Echtzeit-Scheduling und zeitliche Konsistenz bei mehrstufiger, verteilter Datenverarbeitung

Diplomarbeit

Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Institut für Verteilte Systeme

Frederic Rossau

Juli 2004

Kurzfassung

Mobile Roboter nehmen ihre Umwelt über Sensoren wahr. Dazu müssen die rohen Daten der Sensoren in der Regel verarbeitet werden, meist in mehreren Stufen, um relevante Informationen zu extrahieren. Da die Roboter in Echtzeit agieren, benötigen sie die verarbeiteten Sensordaten innerhalb vorgegebener Zeitschranken. Die Sensordaten und die daraus extrahierten Informationen können, bedingt durch die begrenzten Reichweiten und Blickwinkel der Sensoren und anderen technisch Einschränkungen, ungenau und unvollständig sein. Durch die Fusion der Sensordaten von mehren Robotersystemen können die Umgebungsinformationen vervollständigt und ergänzt werden.

Die folgende Arbeit beschäftigt sich mit der Suche nach Lösungen, mit denen es Möglich ist, für die Sensordatenverarbeitung Ende-zu-Ende Zeitgarantien einzuhalten. Diese Garantien betreffen zum einen die Rechtzeitigkeit der Verarbeitung unter Verwendung eines Task-Pair Schedulers. Dabei bedarf die Mehrstufigkeit der Verarbeitung besondere Beachtung. Zum anderen ist es die Gewährleistung der zeitlichen Kohärenz der verteilt gewonnenen Sensordaten, die besonders wichtig ist, wenn die Einsatzumgebung der mobilen Roboter und deren Sensoren eine dynamische Umwelt ist. Zu diesem Zweck wurde eine globale Zeitgesteuerte Lösung der mehrstufigen, verteilten Datenverarbeitung entwickelt und umgesetzt, die die Sensordatenverarbeitung in zeitlich koordinierte Phasen aufteilt. