass, wenn bei einem zu kleinen Ringpuffer der Lesebereich von einem schreibenden Prozess wieder überschrieben wird, noch während der Leseprozess daraus liest, dieser es nicht bemerkt. Lesende Prozesse müssen am Ende ihren Lese-Pointer erneut auf Gültigkeit überprüfen. Zur Einhaltung seiner Zeitbedingungen muss daher für den Leseprozess $c_{read,proc}$ in (5.6) durch c_i aus (5.17) ersetzt werden.

Direkte Schreibpointer sind nur bei nicht-öffentlich beschreibbaren ($write_allow = 0$) Objekten einsetzbar, da wie in Abschnitt 5.5.5 dargelegt wurde, zum Erhalt des Determinismus keine Blockierungen bei der Ausführung von nicht-RTDB-Programmcode erlaubt sind. Größer als 10^4 Bytes und zugleich zeitkritisch sind nach Tabelle 6.5 lediglich Objekte mit Sensorrohdaten wie Videobildern, die ausschließlich von einem Hardware-Schnittstellenprozess geschrieben werden.

6.1.3 Latenzzeit des Rechnersystems einschließlich Ein/Ausgabe

In Kapitel 5.4.1 wurde betont, dass das *gesamte System* mit allen Hard- und Softwarekomponenten betrachtet werden muss. Maßgeblich für diese Arbeit ist davon die *Reaktionszeit des Rechnersystems* t_{pcsys}, die Abschnitt 5.4.2 als die Zeit zwischen dem eingehenden Hardwaresignal und dem korrespondierenden ausgehenden Hardwaresignal definiert. Zur Messung dieser Zeit dient der Testaufbau in Abb. 6.3. Er unterscheidet sich von Abb. 5.7 vor allem darin, dass die Aktoren und Sensoren auf jeweils ihre serielle Schnitt-

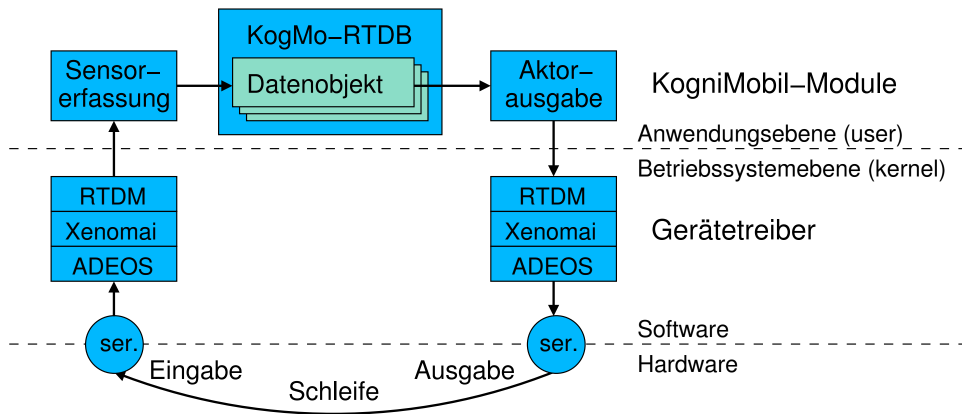


Abbildung 6.3: Messung der gesamten Systemlatenzzeit einschließlich Ein-/Ausgabe

stelle („ser.“) reduziert wurden und das technische System durch eine beide verbindende Schleife (Nullmodem-Kabel) ersetzt wurde. Der gewählte Messkreis bezieht somit Hardware, Gerätetreiber und Betriebssystem mit ein. Die ermittelten Latenzen treten genauso im Fahrzeugrechner auf, der seine Daten vom LIDAR-Sensor ebenfalls seriell bekommt und seine Kommandos über CAN an die Fahrzeugaktorik schickt.

Eine Messung beginnt im Modul „Aktorausgabe“ mit der seriellen Ausgabe eines Test-Zeichens. Dieses durchläuft nun die Echtzeiterweiterung Xenomai und den RTDM-Echtzeittreiber (vgl. Abschnitt 5.4.4 und 5.4.7), bis es über die echtzeitfähige serielle Hardwareschnittstelle ausgegeben wird (vgl. 5.4.6). Über die Verbindungsschleife trifft das Signal an der zweiten Schnittstelle als Sensor-Eingabe ein und löst einen Interrupt aus. Dieser wird von ADEOS an Xenomai und RTDM weitergereicht, wodurch das wartende Modul „Sensorerfassung“ aufgeweckt wird und das es empfangene Zeichen bekommt.

Die Latenzzeit bis zu diesem Punkt wird mit $t_{I/O-Loop}$ bezeichnet. Sie umfasst die E/A-Schleife einschließlich des zweimaligen Durchlaufs der E/A-Subsysteme (Sende- und Empfangsrichtung). Im zweiten Teil der Messung schreibt die „Sensorerfassung“ die empfangenen Daten in ein RTDB-Objekt. Über die neuen Objektdaten wird nun die „Aktorausgabe“ benachrichtigt und der Kreis schließt sich. Die Zeit t_{IPC} beinhaltet diese RTDB-Interprozesskommunikationslatenz. Die Messungen wurden auf einem älteren Rechnersystem mit zwei 2 GHz Opteron 246 Singlecore-Prozessoren vorgenommen und können Tabelle 6.3 entnommen werden. Die Übertragung des Test-Bytes bei 115200 Bit/s benötigt einschließlich Start- und Stopbit 86.8 μs, sodass die Interruptlatenz $t_{IRQ} < 100\mu$s beträgt.

Tabelle 6.3: Latenzzeiten des Echtzeitsystems mit Ein-/Ausgabe in (μs)

KogMo-RTDB-Operation	**Unbelastet**			**Stark ausgelastet**		
	Mittel.	Min.	Max.	Mittel.	Min.	Max.
2 × E/A+RTOS+Treiber ($t_{I/O-Loop}$)	**122.4**	119	154	150.9	119	**184**
RTDB-Interprozesskomm. (t_{IPC})	**30.0**	28	47	44.7	28	**74**
Gesamt t_{pcsys}	**152.4**	147	201	195.6	147	**258**

Tabelle 6.4: Aufgezeichnete Datenobjekte einer Testfahrt (Auszug)

Zeit	ID	Objektname	Objekttyp	Objektgröße
2007-01-31 20:00:52.704761000	57	`camera_left`	0xA20030	307248 bytes
2007-01-31 20:00:52.705145000	61	`c3_vehiclestatus`	0xC10201	88 bytes
2007-01-31 20:00:52.705749000	64	`c3_gpsdata`	0xC30021	124 bytes
2007-01-31 20:00:52.706050000	61	`c3_vehiclestatus`	0xC10201	88 bytes
2007-01-31 20:00:52.707036000	61	`c3_vehiclestatus`	0xC10201	88 bytes
2007-01-31 20:00:52.708024000	61	`c3_vehiclestatus`	0xC10201	88 bytes
2007-01-31 20:00:52.708723000	59	`a2_eigenspur`	0xA20101	184 bytes
2007-01-31 20:00:52.709012000	61	`c3_vehiclestatus`	0xC10201	88 bytes
⋮				
2007-01-31 20:00:52.713077000	62	`c3_vehiclecommand`	0xC10101	96 bytes
⋮				
2007-01-31 20:00:52.738004000	57	`camera_left`	0xA20030	307248 bytes
⋮				
2007-01-31 20:00:52.741996000	59	`a2_eigenspur`	0xA20101	184 bytes
⋮				
2007-01-31 20:00:52.753101000	62	`c3_vehiclecommand`	0xC10101	96 bytes

6.2 Datenaufzeichnung

Sämtliche in der KogMo-RTDB vorhanden Objekte können mit dem dazugehörigen RTDB-Recorder einschließlich ihres Verlaufs, wie in Abschnitt 5.7 beschrieben, auf einem Massenspeicher aufgezeichnet werden. Durch das erweiterte AVI-Format können die Videodatenströme mit einem normalen AVI-Videoplayer abgespielt werden. Für die Wiedergabe aller Daten muss die Aufzeichnung mit dem RTDB-Player in eine laufende RTDB eingespielt werden. Dann erscheinen die Objekte mit ihrer zeitlichen Entwicklung wieder in der RTDB.

Das Protokoll einer Aufzeichnung zu einer Testfahrt, wie es auch beim Wiedereinspielen erscheint, zeigt Tabelle 6.4. Darin ist zu erkennen, dass die linke Fahrzeugkamera alle 33 ms ein neues Bild liefert. Dieses wird ca. 4-5 ms von dem Fahrspurtracker aus [150] verarbeitet, der dann ein Spur-Objekt schreibt. Vom Fahrzeugschnittstellenmodul kommen mit 1 kHz aktuelle Fahrzeugdaten. Der Fahrzeugregler arbeitet mit einer festen Periode von 40 Hz und errechnet neue Sollwerte, die er in ein Kommandoobjekt schreibt, dessen Daten an die Fahrzeugaktorik weitergegeben werden.

6.2.1 Aufzeichnungsbandbreite

Tabelle 6.5 enthält eine Liste gängiger Objekte in kognitiven Automobilen. Es ist zu sehen, dass jedes Objekt seine eigene Größe und gewöhnliche Aktualisierungsfrequenz $\frac{1}{t_{cycle}}$ hat, und die KogMo-RTDB problemlos Größen $n_{bytes} < 100$ Bytes sowie > 1 MB verwalten

kann. Für kleine Objekte wird eine Historie von $T_{history}$ =10 s verwendet, für größere Objekte mit Sensorrohdaten eine kürzere Zeit von 5 s.

Anzahl gibt an, wie viele Instanzen eines Objekts bei einer repräsentativen Aufzeichnung verwendet wurden. Je nach der Ausstattung der Versuchsträger sind nicht immer alle Objekte vorhanden. Damit ergibt sich für jedes Objekt der Speicherbedarf $n_{bytes,dataringbuffer}$ nach (5.10) und damit eine Gesamtgröße von ca. 200 MB, die als Eingangsgröße zur Dimensionierung der Datenbankgröße nach (5.9) dienen kann.

Alle im Versuchsträger eingesetzten Objekte, einschließlich der in Tabelle 6.5 genannten Objekte, verursachen bei der Aufzeichnung insgesamt eine Datenrate von ca. 40 MB/s, die problemlos mit dem beschriebenen Fahrzeugrechner aus Kapitel 4.3.2 auf einer „Raptor“ *SATA*-Festplatte von Western Digital mitprotokolliert wurde. In einer anderen Anwendung [217] wird von einem aufgezeichneten Datenstrom mit 47 MB/s berichtet.

Tabelle 6.5: Objekte in einer RTDB-Aufzeichnung mit resultierenden Bandbreiten

Benutzer-definierter Objekttyp (TID)	**Daten-größe** (n_{bytes}) [Bytes]	**Fre-quenz** ($\frac{1}{t_{cycle}}$) [Hz]	**An-zahl** (n)	**Band-breite** (je Obj.) [MB/s]	**Historien-länge** ($T_{history}$) [s]	**Speicher-bedarf** (je Obj.) [MB]
Fahrzeugstatus	100	1000	1	0.095	10	0.954
Fahrzeugkommando	96	25	1	0.002	10	0.023
Plattformstatus	60	100	1	0.006	10	0.057
Plattformkommando	68	2	1	0.0001	10	0.001
Kamerakalibrierung	76	100	3	0.007	10	0.072
Kamerabild (640x480)	307248	30	4	8.790	5	43.952
Erkannte Fahrspur	184	30	1	0.005	10	0.053
Spurvisualisierung	388	30	1	0.011	10	0.111
IMU/GPS-Position	328	10	1	0.003	10	0.031
2D-LIDAR (Sick)	2928	37.5	3	0.105	5	0.524
3D-LIDAR (Velodyne)	1280124	10	0	12.208	5	61.041
Prozessstatus	76	n. Fkt.			10	
Gesamt				≈35.621		≈178.827

6.3 Automatisierte Prozessanalyse

Im Modell eines idealen RTDB-Prozesses, wie es Abb. 6.4 zeigt, erfolgt sämtliche Kommunikation mit anderen Prozessen, dem Benutzer oder der Prozessperipherie über die KogMo-RTDB. Auf alle Objekte wird mit den gleichen Methoden der KogMo-RTDB zugegriffen. Der Prozess bekommt seine Eingangsdaten über ein oder mehrere Eingabeobjekte $\mathcal{I}$ (*Input*), wobei eines davon als „Trigger“-Objekt einen Verarbeitungszyklus auslöst und im Folgenden mit $\mathcal{I}_T$ bezeichnet wird. Konfigurierbare Parameter werden aus Parameterobjekten $\mathcal{C}$ (*Config*) gelesen. Am Ende eines Verarbeitungszyklus werden ein oder mehrere Ausgabeobjekte $\mathcal{O}$ (*Output*) geschrieben. Das Ausgabeobjekt, nach dessen Schreiben der

Zyklus endet, wird mit $\mathcal{O}_E$ bezeichnet. Die zur nicht-echtzeitkritischen Visualisierung bestimmten Ausgaben erfolgen über Visualisierungsobjekte $\mathcal{V}$ (*Visualization*).

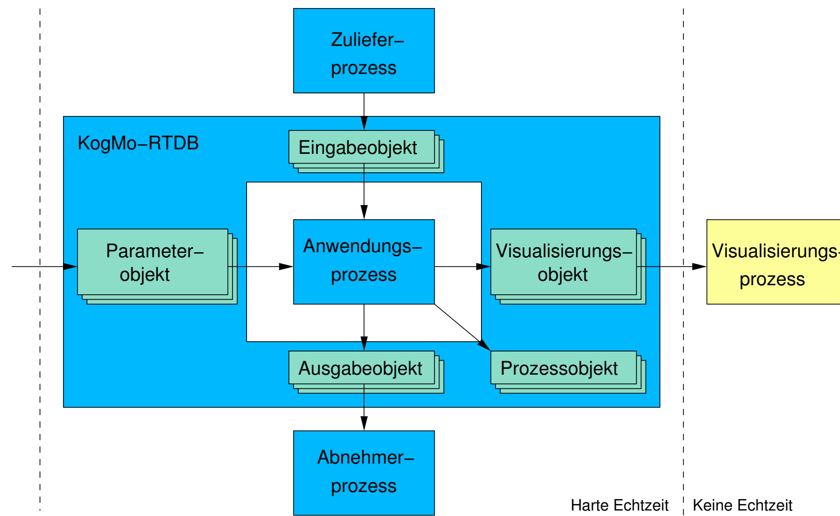


Abbildung 6.4: Modell eines KogMo-RTDB-Prozesses

6.3.1 Ressourcenbedarfsermittlung

Da in der KogMo-RTDB jedes Objekt den anlegenden Prozess als Besitzer hat, lässt sich der KogMo-RTDB-Speicherbedarf durch eine Besitzer-Suche in der RTDB und anschließende Addition der Ringspeichergrößen nach (5.10) ermitteln:

$$n_{bytes,proc} = \sum_{\substack{\mathcal{D} \in \{\mathcal{I},\mathcal{O},\mathcal{C},\mathcal{V},\mathcal{P},\dots\} \\ PID_{created,\mathcal{D}} = PID_{proc}}} n_{bytes,dataringbuffer,\mathcal{D}} \tag{6.1}$$

Gewöhnlich erstellt ein Prozess nur seine Ausgabeobjekte und wartet auf seine Eingabeobjekte, die wiederum die Ausgabeobjekte anderer Prozesse sind.

Jeder mit der KogMo-RTDB verbundene Prozess wird dort automatisch durch ein Prozessobjekt $\mathcal{P}$ abgebildet. Darin werden nach jedem Zyklus wichtige Ressourcenmessdaten wie Hauptspeicherbedarf und Prozessorzeit festgehalten, die sonst nicht anhand des RTDB-Modells erhoben werden könnten. Diese Daten lassen sich analog summieren. Die Ressourcenbedarfsanalyse kann ggf. im Nachhinein anhand einer Aufzeichnung erfolgen.

6.3.2 Laufzeitbestimmung

Für einen Echtzeitnachweis, wie er in Kapitel 5.5.9 durchgeführt wurde, ist die Rechenzeit eines jeden Tasks c_i eine unabdingbare Eingangsgröße. Ist sie unbekannt, kann sie über

die RTDB gemessen werden, wenn ein Ausführungszyklus des implementierten Prozesses durch ein KogMo-RTDB-Ereignis (*Trigger*) ausgelöst wird.

Dazu werden die Zeitstempel des Trigger-Eingangsobjekts $\mathcal{I}_T = (t_{committed,\mathcal{I}_T}, t_{data,\mathcal{I}_T}, \dots)$ und dessen zuletzt geschriebenes Ergebnisobjekt $\mathcal{O}_E = (t_{committed,\mathcal{O}_E}, t_{data,\mathcal{O}_E}, \dots)$ von einem Programm zur automatischen Laufzeitbestimmung nach folgender Methode analysiert:

1. Auf Aktualisierung von $\mathcal{O}_E$ warten.
2. Das korrespondierende $\mathcal{I}_T$ mit $t_{data,\mathcal{I}_T} = t_{data,\mathcal{O}_E}$ aus der Historie abfragen (der korrekte Transfer der Datenzeitstempel nach (5.3) wird vorausgesetzt).
3. Die Rechenzeit des Tasks c_i entspricht der Laufzeit vom Eingangsereignis bis zum Ergebnis der Verarbeitung abzüglich der Interprozesskommunikationszeit t_{IPC} aus Abschnitt 6.1.1 und errechnet sich zu:

$$c_i = t_{committed,\mathcal{O}_E} - t_{committed,\mathcal{I}_T} - t_{IPC} \tag{6.2}$$

Wird der Prozess bei der Berechnung unterbrochen, z. B. weil er nicht die höchste Priorität hat, ist das ermittelte c_i eine Überabschätzung der Rechenzeit. Seine Verwendung würde zu einer Überdimensionierung des Echtzeitsystems führen. Wird innerhalb einer Messreihe der längste Programmpfad nicht durchlaufen, ist das gemessene c_i zu klein und die Gefahr einer Deadline-Verletzung besteht. Da obige Größen im laufenden System und ohne Beeinträchtigung der untersuchten Prozesse ermittelt werden können, können mit dieser Methode anspruchsvollere Schedulingalgorithmen ihre Parameter online über die existierenden KogMo-RTDB-Schnittstellen gewinnen.

6.3.3 Echtzeitfähige Prozessüberwachung

Die zentrale Kommunikation über die KogMo-RTDB ermöglicht eine Systemüberwachung auf RTDB-Objektebene. Die überwachten Prozesse müssen dafür keine besonderen Voraussetzungen erfüllen und benötigen keine zusätzlichen Überwachungsschnittstellen (*Debugging-Ports*). Ein Überwachungsprozess auf Objektebene (*Object-Watchdog*) beobachtet seine Prozesse indirekt, indem er deren Objekte kontrolliert. Dadurch sind folgende Fehler detektierbar:

- Ausbleibende Objektaktualisierungen
- Verspätete und zeitlich unregelmäßige Daten
- Fehlende und entfernte Objekte
- Fehlende und abgestürzte Module (anhand Prozessobjekt $\mathcal{P}$)

Wie in Kapitel 5.3.3 ausgeführt, wird der Schreibzeitstempel $t_{committed}$ bei jedem Schreibzugriff von der KogMo-RTDB automatisch aktualisiert. Durch die Beobachtung der Objekte $\mathcal{O}$ und $\mathcal{P}$ lassen sich daher Aussagen über die Aktivität eines Prozesses ableiten.

Ein Prozess verletzt seine selbst vorgegebene Deadline, wenn die spezifizierte maximale Zykluszeit für $\mathcal{O}$ überschritten wird:

$$t_{cycle,max,\mathcal{O}} < t_{now} - t_{committed,\mathcal{O}} \tag{6.3}$$

Durch die Abbildung aller laufenden Softwaremodule als Prozessobjekte $\mathcal{P}_i$ können durch RTDB-Analyse die eigenen Fähigkeiten des autonomen Fahrzeugs wie in [158] abgeschätzt werden. Da gelöschte Objekte für $T_{history}$ noch in der Historie gehalten werden, ist z. B. bei Sensorstörung eine Notbremsung unter Verwendung des letzten verfügbaren Umgebungswissens realisierbar.

Eine umfangreichere Überwachung liefert die Auswertung der eigentlichen Objektinhalte. Die semantische Analyse erfordert jedoch spezialisierte Überwachungsmodule für jeden einzelnen Objekttyp TID. So kann die Korrektheit aller über die KogMo-RTDB ausgetauschter Daten geprüft werden. Die vollständige Systembeobachtbarkeit der Objektmenge $\mathcal{S}$ erfordert zudem:

$$c_{watchdog} < t_{cycle,watchdog} < \min_{\mathcal{D}_i \in \mathcal{S}} \{T_{history,\mathcal{D}_i}\} \tag{6.4}$$

Systemüberwachung im DARPA Urban Challenge Finalisten „AnnieWAY“

(a) Autonome Fahrt auf Urban Challenge(Quelle: DARPA)

(b) Rechnersystem im Kofferraum

Abbildung 6.5: Autonomer VW Passat „AnnieWAY“

Die KogMo-RTDB dient als zentrales Kommunikationsframework auf dem Rechnersystem (siehe Abb. 6.5(b)) des autonomen VW Passat der Universität Karlsruhe mit der Bezeichnung „AnnieWAY“ [101]. Das Fahrzeug (siehe Abb. 6.5(b)) wurde im SFB/TR 28 *Kognitive Automobile* (vgl. Kapitel 3.6.1) aufgebaut und verfügt über einen 360-Grad 64-Strahl 3D-LIDAR von Velodyne. Es ist ins Finale des *DARPA Urban Challenge 2007* [34] gekommen, ein Wettrennen zwischen autonomen Roboterfahrzeugen. Der Wettbewerb wurde von der Defense Advanced Research Projects Agency (*DARPA*), einer Forschungsbehörde des amerikanischen Verteidigungsministeriums, veranstaltet und fand am 3.11.2007 in Victorville, Kalifornien, USA, auf der ehemaligen George Air Force Base statt. Im Rennen

musste ein 60 Meilen langer Parcours durch die Straßen des bebauten Kasernengebiets in unter 6 Stunden abgefahren werden. Zudem mussten beim Zusammentreffen mit anderen Roboterautos oder von Menschen gelenkten Fremdfahrzeugen zudem die kalifornischen Verkehrsregeln eingehalten werden, wie z. B. das Einhalten der Reihenfolge am sog. *Four-Way-Stop* (vgl. Kapitel 2.2.1).

Im *Urban Challenge* waren alle Fahrzeuge ohne Sicherheitsfahrer unterwegs und konnten nur im Notfall von der DARPA über eine Funkfernsteuerung abgeschaltet werden. Daher hatte Sicherheit oberste Priorität und es wurde folgendes mehrstufige Sicherheitskonzept implementiert [220]:

1. Auf *Fernsteuerbefehl* (E-Stop) des von der DARPA vorgeschriebenen Omnitech DG-CSR Empfängers wird über *Relais* die elektronische Parkbremse aktiviert und die Zündung durch Auslösung der Wegfahrsperre unterbrochen.

2. Ein *Überwachungsblock* des Fahrzeugreglers auf der *dSpace Autobox* aktiviert den Bremsaktor, wenn der Fahrzeugrechner eine bestimmte Zeit *keine gültigen Daten* mehr liefert oder obiger Fernsteuerbefehl kommt.

3. Ein *Software Monitoring System* überwacht die auf dem Fahrzeugrechner laufende Software.

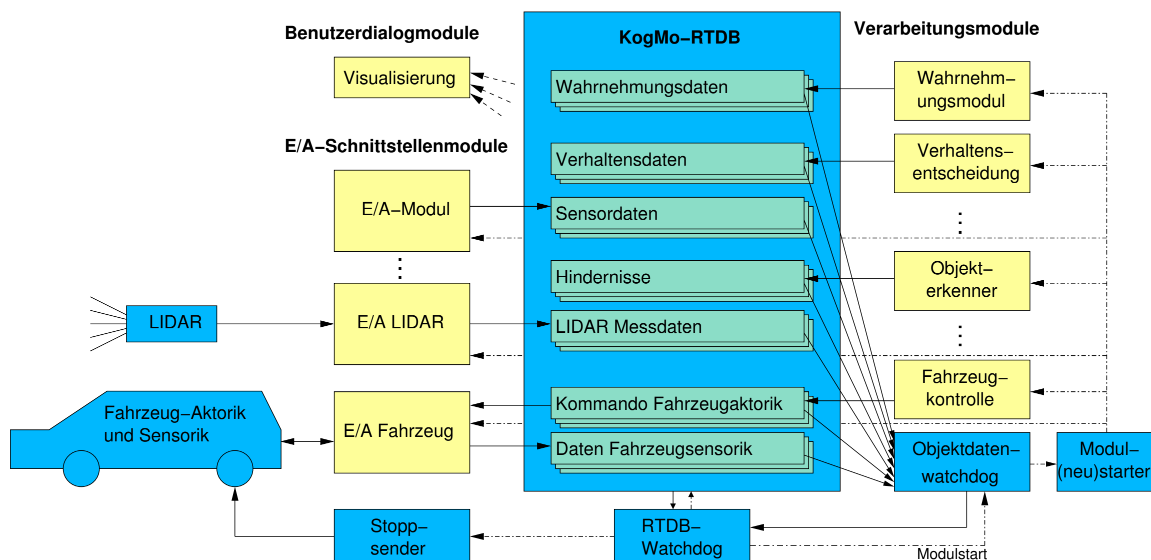


Abbildung 6.6: Systemüberwachung im DARPA-Fahrzeug AnnieWAY

Obiger E-Stop ist eine Notfallebene, die nicht zum Einsatz kommen sollte. Nach Möglichkeit sollten detektierbare Softwarefehler automatisch festgestellt werden und das Fahrzeug kontrolliert zum Stillstand kommen. Dies leistet das in Abb. 6.6 dargestellte Softwareüberwachungssystem, das sich aus folgenden Komponenten zusammensetzt:

Objektdatenüberwachung: Hier werden alle systemrelevanten Prozesse indirekt beobachtet, indem deren geschriebene Daten in der RTDB mit den Methoden aus Abschnitt 6.3.3 betrachtet werden. Sogar der Ausfall von Sensorhardware kann detek-

tiert werden, da dann der entsprechende I/O-Schnittstellenprozess sein Sensorobjekt nicht mehr aktualisiert. Hängende und abgestürzte Prozesse werden durch ein veraltetes oder gelöschtes Prozessobjekt erkannt.

Für eine semantische Analyse wären die Kenntnis häufiger Fehlerbilder und ausgiebige Testfahrten mit unverändertem Softwarestand zur Festlegung geeigneter Grenzwerte notwendig gewesen, was aus zeitlichen Gründen nicht möglich war. So hätte womöglich das zur Disqualifikation führende, zu lange Stehenbleiben am Eingang eines Parkplatzes erkannt werden können und es hätten geeignete Gegenmaßnahmen ergriffen worden können.

Prozess(neu)starter: Diese Komponente wird von der Objektdatenüberwachung benutzt, um einzelne Prozesse neu zu starten. Bei Systemstart fehlen alle Prozesse und werden daher automatisch gestartet. Dies geschieht auch bei einem Neustart, z. B. verursacht durch einen Spannungseinbruch.

Stoppsender: Damit das Fahrzeug bei größeren Schwierigkeiten, wie einem erfolglosen sofortigen Prozessneustart, sicher angehalten werden kann, sendet dieses Modul einen Stopp-Befehl unabhängig von RTDB-Modulen an die Autobox.

RTDB-Überwachung: Um auch den unwahrscheinlichen Fall eines Absturzes der Objektdatenüberwachung oder der RTDB selbst abzufangen, existiert die unabhängige RTDB-Überwachung. Sie baut periodisch neue RTDB-Verbindungen auf und prüft die Objektdatenüberwachung. Im Fehlerfall wird mithilfe des obigen Stoppsenders eine Notbremsung durchgeführt, das Softwaresystem angehalten und ein Neustart durchgeführt. Falls die Objektdatenüberwachung einen Prozess nicht neu starten kann oder dies zu häufig nötig ist, wird genauso verfahren.

6.4 Autonome Versuchsfahrten

Abbildung 6.7: Audi Q7 „Mucci“, VW Passat „AnnieWAY“, VW Touareg „MuCAR“

Die vorgestellte Architektur ist eine Kernkomponente aller im SFB/TR 28 eingesetzten Versuchsträger, wie sie Abb. 6.7 zeigt. Sie wird erfolgreich zur Integration aller kognitiven Funktionen verschiedenster Forschungsinstitute eingesetzt. Öffentliche Fahrzeugdemon-

strationen fanden u. a. auf der *DARPA Urban Challenge 2007* (vgl. Abschnitt 6.3.3) und auf der Konferenz *IEEE Intelligent Vehicles 2008* in Eindhoven, Niederlande, statt.

6.4.1 Videobasiertes Spurhalten mit bewegter Kameraplattform

(a) Frontansicht

(b) Rückansicht bei einer Demonstration auf der Konferenz „IEEE Intelligent Vehicles 2008“

Abbildung 6.8: Autonomer Versuchsträger Audi Q7 „Mucci“

Um den Einsatz der KogMo-RTDB in den autonomen Fahrzeugen zu verdeutlichen, wird hier anhand eines Beispiels das Zusammenspiel verschiedener über die KogMo-RTDB verbundener Module dargestellt. Abb. 6.9 zeigt eine Modulkonfiguration für die Funktion „videobasiertes Spurhalten mit aktiver Blickrichtungssteuerung“, wie sie auf dem Versuchsträger „Mucci“ (Audi Q7) zum Einsatz kam [77]. Der Videodatenstrom jeder Kamera wird von einem dedizierten Modul *I/O Videodaten* in Einzelbildern in der RTDB abgelegt. Auf der verwendeten Hardwarearchitektur aus Kapitel 4 ist dies problemlos mit 4 Kameras gleichzeitig möglich. Die Rohdaten der Bilder stehen so allen Modulen zur Verfügung.

Der *Fahrspurerkenner* [149] verfolgt die Fahrspur im Bild unter Verwendung eines Klothoidenmodells. Er verwendet dabei auch Fahrzeugdaten wie die Eigengeschwindigkeit. Diese liefert ein unterlagerter Fahrzeugregler auf einer dSpace Autobox über CAN und wird von dem Interfacemodul *I/O Fahrzeug* mit genauem Zeitstempel versehen in der RTDB als *Fzg.-Daten* hinterlegt.

Die *Fahrzeugkontrolle* errechnet aus den Parametern im Objekt *Fahrspur* zusammen mit den aktuellen Fahrzeugdaten aus *Fzg.-Daten* die erforderlichen Steuergrößen, die sie in das Objekt *Fzg.-Kommando* schreibt. Das Modul *I/O Fahrzeug* verfolgt jede Änderung von *Fzg.-Kommando* und sendet die neuen Vorgaben zur Autobox.

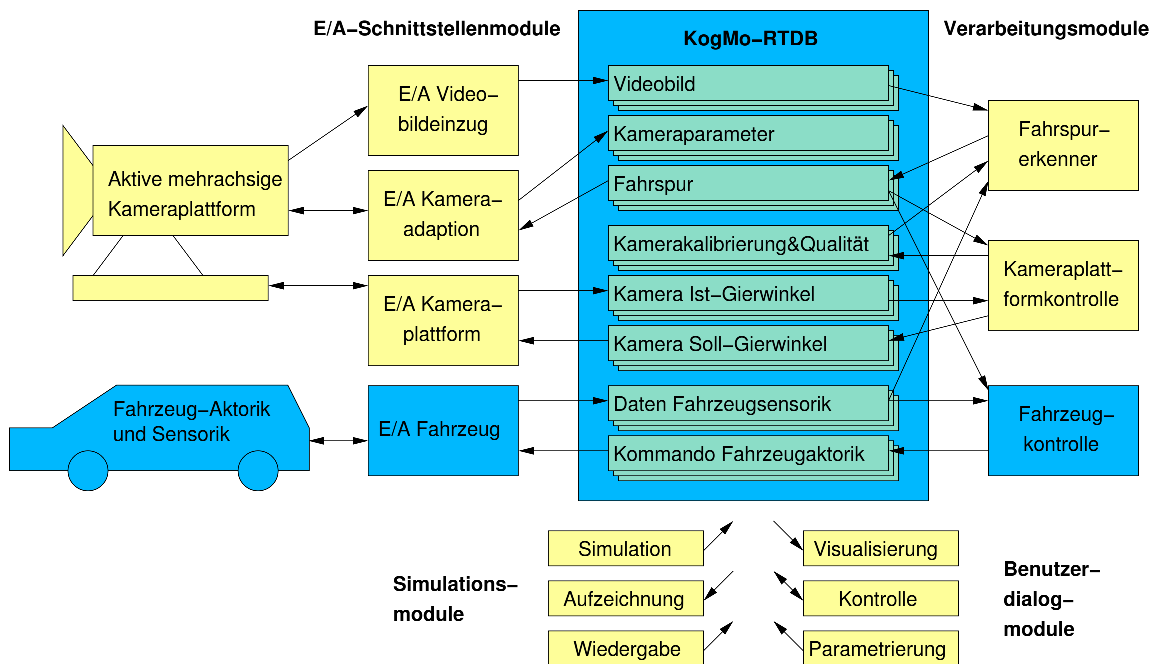


Abbildung 6.9: Modulkonfiguration für „Spurhalten mit aktiver Blickrichtungssteuerung“

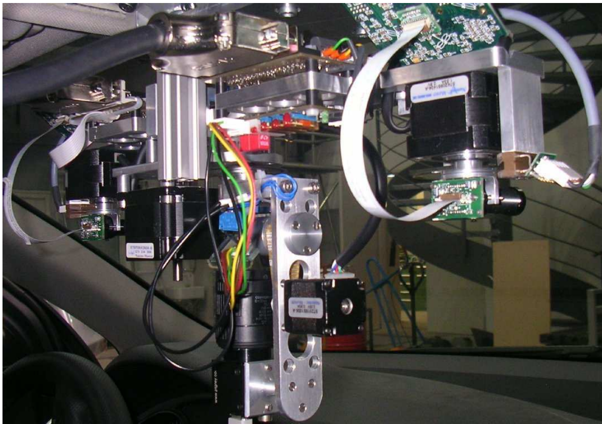
Einbindung einer bewegten Kameraplattform

Für einen weiten Sichtbereich und zur Erkennung entfernter Objekte verfügt der Versuchsträger über die in Abb. 6.10(a) gezeigte aktive Kameraplattform [31]. Sie besteht aus zwei unabhängig in der Gierachse beweglichen Weitwinkelkameras und einer zur Blickstabilisierung zusätzlich in der Nickachse beweglichen Telekamera. Ein Softwaremodul *I/O Kameraplattform* sorgt für eine transparente Anbindung: Die periodisch über CAN gemeldeten Achspositionen werden in einem RTDB-Objekt publiziert, für eine Weitwinkelkamera beispielsweise in einem der Kamera zugeordneten Objekt *Ist-Gierwinkel*. Die *Kameraplattformkontrolle* berechnet daraus aktuelle Kalibrierdaten der Weitwinkelkamera und schreibt sie mit gleichem Datenzeitstempel $t_{data,Calib.} := t_{data,Yaw}$ in das Objekt *Kamerakalibrierung*.

Der *Fahrspurerkenner* kann nun den aktuellen Kameragierwinkel aus der RTDB beziehen. Da Videobilder eine systematische Verzögerung aufweisen, muss bei jeder Abfrage der Datenzeitstempel des jeweiligen Bildes verwendet werden, um mit $t_{data,Calib.} = t_{data,Image}$ die *Kamerakalibrierung* aus der Historie der RTDB zu erhalten, die zum Zeitpunkt der Bildentstehung aktuell war.

Um die Kamera bei einer Kurvenfahrt nachzuführen, liest die *Kameraplattformkontrolle* die Krümmung der aktuellen *Fahrspur* mit und gibt bei Bedarf eine entsprechende Kamerabewegung vor, indem sie einen neuen *Soll-Gierwinkel* schreibt.

Die Nutzung der in der RTDB enthaltenen Kurzzeithistorie der Objekte verdeutlicht folgende Beispielanwendung: Da die Videobilder während eines Kameraschwenks verschmie-



(a) Kameraplattform ViSKA [31]

(b) Fahrzeugsicht mit Visualisierung

Abbildung 6.10: „Spurhalten mit aktiver Blickrichtungssteuerung“ des Audi Q7 „Mucci“

ren können, ist es sinnvoll, diese für die kurze Zeit der Bewegung nicht zu verwenden. Wenn zur Feststellung der Bewegung eine Analyse der vorhandenen RTDB-Historie der Kamerabewegungsdaten durchgeführt wird, ist der Implementierungsaufwand minimal und keine neuen Schnittstellen notwendig. Dazu wird unter Verwendung der gespeicherten *Ist-Gierwinkel* $\psi_{ist}(t)$ eine Größe *Kamera-Ruhezeit* $t_{idle}(t_0)$ zum Zeitpunkt t_0 mit

$$t_{idle}(t_0) = t_0 - t_{-n} \text{ wenn } \psi_{ist}(t_0) = \psi_{ist}(t_{-1}) = \cdots = \psi_{ist}(t_{-n}) \neq \psi_{ist}(t_{-n-1})$$

definiert. Diese kann von der *Kameraplattformkontrolle* zu jedem Zeitpunkt t_0 nur aus den Daten in der RTDB berechnet werden. Daraus wird ein Qualitätsmaß $q(t)$ abgeleitet und einschließlich einer Beruhigungszeit von 50ms exemplarisch definiert zu

$$q(t) = \begin{cases} 0.95 & \text{für } t_{idle}(t) >= 50\text{ms} \\ 0.05 & \text{sonst} \end{cases}$$

Unter Verwendung von $q(t)$ wurde der Fahrspurerkenner so erweitert, dass er minderwertige Bilder mit z. B. $q(t) < 0.5$ nicht auswertet, aber die vergangene Zeit bei der Mitführung seiner Modelle berücksichtigt. Abb. 6.11 zeigt den Verlauf der genannten Größen beim Gieren einer Weitwinkelkamera von 0 auf 10 Grad.

Erweiterung um eine adaptive Belichtungssteuerung

Das folgende Beispiel zeigt, wie mit geringem Aufwand in der vorgestellten Architektur neue Funktionalitäten unter Verwendung vorhandener Daten hinzugefügt werden können: Zusätzlich zur Fahrspur liefert der *Fahrspurerkenner* die zwei Objekte *Fahrspurpositionen* und *Messfenster*. Diese „Annotationsobjekte“ sind mit dem Videobildobjekt als Vaterobjekt verknüpft und enthalten eine Struktur variabler Länge mit einer Liste von Grafikprimitiven wie Liniensegmenten und ihren genauen Positionen im Videobild. Sie können mit

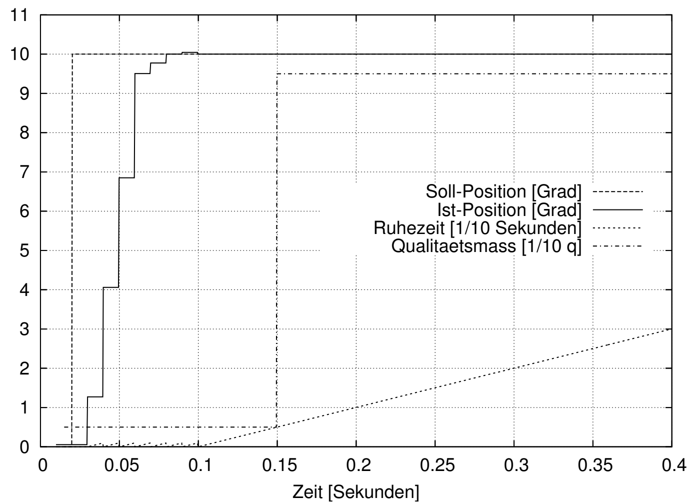


Abbildung 6.11: Kamera-Gierbewegung mit Qualitätsmaß berechnet aus RTDB-Historie

dem entsprechenden Visualisierungsprogramm ausgewählt und in das dargestellte Videobild eingeblendet werden.

Die Positionen der Spurmarkierungen werden nun von einem Modul *I/O Kameraadaption* genutzt, das die Belichtungsregelung der Kameras so parametrisiert, dass nur für die Erkennung relevante Bereiche im Bild in die Regelung eingehen [163]. Die resultierenden Einstellungen und Belichtungswerte der Kamera werden anschließend in einem eigenen Objekt *Kameraparameter* publiziert und können, wie oben gezeigt, ebenfalls in ein Qualitätsmaß eingehen.

6.4.2 Echtzeitfähige automatische Notbremsung

In der Einleitung zu Kapitel 3.1 wurde als Beispiel eine Fahrzeugabstandsregelung angeführt, die in Abb. 3.1 rechts skizziert wurde. Es wird nun das Szenario betrachtet, dass ein stehendes, $w_{obj} = 0.3$ m breites Objekt (z. B. ein Mensch) auf der Fahrbahn auftaucht, auf das eine Notbremsung durchgeführt werden muss. Dieses echtzeitkritische Experiment wurde ebenfalls erfolgreich mit der hier vorgestellten Architektur durchgeführt.

Zuvor soll nun exemplarisch eine Deadline d_{env} für (5.11) errechnet werden:

- Der *Worst-Case* bei der Abstandsregelung auf einen Sicherheitsabstand x_{safety} ist ein neu entdecktes stehendes Objekt im Abstand x_{obj} mit $v_{obj} = 0$ m/s.
- Der *verbleibende Abstand* zum Objekt bei der nun folgenden (Not-)Bremsung teilt sich auf in den vorgegebenen Sicherheitsabstand, den physikalisch bedingten Bremsweg und den zurückgelegten Weg bis zur Reaktion des Echtzeitsystems:

$$x_{obj} = x_{safety} + x_{brake} + x_{reaction} \tag{6.5}$$

- Der Abstand x_{obj} des Objekts bei der erstmaligen Erfassung ist abhängig von der *Detektionsreichweite* des abstandsgebenden Sensors: Im Audi Q7 „Mucci“ werden u. a. Sick LMS291 LIDAR-Sensoren eingesetzt, die eine Winkelauflösung von $\alpha_{res} = 0.5°$ bei $\alpha_{view} = 180°$ Sichtbereich bieten und eine maximale Reichweite von $x_{range} = 80$ m aufweisen. Ein LIDAR-Sensor sendet diskrete Laserstrahlen in einem spezifischen Abstand α_{res} aus. So ergibt sich für die Mindestanzahl an Reflektionen eines w_{obj} breiten Objekts in der Entfernung x_{obj}:

$$n_{obj} = \left\lfloor \frac{\arctan(\frac{w_{obj}}{x_{obj}})}{\alpha_{res}} \right\rfloor \tag{6.6}$$

  Die Forderung nach einer Mindestanzahl von Reflektionen $n_{obj} \geq n_{min}$ führt zu einer Erfassungsreichweite x_{detect}:

$$x_{detect} \leq \frac{w_{obj}}{\tan(n_{min} \cdot \alpha_{res})} \tag{6.7}$$

  Für $n_{min} = 2$ sind das in obigem Beispiel $x_{detect} = 17.2$ m.
  Für die früheste *Objekterfassung* ergibt sich eine Entfernung von:

$$x_{obj} = \begin{cases} x_{detect} & \text{für } x_{detect} < x_{range} \\ x_{range} & \text{sonst} \end{cases} \tag{6.8}$$

- Der Bremsweg x_{brake} für eine ideale ($|\dot{a}| = \infty$, $\mu = 1$) Bremsung mit a_{ego} von $v_0 = v_{ego}$ auf $v_1 = 0$ m/s beansprucht:

$$x_{brake} = t \cdot v_{ego} + \frac{1}{2} \cdot t_{brake}^2 \cdot a_{ego} = -\frac{v_{ego}^2}{2 \cdot a_{ego}} \text{ mit } t_{brake} = -\frac{v_{ego}}{a_{ego}} \tag{6.9}$$

  Für z. B. $v = 30\ \frac{\text{km}}{\text{h}}$ und $a = -4\ \frac{\text{m}}{\text{s}^2}$ ist $x_{brake} = 8.7$ m.

- Die verbleibende Strecke $x_{reaction}$ bestimmt die durch die Umwelt des technischen Systems vorgegebene maximale Reaktionszeit:

$$d_{env} = \frac{x_{reaction}}{v_{ego}} = \frac{1}{v_{ego}} \cdot \left(\frac{w_{obj}}{\tan(n_{min} \cdot \alpha_{res})} - x_{safety} - \frac{v_{ego}^2}{-2 \cdot a_{ego}} \right) \tag{6.10}$$

  Nach obigem Beispiel verbleiben mit $x_{safety} = 1$ m noch $d_{env} = 0.9$ s.

Das ermittelte d_{env} ist nun die gesuchte *Deadline* für (5.11). Der LIDAR-Sensor rotiert mit 37 Hz, daher wird $t_{S,max} = \frac{1}{37}$ s angesetzt, die Bremsansprechzeit wird dem Aktor zugeschlagen, daher wird $t_{A,max} = 0.3$ s angenommen und $T_i = 0$ gesetzt. Für die Hardwareplattform aus Kapitel 4 mit dem Echtzeitbetriebssystem aus Abschnitt 5.4.4 beträgt zudem die maximale Interruptlatenzzeit $t_{IRQ} = 0.1$ ms, die in $t_{V_1,max}$ eingeht. Die verbleibende Zeit begrenzt nun die maximale Rechenzeit $c_i = t_{V_i,max}$ aller an der Notbremsung beteiligten Rechenprozesse:

$$\sum_i c_i \ \leq \ d_{env} - t_{S,max} - t_{A,max} - \sum_i t_{T_i,max} \tag{6.11}$$

Nach obigem Beispiel liegt die *Deadline* der Rechenprozesse somit bei 573 ms. Dabei fällt auf, dass in der Fahrzeuganwendung die Zeiten maßgeblich durch Trägheit der großen Masse des Automobils bestimmt werden. Um so mehr müssen steuernde Prozessrechensysteme rechtzeitig reagieren, da der mögliche Schaden beträchtlich ist.

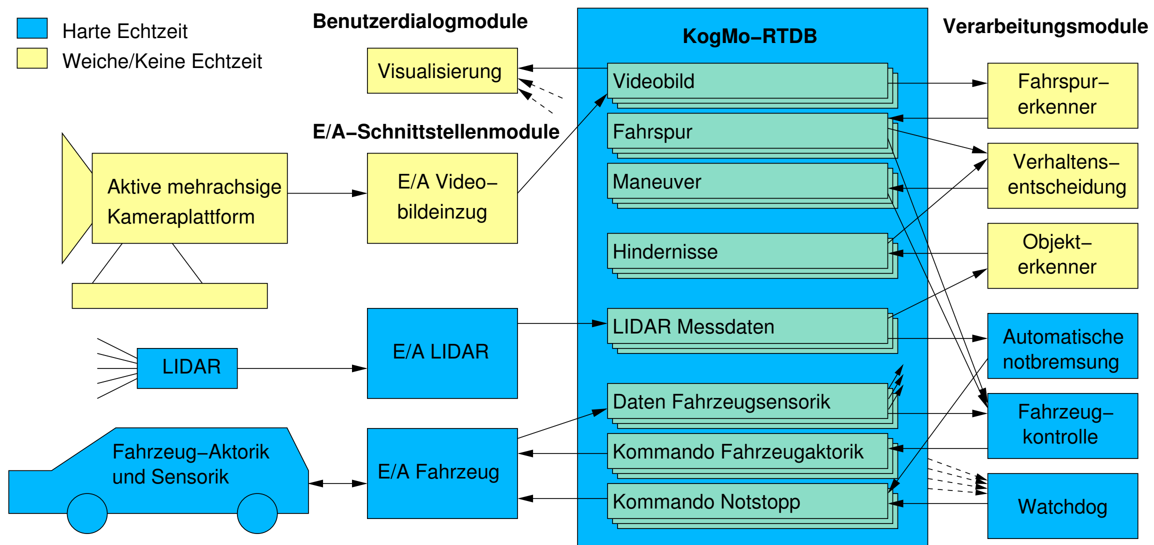


Abbildung 6.12: Modulkonfiguration für eine automatische Notbremsung

Eine Zusammenstellung der für dieses Notbremsszenario relevanten RTDB-Module zeigt Abb. 6.12. Es ist zu erkennen, dass alle für eine Notbremsentscheidung und -durchführung notwendigen Module (blau) einschließlich der KogMo-RTDB hart echtzeitfähig sind:

- Das Schnittstellenmodul *I/O LIDAR* liest kontinuierlich Abstandsdaten vom Sensor ein und stellt alle Messungen einer Umdrehung als RTDB-Objekt zur Verfügung. Dabei wird ein echtzeitfähiger RTDM-Treiber (vgl. Abschnitt 5.4.7) an einer $5 \cdot 10^5$ bit/s schnellen seriellen, über den PCI-Bus angebundenen, echtzeitfähigen Peripherieschnittstelle (vgl. Abschnitt 5.4.6) eingesetzt.

- Das Modul *Automatische Notbremsung* analysiert die erhaltenen LIDAR-Messungen und löst bei Zutreffen von (6.5) die Bremsung aus, indem es das Objekt *Notstopp* schreibt.

- Das Schnittstellenmodul *I/O Fahrzeug* ist verantwortlich für die Kommunikation mit dem Fahrzeug bzw. dem unterlagerten Fahrzeugregler (Autobox). Zur Garantie der Echtzeitfähigkeit werden hier ein RTDM-Treiber, eine PCI-Anbindung und der CAN-Bus verwendet. Bei einer RTDB-Benachrichtigung über ein aktualisiertes *Notstopp*-Objekt, wird diese Information unverzüglich via CAN weitergemeldet.

- Das *Watchdog*-Modul detektiert veraltete Datenobjekte und abgestürzte Prozesse (vgl. Abschnitt 6.3.3) und kann ebenfalls einen Notstopp auslösen.

Die dargestellte Objekterkennung dient primär der Zeitermittlung, denn die Rechenzeit des äußerst simplen Algorithmus ist minimal und der Programmpfad aufgrund seiner

Einfachheit deterministisch. Eine anspruchsvollere Objektverfolgung bietet ein richtiger LIDAR-Objekterkenner [206]. Dieser kann parallel zu diesem Sicherheitssystem laufen und interferenzfrei die gleichen LIDAR-Eingangsdaten nutzen.

6.5 Skalierbarkeit auf eingebettete Systeme

(a) Fahrzeugrechner (l.u.) im Kofferraum des Audi Q7 (b) Eingebettetes System mit ARM-Prozessor

Abbildung 6.13: Bandbreite der Einsatzgebiete für die KogMo-RTDB

Die Skalierbarkeit der entwickelten Architektur verdeutlichen die Abbildungen 6.13: Den Einbauort des in Abschnitt 4.3 vorgestellten Fahrzeugrechners zeigt Abb. 6.13(a). Die effiziente und schlanke KogMo-RTDB läuft jedoch nicht nur auf großen PCs, sondern auch auf ressourcenärmeren kleineren eingebetteten Systemen, wie dem in Abb. 6.13(b) gezeigten Singleboard-Computer. Dieser wird von Linksys als NAS-Server *NSLU2* (*Network Storage Link for USB 2.0*) verkauft. Dessen Firmware im Flash-Speicher wurde ersetzt durch ein angepasstes und neu übersetztes *SlugOS* [152] Linux-System, das wiederum auf OpenEmbedded [119] basiert, einer Linux-Distribution speziell für eingebettete Systeme.

Das Board ist mit einem Intel IXP420 XScale-Prozessor (ein 32 Bit ARMv5TE Kern) bestückt, der mit 266 MHz läuft und über 8 MB Flash und 32 MB SDRAM Speicher verfügt. Die kleinstmögliche Speicherkonfiguration für die RTDB ist 0.5 MB, der Programmcode (RISC) belegt unter 0.2 MB. Ein RTDB-Schreibzugriff dauert beispielsweise ca. 10 μs, eine Benachrichtigung benötigt ca. 300 μs, was verglichen mit Tab. 6.1 recht lange ist, da bei nur einem Prozessorkern immer ein Taskwechsel notwendig ist und der Wechsel (*Context Switch*) zeitaufwändig ist.

Mögliche Anwendungen für diese eingebettete KogMo-RTDB sind u. a. die Messdatenaufnahme und die Ausstattung nicht-autonomer „Hasenfahrzeuge“ mit Vehicle-to-Vehicle Kommunikationstechnik. Das Board verfügt über nicht-echtzeitfähige USB-Schnittstellen, über die bereits GPS-Empfänger, Kameras und WLAN-Adapter angebunden wurden. So

konnten z. B. GPS-Tracks aufgezeichnet werden und Kamerabilder eines anderen Fahrzeugs mit GPS-Position auf das Fahrzeugsystem übermittelt werden. Das System in Abb. 6.13(b) wiegt einschließlich Gehäuse, Akkus, USB-Speicher und GPS-Empfänger unter 400 g, verbraucht ca. 400 mA und konnte als portabler GPS-Logger eingesetzt werden (unter anderem bei einer 4.5 h-Bergwanderung auf den Krottenkopf).

Da auf dem ARM-Prozessor ein vollwertiges Linux-Betriebssystem läuft, konnte der Quellcode vieler Treibermodule vom Fahrzeugsystem unverändert übernommen werden. Um mit einer KogMo-RTDB auf einem x86-System Daten über Funk oder durch Aufzeichnungen auszutauschen, ist darauf zu achten, dass für Integer und Float-Zahlen die gleiche Datenrepräsentation verwendet wird, hier z. B. *Little Endian*.

6.6 Weitere Anwendungsfelder und Projekte

Die vorgestellte *Realzeitdatenbasis für kognitive Automobile* findet inzwischen weit über den SFB/TR 28 hinaus Anwendung. So wird sie am Forschungszentrum Informatik in Karlsruhe in einem für autonomes Fahren modifizierten Smart Roadster [180, 216] eingesetzt, der u. a. Hinderniserkennung [168] und Fahrspurerkennung [82] mit einer PMD-Kamera (Photomischdetektor) demonstriert.

Auch außerhalb des Fahrzeugbereichs hat sich die KogMo-RTDB als nützlich erwiesen: So wird sie beispielsweise im Exzellenzcluster „Cognition for Technical Systems“ (CoTeSys) [23] in München eingesetzt: Die erforschten kognitiven Systeme umfassen unter anderem intelligente Serviceroboter und kognitive Fabriken [11, 217, 122, 166], in denen die Zusammenarbeit von Menschen und Industrierobotern im Produktionsprozess erforscht wird. Gerade die entwickelten Aufzeichnungsmethoden haben sich für maschinelle Lernkomponenten als sehr nützlich erwiesen und die klare Definition der Schnittstellenstrukturen vereinfacht in großen Forschungsgruppen die gemeinsame Integration.

7 Zusammenfassung und Bewertung

In dieser Arbeit wurde gezeigt, dass das vorgestellte Konzept einer realzeitfähigen Hard- und Software-Architektur zur Integration kognitiver Funktionen tragfähig ist. Es wurde erfolgreich in kognitiven Automobilen und anderen Anwendungen mit großem Datenaufkommen eingesetzt. Zur Realisierung der Kernkomponente KogMo-RTDB wurden zudem weitere Ergebnisse auf den Gebieten der Kommunikation in harten Realzeitsystemen, der Realzeitdatenbanken und den Protokollen zum blockierungsfreien Datenaustausch erzielt.

Im folgenden bewertenden Vergleich mit dem Stand der Technik aus Kapitel 2 wird gezeigt, welche Beiträge diese Arbeit zu den jeweiligen Themen leistet.

7.1 Bewertung und Vergleich mit dem Stand der Technik

Die Datenorganisation der KogMo-RTDB entspricht in ihrer Art einem *Blackboard*, ein zentrales Verzeichnis, das alle Module nutzen. Die Verbesserung im Vergleich zu einem klassischen Blackboard besteht darin, dass alte Daten nicht sofort durch Überschreiben verloren gehen, sondern für eine individuell definierbare Zeit (Historie oder Episode) konserviert werden.

Die Kommunikation der Architektur gleicht dem Konzept des *Anonymous Publish/Subscribe* wie in [140]. Sie ist aber erstens nur insofern anonym, als dass der Kommunikationspartner nicht wissen *muss*, wer sein Gegenüber ist, aber sehr wohl anhand der *Prozess-Identifikationsnummer* $PID_{created}$, $PID_{committed}$ und $PID_{deleted}$ dessen Identität z. B. zu Diagnosezwecken *herausfinden kann*. Zum Zweiten muss ein Prozess nicht jede Nachricht seines Datenlieferanten auswerten, was bei stark unterschiedlichen Zykluszeiten zu einer Rechenzeitverschwendung führen würde, sondern kann jederzeit die aktuellste oder eine vergangene Nachricht abrufen.

Die Zugriffsmethoden der *Realzeitdatenbasis für kognitive Automobile* sind einer relationalen Datenbank [36] ähnlich. Als Zugriffsschlüssel dienen jedoch ausschließlich statische Metainformationen und der Zeitpunkt der dynamischen Daten. Die übrigen Daten ändern sich üblicherweise so schnell, dass es ineffizient wäre, darüber bei jeder Aktualisierung einen Index zu erstellen. Zudem würde die Indexerstellung auf der Seite des Schreibenden Zeit kosten, was dem Konzept der KogMo-RTDB entgegenläuft, die Entwickler von Software zu ermutigen, möglichst viele Informationen häufig zu publizieren.

7.1.1 Hardwarearchitektur

In Fahrzeugen wird Modularität immer noch gerne durch Hardwaremodule hergestellt. So bestehen in heutigen Serienfahrzeugen einzelne Funktionen aus separaten Steuergeräten, die durch Bussysteme [50] verbunden werden. Dieses Vorgehen hat vor allem wirtschaftliche und produktpolitische Gründe.

Es ist anzumerken, dass über Busse übertragene Informationen flüchtig sind, sodass wichtige Informationen wiederholt übertragen werden. Für die Übertragung müssen alle Daten zudem serialisiert und mit einer geeigneten Priorität versehen werden. Auch die Rechnerarchitektur vieler Versuchsträger besteht oft aus mehreren vernetzten Teilsystemen [165, 84, 212, 205].

Das Ein- und Auspacken sowie die Verwaltung von Daten kostet in jedem beteiligten Rechnersystem wertvolle Rechenzeit. Die vorgestellte Hardwarearchitektur setzt daher ein lokales Multicore-Prozessornetz („Cluster-in-a-Box") ein. Dessen einzelne Knoten besitzen zwar einen lokalen Speicher, können aber auch mit niedriger Latenz und großer Bandbreite auf entfernten Speicher anderer Knoten zugreifen [196]. Dabei entsteht dem entfernten Prozessor kein Rechenaufwand. Die vorgeschlagene Hardwarearchitektur besteht aus Standardkomponenten, deren Leistungsfähigkeit mit der technologischen Entwichlung kontinuierlich wächst und die einfach zu vervielfältigen sind.

7.1.2 Effizienz des Datenflusses

Bei der genaueren Betrachtung der Architekturen aus Kapitel 2 fällt auf, dass auch innerhalb eines Rechners beim Datenaustausch zwischen Prozessen oder Threads an vielen Stellen zusätzliche Kopien von Nachrichten angefertigt werden [140, 84, 165]. Der Grund ist zum einen die Befürchtung, Daten zu verlieren. Zum anderen dient dies der zeitlichen und organisatorischen Entkopplung einzelner Module.

Das Konzept der KogMo-RTDB bietet hier einen effizienteren Ansatz: Durch die Publikation aller Daten an zentraler Stelle sind sie jederzeit unmittelbar verfügbar, es besteht keine Notwendigkeit, Informationen redundant zu sammeln. Durch die konsequente Führung einer Historie sind auch ältere Versionen erhältlich, sogar wenn Objekte inzwischen gelöscht sind.

Bei dem in manchen Architekturen anzutreffenden Konzept der Nachrichtenwarteschlangen [140] besteht die Gefahr, dass bei einer zeitweisen Verzögerung im Programmablauf ein Modul dann auf alten Daten arbeitet. Erst nach der Aufarbeitung der vergangenen Daten, die sich durch neue Daten verlängert, erreicht es wieder die gegenwärtig relevanten Daten.

Durch die Zugriffsmethoden der KogMo-RTDB wird dafür gesorgt, dass alle Wahrnehmungsmodule stets mit den jüngsten Umweltinformationen arbeiten. Erfolgen während der Rechenzeit eines Moduls mehrere Aktualisierungen eines Datenobjekts, liefert die Lesemethode die aktuellste Information. Anhand der Zeitstempel ist das damit erfolgte

Überspringen älterer Daten sofort ersichtlich. Ältere Daten können dennoch innerhalb ihrer Historienzeitspanne nachgefragt werden. Eine Methode zur richtigen Dimensionierung der Historie liefert Kapitel 5.3.5.

Wenn Daten aktiv zwischen Prozessen verteilt werden, z. B. in Form von Kanälen, muss jeder Prozess die empfangenen Daten wieder mitnotieren [140]. Erhöht man nun die Datenrate auf einem Kanal, verursacht dies bei allen anderen Prozessen erhöhten Rechenaufwand. Bei der passiven Verteilungsmethode der KogMo-RTDB entfällt dieser Aufwand. Allerdings erfordert die Nutzung derselben Daten robuste Schutzmechanismen und durchdachte Protokolle, um Inkonsistenzen im Datenbestand zu vermeiden. Das in Kapitel 5.3.5 entwickelte blockierungsfreie Protokoll erfüllt diese Anforderungen: Es liefert eine konsistente lokale Kopie eines ausgewählten Objekts zu einem gegebenen Zeitpunkt. Alternativ ist auch der direkte Zugriff auf große Datenobjekte wie Sensorrohdaten und Bilder möglich. Hier wird die Zeit für eine lokale Kopie vollständig eingespart.

7.1.3 Zeitverhalten

Viele der untersuchten Fahrzeugarchitekturen aus Kapitel 2 setzen deutlich auf eine zyklische Verarbeitung, die meist durch einen taktgebenden Sensor getrieben wird. So haben EMS-Vision [165] aus Abschnitt 2.1.1 und ANTS [84] aus Abschnitt 2.1.2 ihre Ursprünge in der Bildverarbeitung und synchronisieren ihre Verarbeitung über den Framegrabber mit dem Videotakt. Die erforderliche Synchronisation aller Kameras wird meist durch entsprechende Hardwarevorrichtungen gelöst. Auch in einer frühen Version der KogMo-RTDB wurde die Eignung diskreter Zeitpunkte, sog. *Ticks*, untersucht [59].

Für allgemeinere Architekturen, in denen verschiedene Sensorsysteme gleichberechtigt kombiniert werden sollen, ist diese Vorgabe zu restriktiv. Daher bietet der in der KogMo-RTDB realisierte Ansatz eine freie Zeitgestaltung und die Möglichkeit, jedes Modul individuell zeitgetrieben oder ereignisgesteuert durch RTDB-Veränderungen zu betreiben. Dies lässt beliebige Mischformen auf dem gleichen System gleichzeitig zu. So wird die KogMo-RTDB für videobasierte Funktionen (Kapitel 6.4.1), für LIDAR-gestützte Systeme (Kapitel 6.3.3) und auf weiteren Anwendungsfeldern (Kapitel 6.6) erfolgreich eingesetzt.

Bei verteilten Lösungen wie EMS und ANTS ist die Synchronisation von Teildatenbanken systemimmanent, die diskreten Synchronisationsintervalle limitieren auch bei Prozessen auf einem Rechner die Kommunikationshäufigkeit und erhöhen durch jedes neue zwischengeschaltete Verarbeitungsmodul die Latenz des Gesamtsystems. Die zeitgetriebene Verarbeitung mit periodischer Überprüfung auf neue Nachrichten in der Hauptschleife von TRUCS vergrößert auch dort die Latenz.

Die taktfreie Methodik der KogMo-RTDB erlaubt hingegen durch verzugsfreie Interprozesskommunikation (vgl. Kapitel 6.1.1) die Realisierung schneller Regelkreise mit Kommunikation über RTDB-Objekte. Im Vergleich zu anderen Architekturen gibt es dabei keine Beschränkungen auf einen Prozess, die Kommunikation über die RTDB ist kein „Umweg“, der bei harten Echtzeitanforderungen gemieden werden müsste [84]. Im Gegenteil,

ein feingranularer Aufbau aus vielen Modulen mit trennbaren Teilaufgaben fördert die Modularität und vereinfacht die Fehlersuche.

Aufgrund der kurzen Ausführungszeiten von RTDB-Operationen im μs-Bereich [73] (vgl. Kapitel 6.1) bietet es sich an, auch umfangreiche Zustandsinformationen zusätzlich zu publizieren. Durch das blockierungsfreie Leseprotokoll können diese Daten „angezapft“ und zur Diagnose genutzt werden, ohne die Stabilität einer Regelschleife zu beeinträchtigen.

7.1.4 Skalierbarkeit

Der von vielen Architekturen gewählte Ansatz zur Leistungssteigerung ist die Hinzunahme von weiteren Rechnern zu einem Netzwerk. Dies ist oft aus historischen Gründen gerechtfertigt, da früher keine ausreichend leistungsfähigen Rechnersysteme erhältlich waren. Es ändert sich mit der besseren Verfügbarkeit und kontinuierlichen Weiterentwicklung von skalierbaren Multiprozessorsystemen, die aus vernetzten Rechenknoten mit Mehrkernprozessoren (*multicore-multinode*, siehe Kapitel 4.3.1) bestehen.

Multicore-Systemen stellen jedoch neue Anforderungen an die Skalierbarkeit einer Softwarearchitektur. Um eine maximale Systemausnutzung zu erreichen, ist es essentiell, gegenseitige Blockierungen zwischen Prozessen zu minimieren. Andernfalls entstehen Leistungsverluste durch unnötige Wartezeiten und das System skaliert schlecht bei einer zunehmenden Anzahl von Kernen. Techniken wie Hyper-Threading treiben die Zahl zu berücksichtigender Rechenkerne weiter nach oben. Auch das Zwischenspeichern eingegangener Daten in jedem Prozess, wie es z. B. in TRUCS passiert [140], ist nur bei wenigen Prozessen pro Rechner angemessen.

Die Methoden der KogMo-RTDB, insbesondere das blockierungsfreie Schreib-/Leseprotokoll (vgl. Kapitel 5.3.5), tragen zu einer reibungslosen Zusammenarbeit ohne Zeitverluste bei. Dass die entwickelte Architektur leichtgewichtig genug ist, um auch in umgekehrter Richtung zu skalieren, zeigt der Einsatz auf einem eingebetteten System (vgl. Abschnitt 6.5).

7.1.5 Netzwerklösungen

Das Einsatzgebiet der vorgestellten Architektur ist die Integration kognitiver Funktionen in ein räumlich begrenztes System, z. B. ein Automobil. Wie oben gezeigt, sind durch moderne Multicore-Multiprozessorsysteme keine vernetzen Rechner mehr für ein solches System unbedingt notwendig. Eine räumliche Verteilung ist im Kofferraum eines Fahrzeugs ebenfalls nicht erforderlich. Schwerpunkt dieser Arbeit war daher, eine effiziente Architektur für ein lokales System zu finden, nicht für den Einsatz auf vernetzten Rechnern.

Das Konzept der KogMo-RTDB schließt dabei eine Vernetzung aber nicht aus. Vielmehr wurden im Projekt verschiedene Ansätze zum Fernzugriff und zur Spiegelung ausgewählter Objekte von einer RTDB in eine andere RTDB entwickelt. Es war förderlich, dass die Architektur dazu keine Vorschriften macht sondern im Gegenteil verschiedene Lösungen auch gleichzeitig zulässt. Es wurden lediglich Programmbeispiele zur Übertragung von RTDB-Objekten über TCP/UDP verteilt, wie zur zyklischen Spiegelung einer gegeben Objektmenge zu Überwachungszwecken oder zur Anbindung an ein MATLAB/Simulink-Modell [76]. Die Art und Häufigkeit der Übertragung richtet sich nach dem Anwendungsfall: Für eine Fahrzeug-zu-Fahrzeug-Übertragung [147] muss mit Verlusten gerechnet werden, sodass ein Quality-of-Service Konzept notwendig wird. Für die Synchronisierung zweier RTDBs über eine Direktverbindung zur redundanten Auslegung reicht eine Priorisierung.

Für hauptsächlich verteilte Anwendungen existiert bereits eine große Anzahl etablierter Lösungen wie CORBA, ICE, ANTS, uvm. Eine Auswahl ist meist abhängig von verschiedenen Faktoren wie dem verfügbaren Übertragungsmedium (Ethernet, CAN, Firewire, SCI, Myrinet, ...), den Kosten, den Echtzeiteigenschaften, der Heterogenität und Plattformunabhängigkeit, usw. Zudem beschäftigen sich eigene Forschungsarbeiten mit Synchronisations- und Replikationsstrategien zwischen verteilten Realzeitdatenbanken [136].

Für die genannten Netzwerklösungen bietet sich die KogMo-RTDB ergänzend als echtzeitfähige Architektur für die lokale Kommunikation auf jedem Rechner an. Da die Netzwerkkommunikation als unabhängiger Zusatz wie ein normaler Prozess die Dienste der RTDB nutzt, muss sie nicht in den lokalen Echtzeitnachweis (vgl. Kapitel 5.5.9) einbezogen werden und die Kernkomponenten der RTDB bleiben schlank und deterministisch.

7.1.6 Aufzeichnungsmethoden

Im „Produktivbetrieb“ eines Fahrzeugs besteht im Grunde keine Veranlassung, das Verhalten des Systems mitzuprotokollieren. Dennoch sind umfassende Aufzeichnungsfunktionen einer Architektur für die vorangegangene Entwicklung und Diagnose äußerst hilfreich. Oft werden in vielen Architekturen Methoden zur Aufzeichnung bei der Planung nicht berücksichtigt und erst später hinzugefügt. So wurden beim Aufzeichnungssystem von TRUCS (vgl. Kapitel 2.2.2) Defizite festgestellt, die auf widersprüchliche Anforderungen zwischen der Filmerstellung und dem implementierten Kommunikationsstil zurückgeführt wurden.

Wenn Aufzeichnungsmethoden nicht systemimmanent sind, sondern z. B. Prozesse mit neuen Parametern gestartet oder bestehende Strukturen modifiziert werden müssen, ist die Anfertigung einer Aufzeichnung mit spürbarem Zusatzaufwand verbunden. Dabei beeinflusst die Aufzeichnung das zu untersuchende System, verfälscht also die gewonnenen Daten und gefährdet dessen Echtzeitfähigkeiten. Selten sind alle verfügbaren Daten in einem einheitlichen Format zusammen mit einem synchronisierten Zeitstempel zu erfassen.

Für die vorliegende Architektur wurde eine universelle Aufzeichnungsmethode geschaffen, die es erlaubt, sämtliche Daten einschließlich ihres zeitlichen Verlaufs zeitgenau mitzuprotokollieren. Dazu ist keine Änderung an bestehenden Softwaremodulen notwendig. Es wird ein strukturierter Datenstrom geschrieben, wahlweise in eine Datei oder zur Durchsatzsteigerung auch direkt ohne Dateisystem auf einen Datenträger. Der Verzicht auf mehrere Dateien verhindert, dass der Kopf der Aufzeichnungsfestplatte ununterbrochen springen muss.

Beim Wiedereinspielen von Daten kann der KogMo-RTDB-Player zur Simulation die Systemzeit aller RTDB-Prozesse an die Aufzeichnung anpassen. In der Datenbank ist jederzeit ersichtlich, welche Objekte vom Abspielprozess beigesteuert werden, bestehende Softwaremodule nutzen diese ohne Unterschied. Die Wiedergabe lässt sich beliebig kontrollieren und anhalten. Damit sind die zu TRUCS in [140] vorgeschlagenen Verbesserungen an ein Aufzeichnungssystem erfüllt.

Um das zu untersuchende System in seiner Echtzeitfähigkeit nicht zu beeinträchtigen, wurde für die KogMo-RTDB eine Aufzeichnungsmethode entwickelt, die für jede Transaktion lediglich ein identifizierendes Ereignistupel auskoppelt, dessen Erzeugungsdauer deterministisch ist. Anhand der vorhandenen RTDB-Methoden können die betroffenen Daten ohne zusätzlichen Kommunikationsaufwand direkt aus der RTDB-Historie gewonnen werden. Bei der Aufzeichnung konnten problemlos Datenraten von 40 MB/s erreicht werden [217, 75].

7.1.7 Generik der Architektur

Viele der im Stand der Technik vorgestellten Architekturen wurden zu Beginn für einen bestimmten Anwendungsfall entworfen. Wie auch bei EMS, ANTS und TRUCS war dies hier der Fahrzeugeinsatz. Bei den DARPA Grand Challenges war jeweils zu Beginn der Entwicklung die zu erfüllende Menge an Fähigkeiten bekannt, was hingegen bei Versuchsträgern, in die jederzeit neue Funktionen integrierbar sein müssen, nicht gegeben ist. Da im Transregio-SFB jedoch schon von Gründung an konkurrierende Ansätze verfolgt und verglichen werden sollen, musste dies auch die Architektur abdecken. So wurde auf der DARPA Urban Challenge LIDAR als dominanter Sensor eingesetzt, obwohl bisher hauptsächlich an videobasierten Lösungen gearbeitet wurde.

Die hier vorgestellte Architektur hat bereits weitere, über den Fahrzeugeinsatz hinausgehende Einsatzgebiete durch Dritte gefunden (vgl. Kapitel 6.6), was im praktischen Einsatz nur für wenige Architekturen aus Kapitel 2 gilt. Gerade bei der Handhabung großer Datenmengen v. a. durch mehrere Sensorsysteme ist die KogMo-RTDB gerade dabei, sich zu etablieren.

7.1.8 Flexibilität der Architektur

In der vorliegenden Architektur hat Flexibilität einen hohen Stellenwert. So kann jeder Benutzer eigene Objekte definieren, es wird nur der äußere Rahmen vorgegeben. Es wird keine strenge Programmstruktur erzwungen. Insbesondere die Hauptschleife des Programms bleibt unter der Kontrolle des Programmierers, was unter einigen Architekturen nicht zulässig ist [185, 21, 175]. Die Kommunikation ist daher so einfach zu benutzen wie Datei-Ein-/Ausgabe und ist an jeder Stelle des Programms möglich.

Im direkten Vergleich beispielsweise mit ANTS finden sich einige Punkte, an denen die KogMo-RTDB auf dem lokalen System flexibler ist. So besitzt sie eine echtzeitfähige dynamische Speicherallokation (vgl. Kapitel 5.3.7) und kann zur Laufzeit beliebige neue Objekte anlegen und alte löschen. Man kann sich jedoch auch wie ANTS darauf beschränken, Objekte grundsätzlich nur in der Initialisierungsphase anzulegen und den TLSF-Memory-Allokator durch eine einfache lineare Einmalvergabe ersetzen.

Auch die mit neuen Objekten einhergehende Vergabe eindeutiger Identifikationsnummern zur Laufzeit wurde in der KogMo-RTDB gelöst. Da dazu eine kurzzeitige Sperre benutzt wird, greift hier die entwickelte Methode zur Kooperation verschieden harter Echtzeitprozesse (vgl. Kapitel 5.5) und verhindert echtzeitschädliche Prioritätsinversionen.

7.1.9 Blockierungsfreier Datenaustausch

In Kapitel 2.4 wurden eine Reihe von Algorithmen für blockierungsfreien Datenaustausch betrachtet. Die einfachste Variante [116], die lediglich sperrenfrei ist, arbeitet mit Wiederholungen im Konfliktfall und ist für Echtzeitsysteme nur unter bestimmten Voraussetzungen einsetzbar. In wartefreien Implementierungen hingegen ist die maximal notwendige Anzahl von Programmschritten beschränkt. Dies wird erkauft durch einen hohen Speicherbedarf [90] oder bestimmte Anforderungen an den eingesetzten Scheduler [3] oder die Prozessorarchitektur [7]. Schleifenfreie Protokolle schließlich benötigen keine Wiederholungen, benötigen aber Speicher für zusätzliche Kopien und Vorwissen über das Zeitverhalten der Tasks.

Für den Fall eines Schreibprozesses wurde für die KogMo-RTDB eine schleifenfreie Variante mit einem Ringpuffer entworfen (vgl. Kapitel 5.3.5). Diese rechtfertigt ihren Speicheraufwand durch den erwiesenen Zusatznutzen als zeitlich adressierbare Historie mit dem exakten Verlauf der vergangenen Versionen. Die Protokolle aus dem Stand der Technik sehen keine Möglichkeit vor, auf diese Daten zuzugreifen. Sie setzen Zeitstempel nicht ein, um eine zeitliche Nachvollziehbarkeit herzustellen, sondern lediglich zur Erkennung eines Konflikts [25], in dem veraltete Daten gar nicht erst geschrieben werden und verloren gehen.

Die betrachteten Protokolle mit Ringpuffern benötigen zur Dimensionierung genaue a priori Informationen über alle beteiligten Tasks und verzichten auf eine Konsistenzüberprüfung beim Lesen, da diese theoretisch nicht notwendig ist [27]. In der vorliegenden

Architektur sollen jederzeit neue Tasks hinzukommen können, daher ist dieses Wissen nicht verfügbar. Stattdessen müssen für jedes Datenobjekt eine Gültigkeitszeitspanne und eine maximal erlaubte Aktualisierungsrate spezifiziert werden, um den Speicherbedarf zu dimensionieren. Da beliebig langsame Tasks nicht verboten werden, wird die Datenkonsistenz durch absolute Zeitstempel sichergestellt, die für sich genommen auch einen Zusatznutzen darstellen und in Kapitel 6.3 zur Prozessanalyse eingesetzt werden.

Für den Fall mehrerer Schreibprozesse wurde keine blockierungsfreie Lösung mit vertretbarem Speicher- und Rechenzeitaufwand und gängigen Betriebssystemanforderungen gefunden. Bei den betrachteten Protokollen muss stets a priori die Taskanzahl feststehen, da sie zur Dimensionierung der Speicherstrukturen verwendet wird. Der Ansatz, vorsorglich alle möglichen Prozesse als Schreiber vorzusehen, erzeugt einen ungerechtfertigten Speichermehraufwand für jedes Objekt. Daher werden für den eher seltenen Fall, dass für ein Datenobjekt mehrere Schreibprozesse zugelassen sind, zwischen Schreibern zur Koordination Mutexes mit Prioritätsvererbung eingesetzt und für die entworfene Architektur der Echtzeitnachweis auch bei gemischt echtzeitfähigen Prozessen erbracht. Da meist deutlich mehr Lesezugriffe stattfinden, wurden diese weiterhin schleifenfrei realisiert. Die entwickelte Architektur erlaubt jederzeit neu hinzukommende Lese- und Schreibprozesse, was insbesondere für den Einsatz als Entwicklungssystem wichtig ist. Wenn neue Prozesse eine niedrigere Priorität als alle bisherigen haben, sind aus Sicht eines prioritätsbasierten Schedulings keine Behinderungen möglich, die KogMo-RTDB schließt dann zusätzlich auch Beeinträchtigungen der bisherigen Tasks durch Kommunikation aus.

Dass die Forschung zu nicht-blockierenden Protokollen noch lange nicht abgeschlossen ist, zeigen aktuelle Arbeiten wie [20]: Das dort vorgeschlagene Protokoll zur Leser-Schreiber-Synchronisation auf Multicorearchitekturen erfordert jedoch die zeitweise Abschaltung der Präemption zur Laufzeit, was Betriebssysteme auf Benutzerebene meist nicht zulassen.

7.2 Fazit

Es wurde eine flexible Architektur zur Integration kognitiver Funktionen vorgestellt, die bereits in mehreren Projekten eingesetzt wird. Durch ihre harten Echtzeitfähigkeiten können auch stabile Regelkreise über die KogMo-RTDB geschlossen werden, wie z. B. eine Querregelung zur Spurhaltung. Der Datenaustausch und die Aufzeichnung funktionieren schritthaltend, sowohl für große Datenvolumina wie Videodaten mit 40 MB/s als auch hohe Datenraten im kHz-Bereich. Die ganzheitliche Aufzeichnung mit genauen Zeitstempeln ermöglicht eine umfassende Auswertung.

Rückblickend ist festzustellen, dass man wichtige Objekte verbindlich zu Projektbeginn hätte festlegen sollen. Die Objekte sind grundsätzlich frei definierbar, die RTDB stellt nur die Methoden zur Verwaltung und zum Austausch bereit. So wurden z. B. verschiedene Fahrspurobjekte definiert und in verschiedenen Fahrzeugen verwendet. Beim Austausch der Spurtracker wurde festgestellt, dass nicht die von anderen Modulen benötigten Objekte geschrieben wurden. Zur schnellen Abhilfe wurden kleine „Übersetzungsmodule“

programmiert, die bei jeder Änderung eines Objekts die neuen Daten umrechnen und in ein anderes Objekt schreiben. Dass dies so problemlos und mit wenigen Zeile Quellcode, der schnell im Fahrzeug geschrieben wurde, möglich war, zeigt wiederum eine Stärke der RTDB. Jedoch ist fraglich, ob die RTDB ohne diese große Freiheit bei der Objektdefinition eine so gute Akzeptanz und eine dementsprechend große Verbreitung gefunden hätte.

Die Methode zur ganzheitlichen Aufzeichnung wurde ursprünglich mit dem Schwerpunkt entwickelt, andere echtzeitkritische Module auf keinen Fall zu beeinträchtigen, und wurde daher als zentraler Bestandteil schon beim Design berücksichtigt. Durch das Interesse anderer Forschergruppen hat sich gerade die Aufzeichnungsmethode als ein zusätzliches Qualitätsmerkmal erwiesen.

7.3 Ausblick

Ansatzpunkte für weitere Arbeiten finden sich beispielsweise im Einsatz der KogMo-RTDB bis auf Betriebssystemebene. So könnten Scheduling-Algorithmen für ihre Entscheidungen Daten heranziehen, die sie online aus der KogMo-RTDB gewinnen. Die entwickelten Methoden schaffen wichtige Grundlagen: Anhand von Zeitstempeln und der Besitzerkennung von Objekten in der Historie können Rückschlüsse auf das Verhalten von Prozessen gezogen werden. Die bereits im Prozessobjekt verfügbaren Informationen ließen sich um Rechenzeit- oder Qualitätsanforderungen ergänzen. Das entwickelte blockierungsfreie Leseprotokoll kann verwendet werden, um innerhalb des Betriebssystemkerns alle in der RTDB abgelegten Daten zu untersuchen, ohne laufende Prozesse zu beeinträchtigen.

Darüber hinaus könnte die vorliegende Architektur zur Kommunikation zwischen mehreren virtuellen Maschinen eingesetzt werden. Statisch angelegte öffentlich lesbare Objekte sind sofort nutzbar, für konkurrierende Schreibzugriffe müssen die Kooperationsmethoden für die Scheduler aller Maschinen erweitert werden. Es ist weiterhin denkbar, mehrere RTDBs einzusetzen, auf die von anderen Maschinen nur Lesezugriff erlaubt wird. In ihrer aktuellen Implementierung sorgt die Architektur bisher für eine Isolation auf Zeitebene, durch die vorgeschlagene Virtualisierung wäre zusätzlich eine Isolation auf Betriebssystemebene realisierbar, z. B. um Sicherheitsanforderungen in Fahrerassistenzsystemen zu erfüllen.

Das Ziel der entwickelten Architektur ist nicht, alle anderen Architekturen zu ersetzen. Die KogMo-RTDB versteht sich vielmehr als ein Baustein, der kostenfrei als *Open-Source* erhältlich ist [67] und zum einen als schlankes Framework alleinstehend genutzt werden kann. Die Kommunikation ist darin entsprechend dem Unix-Paradigma so einfach wie die Anwendungen von Dateioperationen. Zum anderen kann die KogMo-RTDB als Integrationslösung für bestehende Frameworks verwendet werden. Darüber hinaus könnte sie selbst in vorhandene Software als echtzeitfähiger Baustein integriert werden, um z. B. die Interprozesskommunikation auf Multicore-Rechnern zu optimieren.

A Protokoll zum blockierungsfreien Datenaustausch

A.1 Ablauf des Schreibalgorithmus

Den Algorithmus der KogMo-RTDB zum Schreiben eines neuen Nutzdatensatzes $\mathcal{W}$ in den Ringspeicher eines RTDB-Objektes zeigt Abb. 5.3. Sein Ablauf wird im Folgenden kurz erläutert:

— In der **Initialisierungsphase** werden noch keine Veränderungen an den Daten der KogMo-RTDB vorgenommen:

1. Als Eingabedaten werden die Identifikationsnummer OID des Objekts und die neuen Nutzdaten $\mathcal{W}$ benötigt.
2. Anhand der gegebenen OID werden die Metadaten $\mathcal{M}_{\mathcal{D}}$ zum RTDB-Objekt $\mathcal{D}$ geholt, die nachfolgend mit $\mathcal{D}$ indiziert werden. Dabei sind vor allem folgende Parameter des Objekts und des zugehörigen Ringspeichers relevant:
   - $n_{slots,\mathcal{D}}$: Anzahl Ringspeicherplätze entsprechend Historienlänge aus (5.2)
   - $i_{current,\mathcal{D}}$: Letzter beschriebener Ringspeicherplatz, enthält die jüngsten Daten
   - $t_{cycle,min,\mathcal{D}}$: Kürzestes erlaubtes Aktualisierungsintervall
   - $n_{bytes_{max},\mathcal{D}}$: Maximale Größe eines Ringspeicherplatzes, um (5.1) zu garantieren
   - $\mathcal{D}[]$: Allozierter Speicherblock mit Ringspeichern (Struktur nach Tabelle 5.1)
   - $flags_{\mathcal{D}}$: Konfigurationsparameter:
     - $write_allow$: Öffentliches Schreibrecht anstelle nur für den besitzenden Prozess
     - $cycle_watch$: Überwachung und ggf. Beschränkung der maximalen Schreibrate auf $\frac{1}{t_{cycle,\mathcal{D}}}$
     - $no_notifies$: Keine Auslösung von Benachrichtigungssignalen an wartende Leser (in Kombination mit $write_allow = 0$ ist auch der Schreibalgorithmus frei von Systemaufrufen)
3. Bei einem fremden Objekt ohne öffentliche Schreibfreigabe wird die Annahme von der KogMo-RTDB verweigert.

4. Die Basisdaten des neuen Datenblocks $\mathcal{W}$ werden nun vorerst lokal aktualisiert.

— In der **Kopiervorbereitungsphase** wird das Objekt in der KogMo-RTDB auf neue Daten vorbereitet:

5. Wenn Schreibzugriffe von andern Prozessen erlaubt und damit möglich sind, muss ein Mutex für das Schreiben dieses Objektes $\mathcal{D}$ erlangt und gesperrt werden.
6. Wenn die Zeit seit der letzten Aktualisierung zu kurz ist, d.h. weniger als das minimale Aktualisierungsintervall beträgt, und die Zykluszeitüberwachung aktiviert ist, wird die Annahme vom Schreibalgorithmus ebenfalls verweigert.
7. Nun kann die Nummer des nächsten Ringspeicherplatzes ermittelt werden.
8. Jetzt wird der Zielspeicherplatz in der KogMo-RTDB invalidiert. Bei 64 Bit Zeitstempeln auf einem 32 Bit System muss der höherwertige Teil zuerst auf Null gesetzt werden, da nur dieser zur Validitätsprüfung herangezogen wird. Der Wertebereich der Zeitstempel wird dort minimal eingeschränkt: Mit der gewählten Zeitauflösung aus Abschnitt 5.3.3 sind dies die ersten 4.3 s des Jahres 1970.

— In der eigentlichen **Kopierphase** werden die Daten von der Anwendung in den Datenbereich der KogMo-RTDB kopiert:

9. Der neue Datenblock $\mathcal{W}$ wird auf den neuen Speicherplatz i_{next} kopiert, dabei wird $t_{committed,\mathcal{D}}[i_{next}]$ als ungültig markiert belassen. Der Kopiervorgang wird nach $n_{bytes,\mathcal{W}} \leq n_{bytes_{max},\mathcal{D}}$ Bytes beendet.

— In der **Kopierabschlussphase** wird der Zielspeicherplatz in der KogMo-RTDB für anschließende Lesezugriffe wieder freigegeben:

10. Wenn Benachrichtigungen aktiviert sind, muss der für Benachrichtigungen bestimmte Mutex des Objekts gesperrt werden.
11. Im unwahrscheinlichen Fall, dass die Zeit zwischen zwei Schreibzugriffen auf das Objekt durch die in Anschnitt 5.3.3 gewählte Zeitauflösung nicht differenziert werden kann, was einer Schreibrate von > 1 GHz entspricht, muss aus Konsistenzgründen der Schreibzeitstempel inkrementiert werden. Dieser Sonderfall muss in einer späteren Auswertung der Zeiten berücksichtigt werden.
12. Nun wird der Speicherplatz in der KogMo-RTDB mit dem neuen Zeitstempel wieder als gültig markiert. Bei 64 Bit Wortbreite auf einem 32 Bit-System muss der höherwertige Teil zuletzt geschrieben werden.
13. Abschließend wird der Index des jüngsten Speicherplatzes aktualisiert. Dies geschieht durch eine atomare Schreiboperation, die Speicherwortbreite von $i_{current,\mathcal{D}}$ darf nicht größer als die Prozessorwortbreite sein.

14. Wenn Schreibzugriffe von anderen Prozessen möglich sind, wird der zuvor gesperrte Schreibmutex dieses Objektes wieder freigeben.

— **Benachrichtigungsphase:**

15. Wenn Benachrichtigungen aktiviert sind, werden alle Prozesse aufgeweckt, die auf eine Aktualisierung dieses Objekts warten (vgl. Abschnitt 5.3.6). Dies geschieht durch die Signalisierung einer Zustandsvariable (*condition variable*, siehe Abschnitt 5.5.6) und Freigabe des Benachrichtigungsmutex.
16. Falls die Aufzeichnung aktiv ist, wird der Ereignistupel EV an den Aufzeichnungsprozess gesendet (siehe Kapitel 5.7.1).

A.2 Ablauf des Lesealgorithmus

Der dem entwickelten Protokoll entsprechende Lesealgorithmus der KogMo-RTDB für einen bestimmten Nutzdatensatz $\mathcal{R}$ ist in Abb. 5.4 gezeigt und hat folgenden Ablauf:

— In der **Initialisierungsphase** wird der gewünschte Ringspeicherplatz (*slot*) gesucht, der dem spezifizierten Zeitpunkt entspricht:

1. Als Eingabe ist zum einen die Identifikationsnummer OID des Objekts nötig, zum anderen ein Zeitpunkt t_{seek}, die Angabe $MODE$ ob sich dies auf den Schreib- oder Datenzeitpunkt bezieht und ein zugehöriger Vergleichsoperator „$\lesseqgtr$“. Damit lässt sich zudem durch wiederholten Aufruf dieser Lesemethode durch die Historie navigieren.
2. Anhand der gegebenen OID werden wie beim Schreibalgorithmus in Abb. 5.3 die Metadaten $\mathcal{M}$ zum RTDB-Objekt $\mathcal{D}$ geholt:
   - Dabei sind zusätzlich folgende Konfigurationsparameter in $flags_{\mathcal{D}}$ bedeutsam:
     - *read_deny*: Das Objekt bietet kein öffentliches Leserecht
3. Bei einem fremden Objekt ohne öffentliche Lesefreigabe wird der Lesezugriff von der KogMo-RTDB verweigert.
4. Für die nun folgende Suche wird zuerst der Speicherplatz der letzten Aktualisierung festgehalten. Nach diesem Schritt erfolgende Schreibzugriffe werden damit nicht berücksichtigt.
5. Nun wird die Nummer des Ringspeicherplatzes i_{read} mit dem gewünschten Datensatz ermittelt: Dabei wird der Ringspeicher linear rückwärts durchsucht und der Schreib- bzw. Datenzeitpunkt mit dem gegebenen Suchzeitpunkt verglichen. Je nach Suchauftrag (früher/später) wird die Suche fortgesetzt. Wird der gewünschte Zeitpunkt nicht gefunden, bricht die Suche mit der entsprechenden Fehlermeldung ab. Vor jedem Vergleich wird der gültigkeitsbestimmende eindeutige Schreibzeitstempel des betrachteten Speicherplatzes notiert und auf Validität, d. h. ungleich Null,

geprüft. Bei 64 Bit Zeitstempeln auf einem 32 Bit System darf nur der höherwertige Teil zur Validitätsprüfung herangezogen werden und muss zuerst gelesen werden.

— **Kopierphase:**

6. Der gefundene Datenblock wird in den Ergebnisbereich $\mathcal{R}$ des aufrufenden Prozesses kopiert. Dabei werden nur die ersten $n_{bytes,\mathcal{D}}[i_{read}]$ Bytes kopiert, die das Objekt mit seiner Größe zum ausgewählten Zeitpunkt umfassen.

— **Verifikationsphase:**

7. Nun wird der Zeitstempel des gefundenen Speicherplatzes mit dem notierten Zeitstempel erneut verglichen. Auf einem 32 Bit System mit 64 Bit Zeitstempeln muss der höherwertige Teil zuletzt gelesen werden. Bei Ungleichheit wurde der Platz während der Kopieroperation überschrieben (*history wrap-around*), der Ergebnisdatenblock $\mathcal{R}$ ist somit inkonsistent. Ein erneuter Lesezugriff kann diese Situation nicht beheben, da die alten Daten bereits überschrieben sind. Eine entsprechende Reaktion ist Aufgabe des lesenden Prozesses.
   Eine entsprechende Dimensionierung von $T_{history}$ nach (5.8) bei gleichzeitigem Einsatz der Schreibratenüberwachung (*cycle_watch*) hilft, dieses Problem zu vermeiden.

8. Im Erfolgsfall werden die gelesenen Nutzdaten $\mathcal{R}$ als Ergebnis geliefert.

Bei der Verwaltung der Zeitreihen wurde bewusst darauf verzichtet, Indizes oder Hash-Tabellen wie aus [112] aufzubauen. Auch in Hash-Tabellen entstehen in *Worst Case* lange Laufzeiten durch Kollisionen. Zum einen ändern sich die Daten dafür in der Regel zu schnell, sodass dieses mehr Rechenaufwand erzeugen würde. Zum anderen entstünde dieser Rechenaufwand beim Schreiben und damit im Kontext des publizierenden Prozesses. Konzept der KogMo-RTDB ist es jedoch, diesem keine „Nachteile“ entstehen zu lassen.

Die Suche lässt sich mit dem Vorwissen aus t_{cycle} an der wahrscheinlichsten Stelle starten. Im ungünstigsten Fall muss aber trotzdem mit dem vollständigen Durchsuchen des Ringspeichers gerechnet werden.

Literaturverzeichnis

[1] AMD, One AMD Place, Sunnyvale, CA 94088, USA: *AMD x86-64 Architecture Programmers's Manual Volume 1: Application Programming*, 2002.

[2] Anderson, J.H., R. Jain, and K. Jeffay: *Efficient Object Sharing in Quantum-based Real-Time Systems.* In *Proc. of the 19th IEEE Real-Time Systems Symposium*, pages 346–355. IEEE Computer Society, 1998.

[3] Anderson, J.H., R. Jain, and S. Ramamurthy: *Wait-Free Object-Sharing Schemes for Real-Time Uniprocessors and Multiprocessors.* In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 111–122. IEEE Computer Society, 1997.

[4] Anderson, J.H. and M. Moir: *Wait-Free Synchronization in Multiprogrammed Systems: Integrating Priority-based and Quantum-based Scheduling.* In *Proceedings of 18th ACM Symposium on Principles of Distributed Computing*, pages 123–132, Atlanta, Georgia, USA, 1999.

[5] Anderson, J.H. and S. Ramamurthy: *A Framework for Implementing Objects and Scheduling Tasks in Lock-Free Real-Time Systems.* In *Proc. of the 17th IEEE Real-Time Systems Symposium*, pages 92–105. IEEE Computer Society, 1996.

[6] Anderson, J.H., S. Ramamurthy, and R. Jain: *Implementing Wait-Free Objects on Priority-based Systems.* In *Proceedings of 16th ACM Symposium on Principles of Distributed Computing*, pages 229–238, Santa Barbara, California, USA, 1997.

[7] Anderson, J.H., S. Ramamurthy, and K. Jeffay: *Real-Time Computing with Lock-Free Shared Objects.* ACM Transactions on Computer Systems, 15(2):134–165, 1997.

[8] Anderson, J.H., S. Ramamurthy, M. Moir, and K. Jeffay: *Lock-Free Transactions for Real-Time Systems.* In Bestavros, A., S.H. Son, and K.J. Lin (editors): *Real-time database systems: Issues and applications*, pages 215–234. Kluwer Academic Publishers Norwell, MA, USA, 1997.

[9] Audsley, N., A. Burns, M. Richardson, K. Tindell, and A.J. Wellings: *Applying new scheduling theory to static priority pre-emptive scheduling.* Software Engineering Journal, 8(5):284–292, Sep 1993.

[10] Bacha, Andrew, Cheryl Bauman, Ruel Faruque, Michael Fleming, Chris Terwelp, Charles Reinholtz, Dennis Hong, Al Wicks, Thomas

Alberi, David Anderson, Stephen Cacciola, Patrick Currier, Aaron Dalton, Jesse Farmer, Jesse Hurdus, Shawn Kimmel, Peter King, Andrew Taylor, David Van Covern, and Mike Webster: *Odin: Team VictorTango's entry in the DARPA Urban Challenge.* Journal of Field Robotics - Special Issue on the 2007 DARPA Urban Challenge, 25(8):467–492, June 2008.

[11] Bardins, Stanislavs, Tony Poitschke, and Stefan Kohlbecher: *Gaze-based Interaction in Various Environments.* In *Proceeding of the 1st ACM workshop on Vision networks for behavior analysis*, pages 47–54. ACM, 2008.

[12] Beazley, David M.: *SWIG: An Easy to Use Tool For Integrating Scripting Languages with C and C++.* In *Fourth Annual USENIX Tcl/Tk Workshop*, Monterey, CA, 1996.

[13] Bestavros, A., S.H. Son, and K.J. Lin: *Real-time database systems: Issues and applications.* Kluwer Academic Publishers Norwell, MA, USA, 1997.

[14] Bianchi, E. and L. Dozio: *Some Experiences in fast hard realtime control in user space with RTAI-LXRT.* In *2nd Realtime Linux Workshop*, Orlando, 2000.

[15] Bianchi, E., L. Dozio, G.L. Ghiringhelli, and P. Mantegazza: *Complex Control Systems, Applications of DIAPM–RTAI at DIAPM.* In *1st Realtime Linux Workshop*, Vienna, 1999.

[16] Bibliographisches Institut & F. A. Brockhaus AG: *Brockhaus-Enzyklopädie online.* Brockhaus, Leipzig, Mannheim, 2008, besucht am 18.8.2008.

[17] Blum, Stefan: *OSCAR - Eine Systemarchitektur für den autonomen, mobilen Roboter MARVIN.* In: *Autonome Mobile Systeme*, Informatik aktuell, Seiten 218–230. Springer-Verlag, November 2000.

[18] Bonwick, Jeff: *The Slab Allocator: An Object-Caching Kernel Memory Allocator.* In: *Proceedings of the USENIX Summer 1994 Technical Conference.* USENIX Association, 1994.

[19] Bovet, Daniel P. and Marco Cesati: *Understanding the Linux Kernel.* O'Reilly, Sebastopol, CA, 3nd edition, 2006.

[20] Brandenburg, B. and J. Anderson: *Reader-Writer Synchronization for Shared-Memory Multiprocessor Real-Time Systems.* In *Proc. of the 21st Euromicro Conference on Real-Time Systems*, pages 2–9, Dublin, Ireland, 2009.

[21] Bruyninckx, H.: *Open Robot Control Software: The OROCOS Project.* In *IEEE Intl. Conf. on Robotics and Automation*, volume 3, pages 2523–2528, 2001.

[22] Bruyninckx, H., P. Soetens, and B. Koninckx: *The Real-Time Motion Control Core of the Orocos Project.* In *IEEE Intl. Conf. on Robotics and Automation*, volume 2, pages 2766–2771, 2003.

[23] Buss, Martin, Michael Beetz, and Uwe Haass: *CoTeSys - Cognition for Technical Systems - Informationen zum Exzellenzcluster.* Technical report,

Lehrstuhl für Steuerungs- und Regelungstechnik, Technische Universität München, 2008.

[24] Bülow, Alexander von: *Optimale Cache-Nutzung für Realzeitsoftware auf Multiprozessorsystemen.* Doktorarbeit, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2005.

[25] Chen, J.: *A Loop-Free Asynchronous Data Sharing Mechanism in Multiprocessor Real-Time Systems based on Timing Properties.* In: *Proc. of the 23rd Intl. Conf. on Distributed Computing Systems Workshops*, Seiten 184–190, Providence, Rhode Island, USA, 2003.

[26] Chen, J. und A. Burns: *Asynchronous Data Sharing in Multiprocessor Real-Time Systems using Process Consensus.* In: *Proc. of the 10th Euromicro Workshop on Real-Time Systems*, Seiten 2–9, Berlin, Germany, 1998.

[27] Chen, J. und A. Burns: *Loop-Free Asynchronous Data Sharing in Multiprocessor Real-Time Systems based on Timing Properties.* In: *6th Intl. Conf. on Real-Time Computing Systems and Applications*, Seiten 236–246, Hong Kong, China, 1999.

[28] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, SGI et al.: *Itanium C++ ABI (Revision: 1.86).* http://www.codesourcery.com/public/cxx-abi/abi.html, 2001, besucht am 19.8.2008.

[29] Corbet, Jonathan, Alessandro Rubini, and Greg Kroah-Hartman: *Linux Device Drivers.* O'Reilly, Sebastopol, CA, 3nd edition, 2005.

[30] Crespo, A., I. Ripoll, and M. Masmano: *Dynamic Memory Management for Embedded Real-Time Systems.* In *IFIP Conf. on Distributed and Parallel Embedded Systems*, Braga, Portugal, 2006.

[31] Dang, Thao, C. Hoffmann, and C. Stiller: *Self-calibration for Active Automotive Stereo Vision.* In *Proc. IEEE Intelligent Vehicles Symposium*, pages 364–369, 2006.

[32] DARPA: *Press Release: $2 Million Cash Prize Awarded to Stanfords Stanley as Five Autonomous Ground Vehicles Complete DARPA Grand Challenge Course (October 9, 2005).* http://www.darpa.mil/grandchallenge05/GC05winnerFINALwTerraM.pdf, 2005.

[33] DARPA: *Press Release: DARPA Announces Third Grand Challenge: Urban Challenge Moves to the City (5/1/06).* http://www.darpa.mil/grandchallenge/docs/PR_UC_Announce_Update_12_06.pdf, 2006.

[34] DARPA: *Press Release: Finalists announced (11/1/07).* http://www.darpa.mil/grandchallenge/docs/FinalistsUpdated.pdf, 2007.

[35] DARPA: *Press Release: Tartan Racing wins $2 million prize for DARPA Urban Challenge (11/4/07).* http://www.darpa.mil/grandchallenge/ucwinnertt.pdf, 2007.

[36] Date, C. J. and Hugh Darwen: *A guide to the SQL standard.* Addison-Wesley, Reading, Mass., 4. edition, 1997.

[37] Dickmanns, Dirk: *Rahmensystem für visuelle Wahrnehmung veränderlicher Szenen durch Computer.* Doktorarbeit, Universität der Bundeswehr München, 1998.

[38] Dickmanns, E. D.: *The 4D-Approach to Dynamic Machine Vision.* In: *Proc. IEEE Decision and Control*, Band 4, Seiten 3770–3775, 1994.

[39] Dickmanns, E. D.: *The development of machine vision for road vehicles in the last decade.* In: *Proc. IEEE Intelligent Vehicles Symposium*, Band 1, Seiten 268–281 vol.1, 2002.

[40] Dickmanns, E. D.: *Dynamic Vision-Based Intelligence.* AI Magazine, 25(2):10–30, 2004.

[41] Dickmanns, E. D.: *Dynamic Vision for Perception and Control of Motion.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[42] Dickmanns, E. D., R. Behringer, D. Dickmanns, T. Hildebrandt, M. Maurer, F. Thomanek und J. Schiehlen: *The seeing passenger car 'VaMoRs'.* In: *Proc. IEEE Intelligent Vehicles Symposium*, Seiten 68–73, 1994.

[43] Dickmanns, E. D. und V. Graefe: *Applications of dynamic monocular machine vision.* Machine Vision and Applications, 1(4):241–261, 1988.

[44] Dickmanns, E. D. und V. Graefe: *Dynamic Monocular Machine Vision.* Machine Vision and Applications, 1(4):223–240, 1988.

[45] Dijkstra, E.W.: *Cooperating sequential processes.* In: Genuys, F. (Herausgeber): *Programming languages*, Seiten 43–112. Academic Press, New York, 1968.

[46] DIN Deutsches Institut für Normung e.V. (Herausgeber): *Formelzeichen für die Elektrotechnik - Teil 2: Telekommunikation und Elektronik (IEC 60027-2:2005); Deutsche Fassung DIN EN 60027-2:2007.* Beuth Verlag, Berlin, Wien, Zürich, 2007.

[47] DIN Deutsches Institut für Normung e.V. (Herausgeber): *Leittechnik - Prozeßautomatisierung - Automatisierung mit Prozeßrechensystemen, Begriffe; DIN V 19233:1998-07.* Beuth Verlag, Berlin, Wien, Zürich, 2008.

[48] Dozio, L. und P. Mantegazza: *Linux Real Time Application Interface (RTAI) in low cost high performance motion control.* In: *Motion Control 2003.* National Italian Association for Automation, 2003.

[49] Duchow, Christian und Björn Körtner: *Aggregating lane markings into lanes for intersection assistance.* In: *Proc. IEEE Intelligent Vehicles Symposium*, Seiten 722–727, 2007.

[50] FlexRay Consortium: *The communication system for advanced automotive control applications.* `http://www.flexray.com`.

[51] FOG, AGNER: *Calling conventions for different C++ compilers and operating systems.* Technischer Bericht, Copenhagen University College of Engineering, 2008, Version vom 2008-06-29.

[52] FRANKE, HUBERTUS, RUSTY RUSSELL und MATTHEW KIRKWOOD: *Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux.* In: *Ottawa Linux Symposium*, Ottawa, USA, 2002.

[53] FRANKE, U., D. GAVRILA, S. GÖRZIG, F. LINDNER, F. PAETZOLD und C. WÖHLER: *Bildverstehen im innerstädtischen Verkehr.* In: *Autonome Mobile Systeme 1998*, Seiten 37–44. Springer-Verlag, 1998.

[54] FRANKE, U., D. GAVRILA, S. GÖRZIG, F. LINDNER, F. PUETZOLD und C. WÖHLER: *Autonomous Driving Goes Downtown.* IEEE Intelligent Systems and their Applications, 13(6):40–48, 1998.

[55] FRANKE, U., S. GÖRZIG, F. LINDNER, D. MEHREN und F. PAETZOLD: *Steps towards an Intelligent Vision System for Driver Assistance in Urban Traffic.* In: *IEEE Conference on Intelligent Transportation Systems 1997*, Seiten 601–606, 1997.

[56] FREE SOFTWARE FOUNDATION: *GNU General Public License Version 2.0.* Boston, MA, 1991.

[57] FRESE, C., T. BATZ, and J. BEYERER: *Kooperative Verhaltensentscheidung für Gruppen kognitiver Automobile auf Grundlage des gemeinsamen Lagebilds.* at-Automatisierungstechnik, 56(12):644–652, 2008.

[58] FÄRBER, GEORG: *Prozessrechentechnik.* Springer, Berlin, 3. Auflage, 1994.

[59] FÄRBER, GEORG und OTTHEIN HERZOG: *ASKOF – Architektur und Schnittstellen für kognitive Funktionen in Fahrzeugen.* Arbeits- und Ergebnisbericht 2003-2004 an die Deutsche Forschungsgemeinschaft, Dezember 2004.

[60] GALLMEISTER, BILL: *POSIX.4 - Programming for the Real World.* O'Reilly, Sebastopol, CA, 1995.

[61] GAVRILA, D., U. FRANKE, C. WÖHLER und S. GÖRZIG: *Real-Time Vision for Intelligent Vehicles.* IEEE Instrumentation & Measurement Magazine, 4(2):22–27, 2001.

[62] GCC TEAM: *GCC 3.4 Release Series - Changes, New Features, and Fixes*, zuletzt besucht am 19.8.2008.

[63] GCC TEAM: *Using the GNU Compiler Collection - Binary Compatibility*, zuletzt besucht am 19.8.2008.

[64] GERKEY, B., R.T. VAUGHAN und A. HOWARD: *The player/stage project: Tools for multi-robot and distributed sensor systems.* In: *Proceedings of the 11th International Conference on Advanced Robotics*, Seiten 317–323, 2003.

[65] GERUM, PHILIPPE: *Xenomai - Implementing a RTOS emulation framework on GNU/Linux*. Technischer Bericht, http://www.xenomai.org, April 2004.

[66] GINDELE, TOBIAS, DANIEL JAGSZENT, BENJAMIN PITZER und RÜDIGER DILLMANN: *Design of the planner of Team AnnieWAYs autonomous vehicle used in the DARPA Urban Challenge 2007*. In: *Proc. IEEE Intelligent Vehicles Symposium*, Seiten 1131–1136. IEEE Press, Juni 2008.

[67] GOEBL, MATTHIAS: *Homepage der Realzeitdatenbasis für kognitive Automobile (KogMo-RTDB)*. `http://www.kogmo-rtdb.de`.

[68] GOEBL, MATTHIAS: *Einflüsse der Bussysteme moderner PCs auf das Laufzeitverhalten von Realzeitsoftware*. Diplomarbeit am Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2003.

[69] GOEBL, MATTHIAS: *Vorschlag zur Spezifikation des Fahrzeugrechners für das kognitive Automobil (C3)*. Technischer Bericht, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2006.

[70] GOEBL, MATTHIAS: *KogMo-RTDB: Einführung in die Realzeitdatenbasis für Kognitive Automobile (C3)*. Technischer Bericht, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2007.

[71] GOEBL, MATTHIAS, MATTHIAS ALTHOFF, MARTIN BUSS, GEORG FÄRBER, FALK HECKER, BERND HEISSING, SVEN KRAUS, ROBERT NAGEL, FERNANDO PUENTE LEÓN, FLORIAN RATTEI, MARTIN RUSS, MICHAEL SCHWEITZER, MICHAEL THUY, CHENG WANG und HANS JOACHIM WUENSCHE: *Design and Capabilities of the Munich Cognitive Automobile*. In: *Proc. IEEE Intelligent Vehicles Symposium*, Seiten 1101–1107. IEEE Press, Juni 2008.

[72] GOEBL, MATTHIAS, SEBASTIAN DRÖSSLER und GEORG FÄRBER: *Systemplattform für videobasierte Fahrerassistenzsysteme*. In: LEVI, P., M. SCHANZ, R. LAFRENZ und V. AVRUTIN (Herausgeber): *Autonome Mobile Systeme 2005*, Informatik Aktuell, Seiten 187–193. Springer-Verlag, Dezember 2005.

[73] GOEBL, MATTHIAS und GEORG FÄRBER: *A Real-Time-capable Hard- and Software Architecture for Joint Image and Knowledge Processing in Cognitive Automobiles*. In: *Proc. IEEE Intelligent Vehicles Symposium*, Seiten 734–740. IEEE Press, Juni 2007.

[74] GOEBL, MATTHIAS und GEORG FÄRBER: *Eine realzeitfähige Softwarearchitektur für kognitive Automobile*. In: BERNS, K. und T. LUKSCH (Herausgeber): *Autonome Mobile Systeme 2007*, Informatik Aktuell, Seiten 198–204. Springer-Verlag, Oktober 2007.

[75] GOEBL, MATTHIAS und GEORG FÄRBER: *Interfaces for Integrating Cognitive Functions into Intelligent Vehicles*. In: *Proc. IEEE Intelligent Vehicles Symposium*, Seiten 1093–1100. IEEE Press, Juni 2008.

[76] GOEBL, MATTHIAS und FLORIAN RATTEI: *KogMo-RTDB + MATLAB/Simulink: Schnittstellen der Realzeitdatenbasis für Kognitive Automobile zu MATLAB/Simulink (C3).* Technischer Bericht, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2006.

[77] GOEBL, MATTHIAS, FLORIAN RATTEI, STEPHAN NEUMAIER und GEORG FÄRBER: *Eine leistungsfähige Hard- und Softwarearchitektur für kognitive Funktionen in Fahrzeugen.* In: *5. Workshop Fahrerassistenzsysteme FAS2008*, Seiten 30–37, Walting im Altmühltal, Germany, April 2008.

[78] GOEBL, MATTHIAS und JOACHIM SCHRÖDER: *Das KogniMobil-Referenzsystem: HOWTO zur Benutzung (C3/QAG4).* Technischer Bericht, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2007.

[79] GRAEFE, VOLKER: *The BVV-Family of Robot Vision Systems.* In: *IEEE International Workshop on Intelligent Motion Control*, Seiten 55–65, Istanbul, Turkey, 1990.

[80] GREGOR, R., M. LÜTZELER, M. PELLKOFER, K.-H. SIEDERSBERGER und E. D. DICKMANNS: *A vision system for autonomous ground vehicles with a wide range of maneuverng capabilities.* In: *Proceedings of the Second International Workshop on Computer Vision Systems*, Seiten 1 – 20. Springer-Verlag, 2001.

[81] GRESSER, KLAUS: *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme.* Nummer 268 in *Fortschrittsberichte VDI, Reihe 10*. VDI–Verlag, Düsseldorf, 1993.

[82] GUMPP, THOMAS, THOMAS SCHAMM, STEPHAN BERGMANN, J. MARIUS ZÖLLNER und RÜDIGER DILLMANN: *PMD basierte Fahrspurerkennung und -verfolgung für Fahrerassistenzsysteme.* In: BERNS, K. und T. LUKSCH (Herausgeber): *Autonome Mobile Systeme 2007*, Informatik Aktuell, Seiten 226–232. Springer-Verlag, Oktober 2007.

[83] GÖRZIG, STEFFEN: *CPPvm - Parallel Programming in C++.* In: *IEEE International Conference on Cluster Computing*, Seiten 141–144, 2001.

[84] GÖRZIG, STEFFEN: *Eine generische Software-Architektur für Multi-Agentensysteme und ihr Einsatz am Beispiel von Fahrerassistenzsystemen.* Shaker Verlag, Aachen, 2003.

[85] GÖRZIG, STEFFEN und UWE FRANKE: *ANTS - Intelligent Vision in Urban Traffic.* In: *IEEE Conf. Intelligent Transportation Systems*, Seiten 545–549, 1998.

[86] GÖRZIG, STEFFEN, AXEL GERN und PAUL LEVI: *Realzeitfähige Multiagentenarchitektur für autonome Fahrzeuge.* In: *Autonome Mobile Systeme 1999*, Seiten 44–55. Springer-Verlag, 2000.

[87] HARMS, PHILIPP und STEPHAN NEUMAIER: *Bioanaloge Ansätze zur Steigerung der Robustheit maschinellen Sehens bei Fahrzeugen.* In: *3. Workshop Fahrerassistenzsysteme FAS2005*, Walting im Altmühltal, April 2005.

[88] Heinecke, H., K.P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.L. Maté, K. Nishikawa und T. Scharnhorst: *AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures.* In: *Convergence Conference 2004*, Detroit, USA, 2004.

[89] Heinze, F., L. Schafers, C. Scheidler und W. Obeloer: *Trapper: Eliminating Performance Bottlenecks in a Parallel Embedded Application.* IEEE Concurrency, 5(3):28–37, 1997.

[90] Herlihy, Maurice: *Wait-free synchronization.* ACM Transactions on Programming Languages and Systems, 13(1):124–149, 1991.

[91] Hoare, C.A.R.: *Monitors: an operating system structuring concept.* Communications of the ACM, 17(10):549–557, October 1974.

[92] Hohmuth, M. and H. Härtig: *Pragmatic Nonblocking Synchronization for Real-Time Systems.* In *Proc. of the 2001 USENIX Annual Technical Conference*, pages 217–230. USENIX Association, 2001.

[93] Hopfner, Thomas, Jürgen Stohr, Wolfram Faul und Georg Färber: *RTCPU – Realzeitanwendungen auf Dual-Prozessor PC Architekturen.* it+ti — Informationstechnik und Technische Informatik, 43(6):291, Dezember 2001.

[94] Härtig, H., M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter: *The performance of μ-kernel-based systems.* In *Proceedings of the 16th ACM Symposium on Operating System Principles*, page 6677, Saint-Malo, France, 1997.

[95] Härtig, Hermann, Michael Hohmuth, and Jean Wolter: *Taming linux.* In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems*, Adelaide, Australia, 1998.

[96] Härtig, Hermann and Michael Roitzsch: *Ten years of research on l4-based real-time.* In *Proceedings of the Eighth Real-Time Linux Workshop*, Lanzhou, China, 2006.

[97] IEEE Computer Society Portable Application Standards Committee (editor): *IEEE Std 1003.1d-1999: Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) - Amendment d: Additional Realtime Extensions [C Language].* Institute of Electrical and Electronics Engineers, Inc., New York, USA, 1999.

[98] Intel Corporation, Mt. Prospect, IL, USA: *IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, 2002.

[99] Jaus Working Group: *The Joint Architecture for Unmanned Systems - Reference Architecture Specification - Version 3.3.* `http://www.jauswg.org/baseline/refarch.html`, 2007, zuletzt besucht am 11.5.2008.

[100] Kammel, Sören and Benjamin Pitzer: *Lidar-based lane marker detection and mapping.* In *Proc. IEEE Intelligent Vehicles Symposium*, pages 1137–1142. IEEE Press, June 2008.

[101] Kammel, Sören, Benjamin Pitzer, Stefan Vacek, Joachim Schroeder, Christian Frese, Moritz Werling, and Matthias Goebl: *Team AnnieWAY Technical System Description.* Technical report, DARPA Urban Challenge Finalist Technical Paper, 2007.

[102] Kammel, Sören, Julius Ziegler, Benjamin Pitzer, Moritz Werling, Tobias Gindele, Daniel Jagzent, Joachim Schröder, Michael Thuy, Matthias Goebl, Felix von Hundelshausen, Oliver Pink, Christian Frese, and Christoph Stiller: *Team AnnieWAY's autonomous system for the DARPA Urban Challenge 2007.* In *International Journal of Field Robotics Research.* John Wiley & Sons, Inc., 2008.

[103] Kao, B. and H. Garcia-Molina: *An Overview of Real-Time Database Systems*, pages 463–486. Prentice-Hall, 1995.

[104] Keltcher, C. N., K. J. McGrath, A. Ahmed, and P. Conway: *The Opteron processor for multiprocessor servers.* In *IEEE Micro*, volume 23, pages 66–76, 2003.

[105] Kerscher, Thilo, Arne Rönnau, Marco. Ziegenmeyer, Bernd Gassmann, J. Marius Zöllner, and Rüdiger Dillmann: *Behaviour-based control of a six-legged walking machine LAURON IVc.* In *11th Intl. Conf. on Climbing and Walking Robots*, Coimbra, Portugal, 2008.

[106] Kiszka, J.: *The Real-Time Driver Model and First Applications.* In *Seventh Real-Time Linux Workshop*, Lille, November 2005.

[107] Kiszka, J., N. Hagge, P. Hohmann, and B. Wagner: *RTnet - Eine Open-Source-Lösung zur Echtzeitkommunikation über Ethernet.* In *Telematik 2003, VDI-Berichte 1785*, pages 55–64, Siegen, Germany, 2003.

[108] Kiszka, J. and B. Wagner: *Securing Software-Based Hard Real-Time Ethernet.* In *2nd IEEE International Conference on Industrial Informatics*, Berlin, Germany, 2004.

[109] Kiszka, J., B. Wagner, Y. Zhang, and J. Broenink: *RTnet - A Flexible Hard Real-Time Networking Framework.* In *10th IEEE International Conference on Emerging Technologies and Factory Automation*, Catania, Italy, 2005.

[110] Knoop, Steffen, Stefan Vacek, Raoul Zöllner, Christian Au, and Rüdiger Dillmann: *A CORBA-based distributed software architecture for control of service robots.* In *Int. Conf. Intelligent Robots and Systems*, volume 4, pages 3656–3661, Sendai, Japan, 2004.

[111] Knuth, Donald Ervin: *The art of computer programming: 1 Fundamental algorithms.* Addison-Wesley, 1972.

[112] Knuth, Donald Ervin: *The art of computer programming: 2 Sorting and Searching.* Addison-Wesley, 2. edition, 1975.

[113] Kopetz, H. and J. Reisinger: *The non-blocking write protocol: solution to a real-time synchronization problem.* In *Proc. IEEE Real-Time Systems Symposium*, pages 131–137, 1993.

[114] Kota, Rajesh and R. Oehler: *Horus: large-scale symmetric multiprocessing for opteron systems.* In *Micro, IEEE*, volume 25, pages 30–40, 2005.

[115] Kroah-Hartman, Greg: *Linux Kernel Development.* In *Proceedings of the Linux Symposium 2007*, Ottawa, Ontario, Canada, 2007.

[116] Lamport, Leslie: *Concurrent Reading and Writing.* Communications of the ACM, 20(11):806–811, 1977.

[117] Lampson, Butler W. and David D. Redell: *Experience with processes and monitors in Mesa.* Commun. ACM, 23(2):105–117, 1980.

[118] Lattner, A. D., J. D. Gehrke, I. J. Timm, and O. Herzog: *A Knowledge-based Approach to Behavior Decision in Intelligent Vehicles.* In *Proc. IEEE Intelligent Vehicles Symposium*, pages 466–471, 2005.

[119] Lauer, Michael: *Building Embedded Linux Distributions with BitBake and OpenEmbedded.* In *In Proceedings of the Free and Open Source Software Developers European Meeting (FOSDEM)*, Brussels, Belgium, 2005.

[120] Lea, Doug: *A Memory Allocator.* Technical report, Computer Science Department, State University of New York at Oswego, 1996.

[121] Lehoczky, J., L. Sha, and Y. Ding: *The rate monotonic scheduling algorithm: exact characterization and average case behavior.* Proc. of the Real Time Systems Symposium, pages 166–171, Dec 1989.

[122] Lenz, C., S. Nair, M. Rickert, A. Knoll, W. Rosel, J. Gast, A. Bannat, and F. Wallhoff: *Joint-Action for Humans and Industrial Robots for Assembly Tasks.* In *IEEE Intl. Symposium on Robot and Human Interactive Communication*, pages 130–135, 2008.

[123] Levine, D.L., S. Flores-Gaitan, and D.C. Schmidt: *An empirical evaluation of OS support for real-time CORBA object request brokers.* In *Proceedings of the Multimedia Computing and Networking 2000 conference, SPIE*, 2000.

[124] Levine, John R.: *Linkers and loaders.* Morgan Kaufmann, San Francisco, CA, 2005.

[125] Liedtke, Jochen: *On μ–kernel construction.* In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, Colorado, December 1995.

[126] *Linux Kernel - Quellcode des Betriebssystems.* `http://www.kernel.org`.

[127] LIU, C. L. and JAMES W. LAYLAND: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* J. ACM, 20(1):46–61, 1973.

[128] LIU, JANE W. S.: *Real–Time Systems.* Prentice Hall, New Jersey, 2000.

[129] LOESER, JORK and HERMANN HÄRTIG: *Real Time on Ethernet using off-the-shelf Hardware.* In *Proceedings of the 1st Intl Workshop on Real-Time LANs in the Internet Age*, Vienna, Austria, 2002.

[130] LOESER, JORK and HERMANN HÄRTIG: *Low-latency Hard Real-Time Communication over Switched Ethernet.* In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, Catania, Italy, 2004.

[131] LOESER, JORK and HERMANN HÄRTIG: *Using Switched Ethernet for Hard Real-Time Communication.* In *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, Dresden, Germany, 2004.

[132] LOVE, ROBERT: *Linux-Kernel-Handbuch - Leitfaden zu Design und Implementierung von Kernel 2.6.* Addison-Wesley, München, Boston, 2006.

[133] MASMANO, M., I. RIPOLL, P. BALBASTRE und A. CRESPO: *A constant-time dynamic storage allocator for real-time systems.* Real-Time Systems, 2008.

[134] MASMANO, M., I. RIPOLL, A. CRESPO und J. REAL: *TLSF: A New Dynamic Memory Allocator for Real-Time Systems.* In: *16th Euromicro Conference on Real-Time Systems*, Seiten 79–88, 2004.

[135] MASMANO, M., I. RIPOLL, J. REAL, A. CRESPO und A. J. WELLINGS: *Implementation of a constant-time dynamic storage allocator.* Software: Practice and Experience, 38(10):995–1026, 2008.

[136] MATHIASON, G., S.F. ANDLER und S.H. SON: *Virtual Full Replication by Adaptive Segmentation.* In: *13th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications*, Seiten 327–336, Daegu, Korea, 2007.

[137] MAURER, M. und E. D. DICKMANNS: *System architecture for autonomous visual road vehicle guidance.* In: *IEEE Intelligent Transportation Systems*, Seiten 578–583, 1997.

[138] MAURER, MARKUS: *Flexible Automatisierung von Straßenfahrzeugen mit Rechnersehen.* Doktorarbeit, Universität der Bundeswehr München, 2000.

[139] MCKENNEY, P.E. und J.D. SLINGWINE: *Read-copy update: using execution history to solve concurrency problems.* In: *Proc. of the 11th Intl. Conf. on Parallel and Distributed Computing and Systems*, Seiten 509–518, 1998.

[140] MCNAUGHTON, MATTHEW, CHRISTOPHER BAKER, TUGRUL GALATALI, BRYAN SALESKY, CHRISTOPHER URMSON und JASON ZIGLAR: *Software Infrastructure for an Autonomous Ground Vehicle.* Journal of Aerospace Computing, Information, and Communication, 5(12):491–505, 2008.

[141] Mehnert, F., M. Hohmuth, and H. Härtig: *Cost and benefit of separate address spaces in real-time operating systems.* In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 124–133, Austin, Texas, USA, 2002.

[142] MISRA Consortium (The Motor Industry Software Reliability Association): *Guidelines for the Use of the C Language in Vehicle Based Software.* The Motor Industry Research Association, Warwickshire, UK, fourth impression 2002 edition, 1998.

[143] Molnar, Ingo: *Linux RT-Preemption Patch.* `http://www.kernel.org/pub/linux/kernel/projects/rt/`, 2008, besucht am 29.8.2008.

[144] Montemerlo, Michael, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pflueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun: *Junior: The Stanford entry in the Urban Challenge.* Journal of Field Robotics - Special Issue on the 2007 DARPA Urban Challenge, 25(9):569 – 597, June 2008.

[145] Montemerlo, Michael, Nicholas Roy, and Sebastian Thrun: *Perspectives on Standardization in Mobile Robot Programming: The Carnegie Mellon Navigation (CARMEN) Toolkit.* In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, volume 3, pages 2436–2441, Las Vegas, NV, 2003.

[146] Münnich, Alexander, Marcel Birkhold, Georg Färber, and Peter Woitschach: *Towards an Architecture for Reactive Systems Using an Active Real-Time Database and Standardized Components.* In *Proc. of the IEEE Intl. Symposium on Database Engineering and Applications*, Montreal, Canada, 1999.

[147] Nagel, Robert, Stephan Eichler, and Jörg Eberspächer: *Intelligent Wireless Communication for Future Autonomous and Cognitive Automobiles.* In *Proc. IEEE Intelligent Vehicles Symposium*, pages 716 – 721, 2007.

[148] Nesnas, I.A., , R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I.H. Shu, et al.: *CLARAty: Challenges and steps toward reusable robotic software.* International Journal of Advanced Robotic Systems, 3(1):023–030, 2006.

[149] Neumaier, Stephan and Georg Färber: *Videobasierte 4D-Umfelderfassung für erweiterte Assistenzfunktionen.* it Information Technology, 49(1):33–39, 2007.

[150] Neumaier, Stephan, Philipp Harms, and Georg Färber: *Videobasierte Umfelderfassung zur Fahrerassistenz.* In *4. Workshop Fahrerassistenzsysteme FAS2006*, Löwenstein, October 2006.

[151] *The NML Configuration Files.* Manufacturing Engineering Laboratory, National Institute of Standards and Technology, `http://www.isd.mel.nist.gov/projects/rcslib/NMLcfg.html`, 2006, besucht am 10.5.2008.

[152] NSLU2-Linux Entwickler- und Benutzergruppe: *SlugOS-Firmware.* `http://www.nslu2-linux.org/wiki/SlugOS/`, zuletzt besucht am 8.5.2008.

[153] Nyström, D., A. Tešanović, M. Nolin, C. Norström, and J. Hansson: *COMET: A Component-Based Real-Time Database for Automotive Systems.* In *Proc. of the Workshop on Software Engineering for Automotive Systems at 26th Intl. Conf. on Software engineering*, 2004.

[154] Object Management Group: *Real-time CORBA Specification - Version 1.2.* `http://www.omg.org/docs/formal/05-01-04.pdf`, 2005.

[155] Object Management Group: *Common Object Request Broker Architecture (CORBA/IIOP) - Specifications, Version 3.1.* `http://www.omg.org/spec/CORBA/3.1/`, 2008.

[156] Olson, M.A., K. Bostic, and M. Seltzer: *Berkeley DB.* In *USENIX Annual Technical Conference: Proceedings of the FREENIX Track*, pages 183–192, 1999.

[157] P. Puschner, A.V. Schedl: *Computing Maximum Task Execution Times – A Graph-Based Approach.* Real Time Systems, 13(1):67–91, 1997.

[158] Pellkofer, M. and E.D. Dickmanns: *Behavior decision in autonomous vehicles.* In *Proc. IEEE Intelligent Vehicles Symposium*, pages 495–500, 2002.

[159] Petters, Stefan M.: *Worst Case Execution Time Estimation for Advanced Processor Architectures.* PhD thesis, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, September 2002.

[160] Pilato, C. Michael, Ben Collins-Sussman, and Brian W. Fitzpatrick: *Version Control with Subversion.* O'Reilly, Sebastopol, CA, 2004.

[161] Puaut, Isabelle: *Real-Time Performance of Dynamic Memory Allocation Algorithms.* In *Proc. of the 14th Euromicro Conference on Real-Time Systems*, pages 41–49, Vienna, Austria, 2002.

[162] Ramamurthy, Srikanth, Mark Moir, and James H. Anderson: *Real-Time Object Sharing with Minimal System Support.* In *Proceedings of 15th ACM Symposium on Principles of Distributed Computing*, pages 233–242. ACM, 1996.

[163] Rattei, Florian, Matthias Goebl und Georg Färber: *Beitrag zur Robustheitssteigerung videobasierter Fahrerassistenzsysteme durch frühe Rückkopplungen zur Sensorebene.* In: *Bildverarbeitung in der Mess- und Automatisierungstechnik*, Seiten 225–236. VDI-Berichte, VDI Verlag, Düsseldorf, 2007.

[164] Reichardt, Dirk: *Kontinuierliche Verhaltenssteuerung eines autonomen Fahrzeugs in dynamischer Umgebung.* Doktorarbeit, Universität Kaiserslautern, Forschung F1M/IA Daimler Benz, 1996.

[165] Rieder, André: *Multisensorielle Fahrzeugerkennung in einem verteilten Rechnersystem für autonome Fahrzeuge.* Doktorarbeit, Universität der Bundeswehr München, 2000.

[166] Rodríguez, Laura Vázquez, Martin Felder und Alois Knoll: *A cognitive architecture framework for CoTeSys.* In: *Proc. of the 1st Int. Workshop on Cognition for Technical Systems*, Munich, Germany, 2008.

[167] Rosenblum, Mark: *Team Urbanator Technical Description.* Technical report, DARPA Urban Challenge Semifinalist Technical Paper, 2007.

[168] Schamm, Thomas, J. Marius Zöllner, Stefan Vacek, Joachim Schröder, and Rüdiger Dillmann: *Obstacle detection with a Photonic Mixing Device-camera in autonomous vehicles.* Int. J. Intelligent Systems Technologies and Applications, 5(3/4):315–324, 2008.

[169] Schiehlen, J. and E. D. Dickmanns: *A Camera Platform for Intelligent Vehicles.* In *Proc. IEEE Intelligent Vehicles Symposium*, pages 393–398, 1994.

[170] Schmidt, André, Steffen Görzig und Paul Levi: *ANTSRT - Eine Software-Architektur für Fahrerassistenzsysteme.* In: *Autonome Mobile Systeme 2003*, Seiten 44–55. Springer-Verlag, 2003.

[171] Schmidt, D.C., M. Deshpande und C. ORyan: *Operating system performance in support of real-time middleware.* In: *Proc. of the 7th Workshop on Object-oriented Real-time Dependable Systems, IEEE*, San Diego, CA, USA, 2002.

[172] Schmidt, Douglas C. und Fred Kuhns: *An Overview of the Real-Time CORBA Specification.* IEEE Computer, 33(6):56–63, 2000.

[173] Schmidt-Rohr, Sven R., Martin Loesch, Zhixing Xue und Rüdiger Dillmann: *Hardware and Software Architecture for Robust Autonomous Behavior of a Domestic Robot Assistant.* In: *Proc. of the Intl. Conf. on Intelligent Autonomous Systems 10*, Baden Baden, 2008.

[174] Scholl, Kay-Ulrich: *MCA2 Programmer's Guide.* `http://www.mca2.org/generated/base/index.html`, zuletzt besucht am 12.5.2008.

[175] Scholl, Kay-Ulrich, Jan Albiez und Bernd Gassmann: *MCA - An Expandable Modular Controller Architecture.* In: *3rd Real-Time Linux Workshop*, Milano, Italy, 2001.

[176] Scholl, Kay-Ulrich Scholl, Volker Kepplin, Jan Albiez und Rüdiger Dillmann: *Developing Robot Prototypes with an Expandable Modular Controller Architecture.* In: *Proc. of the Intl. Conf. on Intelligent Autonomous Systems 6*, Seiten 67–74, Venice, Italy, 2000.

[177] Schröder, Joachim, Tobias Gindele, Daniel Jagszent und Rüdiger Dillmann: *Path Planning for Cognitive Vehicles using Risk Maps.* In: *Proc. IEEE Intelligent Vehicles Symposium*, Seiten 1119–1124. IEEE Press, Juni 2008.

[178] Schröder, Joachim, Tilo Gockel und Rüdiger Dillmann: *Embedded Linux: Das Praxisbuch.* Springer-Verlag, Berlin, 2009.

[179] Schröder, Joachim, Markus Hoffmann und Rüdiger Dillmann: *Behavior Decision and Path Planning for Cognitive Vehicles using Behavior Networks.* In: *Proc. IEEE Intelligent Vehicles Symposium*, 2007.

[180] Schröder, Joachim, Udo Müller und Rüdiger Dillmann: *Smart Roadster Project: Setting up Drive-by-Wire or How to Remote-Control your Car.* In: *Intelligent Autonomous Vehicles 2006*, Tokyo, Japan, 2006.

[181] Schröder, Joachim, Peter Steinhaus, Tilo Gockel und Rüdiger Dillmann: *Design of a Holonomous Platform for a Humanoid Robot using MCA2.* In: *Proc. of the Intl. Conf. on Intelligent Autonomous Systems 8*, Seiten 836–843, Amsterdam, Niederlande, 2004.

[182] Sha, L., R. Rajkumar und J.P. Lehoczky: *Priority inheritance protocols: an approach to real-time synchronization.* IEEE Transactions on Computers, 39(9):1175–1185, Sep 1990.

[183] Siedersberger, Karl-Heinz: *Komponenten zur automatischen Fahrzeugführung in sehenden (semi-)autonomen Fahrzeugen.* Doktorarbeit, Universität der Bundeswehr München, 2003.

[184] Simmons, Reid und David Apfelbaum: *A task description language for robot control.* In: *Proc. of the Conf. on Intelligent Robotics and Systems (IROS)*, Victoria, B.C., Canada, 1998.

[185] Simmons, Reid und Dale James: *IPC - Inter-Process Communication - A Reference Manual - for IPC Version 3.6.* Technischer Bericht, Carnegie Mellon University, School of Computer Science/Robotics Institute, 2001.

[186] Simpson, H.R.: *Four-slot fully asynchronous communication mechanism.* IEE Proceedings - Computers and Digital Techniques, 137(1):17–30, 1990.

[187] Siro, Arther, Carsten Emde und Nicholas McGuire: *Assessment of the Realtime Preemption Patches (RT-Preempt) and their impact on the general purpose performance of the system.* In: *Proceedings of the Real-Time Linux Workshop 2007*, Linz, Austria, 2007.

[188] Smith, Reid G.: *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver.* IEEE Transactions on Computers, C-29(12):1104–1113, 1980.

[189] Soetens, Peter: *A Software Framework for Real-Time and Distributed Robot and Machine Control.* Doktorarbeit, Katholieke Universiteit Leuven, Belgien, 2006.

[190] Sonderforschungsbereich Transregio 28: *Kognitive Automobile.* `http://www.kognimobil.org`.

[191] Speed, Derek S.: *The Moblin.org Open Source Project.* Technischer Bericht, Open Source Technology Center, Intel Corporation, 2008.

[192] Stanford Racing Team: *Stanford's Robotic Vehicle J̈unior:̈ Interim Report.* Technical report, DARPA Urban Challenge Finalist Technical Paper, 2007.

[193] Stevens, R.W. and S.A. Rago: *Advanced Programming in the UNIX Environment.* Addison-Wesley Professional, 2. edition, 2005.

[194] Stiller, Christoph, Georg Färber, and Sören Kammel: *Cooperative Cognitive Automobiles.* In *Proc. IEEE Intelligent Vehicles Symposium*, pages 215–220, 2007.

[195] Stiller, Christoph und Georg Färber: *Kognitive Automobile – Sonderforschungsbereich/Transregio 28.* Finanzierungsantrag 2006-1009 bei der Deutschen Forschungsgemeinschaft, 2005.

[196] Stohr, Jürgen: *Auswirkungen der Peripherieanbindung auf das Realzeitverhalten PC-basierter Multiprozessorsysteme.* Doktorarbeit, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, März 2006.

[197] Stohr, Jürgen, Alexander von Bülow und Georg Färber: *Controlling the Influence of PCI DMA Transfers on Worst Case Execution Times of Real–Time Software.* In: *Proc. 4th International Workshop on Worst Case Execution Time Analysis in conjunction with the 16th Euromicro Conference on Real–Time Systems*, 2004.

[198] Stohr, Jürgen, Alexander von Bülow und Georg Färber: *Bounding Worst-Case Access Times in Modern Multiprocessor Systems.* In: *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, 2005.

[199] Stohr, Jürgen, Alexander von Bülow, and Georg Färber: *Using State of the Art Multiprocessor Systems as Real-Time Systems – The RECOMS Software Architecture.* In *Proceedings of the 16th Euromicro Conference on Real-Time Systems – Work in Progress Session*, Catania, Italy, June 2004.

[200] Stohr, Jürgen, Alexander von Bülow und Matthias Goebl: *Einflüsse des PCI–Busses auf das Laufzeitverhalten von Realzeitsoftware.* Technischer Bericht, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2003.

[201] Sundell, H.: *Applications of Non-Blocking Data Structures to Real-Time Systems.* PhD thesis, Chalmers University of Technology and Göteborg University, 2002.

[202] Sundell, H. and P. Tsigas: *Space Efficient Wait-Free Buffer Sharing in Multiprocessor Real-Time Systems based on Timing Information.* In *Proc. of the 7th Intl. Conf. on Real-Time Computing Systems and Applications*, pages 433–440, Cheju Island, South Korea, 2000.

[203] Tanenbaum, Andrew S.: *Moderne Betriebssysteme.* Prentice Hall/Pearson Studium, 2002.

[204] Tanenbaum, Andrew S. and James Goodman: *Computerarchitektur*. Prentice Hall, München [i.e.]; London, Mexiko, New York, 1999.

[205] Thrun, Sebastian, Michael Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney: *Stanley: The Robot that Won the DARPA Grand Challenge*. Journal of Field Robotics, 23(9):661–692, June 2006.

[206] Thuy, Michael, Matthias Althoff, Martin Buss, Klaus Diepold, Jörg Eberspächer, Georg Färber, Matthias Goebl, Bernd Heissing, Sven Kraus, Robert Nagel, Youssef Naous, Florian Obermeier, Fernando Puente León, Florian Rattei, Cheng Wang, Michael Schweitzer, and Hans-Joachim Wünsche: *Kognitive Automobile - Neue Konzepte und Ideen des Sonderforschungsbereiches/TR-28*. In *3. Tagung Aktive Sicherheit durch Fahrerassistenz*, Garching bei München, Germany, April 2008.

[207] Thuy, Michael, A. Saber Tehrani, and Fernando Puente León: *Bayessche fusion von stereobildfolgen und lidardaten*. In Puente, Fernando and Michael Heizmann (editors): *Bildverarbeitung in der Mess- und Automatisierungstechnik*, volume 1981 of *VDI-Berichte*, pages 67–78. VDI Verlag, 2007.

[208] Traut, Manuel: *Real-time CORBA performance on Linux-RT PREEMPT*. In *11th Real-Time Linux Workshop*, Linz, Österreich, 2007.

[209] Trodden, J. and D. Anderson: *HyperTransport System Architecture*. Addison–Wesley Publishing Company, Boston, 2003.

[210] Ulmer, Berthold: *VITA II - Active Collision Avoidance in Real Traffic*. In *Proceedings of the IEEE Intelligent Vehicles*, pages 1–6, Paris, France, 1994.

[211] Urmson, C. et al.: *Tartan Racing: A Multi-Modal Approach to the DARPA Urban Challenge*. Technical report, DARPA Urban Challenge Finalist Technical Paper, 2007.

[212] Urmson, Christopher, Joshua Anhalt, Hong Bae, J. Andrew (Drew) Bagnell, Christopher Baker, Robert E. Bittner, Thomas Brown, M. N. Clark, Michael Darms, Daniel Demitrish, John Dolan, David Duggins, David Ferguson, Tugrul Galatali, Christopher M. Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas Howard, Sascha Kolski, Maxim Likhachev, Bakhtiar Litkouhi, Alonzo Kelly, Matthew McNaughton, Nick Miller, Jim Nickolaou, Kevin Peterson, Brian Pilnick, Ragunathan Rajkumar, Paul Rybski, Varsha Sadekar, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod M. Snider,

JOSHUA C. STRUBLE, ANTHONY (TONY) STENTZ, MICHAEL TAYLOR, WILLIAM RED L. WHITTAKER, ZIV WOLKOWICKI, WENDE ZHANG, and JASON ZIGLAR: *Autonomous driving in urban environments: Boss and the Urban Challenge.* Journal of Field Robotics - Special Issue on the 2007 DARPA Urban Challenge, Part I, 25(9):425–466, June 2008.

[213] UTZ, H., S. SABLATNOG, S. ENDERLE, and G. KRAETZSCHMAR: *Miro - Middleware for Mobile Robot Applications.* IEEE Transactions on Robotics and Automation, 18(4):493–497, 2002.

[214] VACEK, STEFAN, TOBIAS GINDELE, and RÜDIGER DILLMANN: *Situation Classification for Cognitive Automobiles using Case-Based Reasoning.* In *Proc. IEEE Intelligent Vehicles Symposium*, pages 704 – 709, 2007.

[215] VACEK, STEFAN, ROBERT NAGEL, THOMAS BATZ, FRANK MOOSMANN, and RÜDIGER DILLMANN: *An Integrated Simulation Framework for Cognitive Automobiles.* In *Proc. IEEE Intelligent Vehicles Symposium*, pages 221 – 226, 2007.

[216] VACEK, STEFAN, THOMAS SCHAMM, JOACHIM SCHRÖDER, J. MARIUS ZÖLLNER, and RÜDIGER DILLMANN: *Collision avoidance for cognitive automobiles using a 3D PMD camera.* In *Intelligent Autonomous Vehicles 2007*, Toulouse, France, 2007.

[217] WALLHOFF, FRANK, JÜRGEN GAST, ALEXANDER BANNAT, STEFAN SCHWÄRZLER, GERHARD RIGOLL, CORNELIA WENDT, SABRINA SCHMIDT, MICHAEL POPP, and BERTHOLD FÄRBER: *Real-time Framework for On- and Off-line Multimodal Human-Human and Human-Robot Interaction.* In *Proc. of the 1st Int. Workshop on Cognition for Technical Systems*, Munich, Germany, 2008.

[218] WANG, C., S. KRAUS, F. KOHLHUBER, and B. HEISSING: *Safety concept of the cognitive vehicle control.* In *FISITA World Automotive Congress 2008*, München, 2008.

[219] WERLING, MORITZ, TOBIAS GINDELE, DANIEL JAGSZENT, and LUTZ GRÖLL: *A Robust Algorithm for Handling Moving Traffic in Urban Scenarios.* In *Proc. IEEE Intelligent Vehicles Symposium*, pages 1108–1112. IEEE Press, June 2008.

[220] WERLING, MORITZ, MATTHIAS GOEBL, OLIVER PINK, and CHRISTOPH STILLER: *A Hardware and Software Framework for Cognitive Automobiles.* In *Proc. IEEE Intelligent Vehicles Symposium*, pages 1080–1085. IEEE Press, June 2008.

[221] WERLING, MORITZ and LUTZ GRÖLL: *Low-level Controllers Realizing High-level Decisions in an Autonomous Vehicle.* In *Proc. IEEE Intelligent Vehicles Symposium*, pages 1113–1118. IEEE Press, June 2008.

[222] WILHELM, R., J. ENGBLOM, A. ERMEDAHL, N. HOLSTI, S. THESING, D. WHALLEY, G. BERNAT, C. FERDINAND, R. HECKMANN, T. MITRA, et al.: *The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools.* ACM Transactions on Programming Languages and Systems, 7(3):1–53, 2008.

[223] Wilson, Paul R.: *Uniprocessor Garbage Collection Techniques.* Lecture Notes In Computer Science, Vol. 637: Proceedings of the International Workshop on Memory Management, 1992.

[224] Wilson, Paul R., Mark S. Johnstone, Michael Neely, and David Boles: *Dynamic storage allocation: A survey and critical review.* Lecture Notes In Computer Science, Vol. 986: Proceedings of the International Workshop on Memory Management, 1995.

[225] Wilson, Paul R. and Sheetal V. Kakkad V.: *Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware.* In *Intl. Workshop on Object Orientation and Operating Systems*, pages 364–377. IEEE Comp. Society Press, 1992.

[226] Wulf, Oliver, Jan Kiszka, and Bernardo Wagner: *A Compact Software Framework for Distributed Real-Time Computing.* In *5th Real-Time Linux Workshop*, Valencia, Spain, 2003.

[227] Yaghmour, Karim: *A Practical Approach to Linux Clusters on SMP Hardware using Adeos.* Technical report, Opersys Inc., Quebec, Canada, 2001.

[228] Yaghmour, Karim: *Adaptive Domain Environment for Operating Systems.* Technical report, Opersys Inc., Quebec, Canada, 2001.

[229] Yaghmour, Karim: *Building a RTOS over Adeos.* Technical report, Opersys Inc., Quebec, Canada, 2001.

[230] Yaghmour, Karim: *Building embedded Linux systems.* O'Reilly, Sebastopol, CA, 2003.

[231] Yodaiken, Victor J.: *The RTLinux Manifesto.* In *Proceedings of The 5th Linux Expo*, Raleigh North Carolina, March 1999.

[232] Yodaiken, Victor J.: *US5995745: Adding real-time support to general purpose operating systems.* United States Patent, Socorro, NM, 1999.

[233] Zhang, Yuchen, Bojan Orlic, Peter Visser, and Jan Broenink: *Hard Real-Time Networking on Firewire.* In *7th Real-Time Linux Workshop*, Lille, France, 2005.

[234] Ziegler, Julius, Moritz Werling, and Joachim Schröder: *Navigating car-like vehicles in unstructured environment.* In *Proc. IEEE Intelligent Vehicles Symposium*, pages 787–791. IEEE Press, June 2008.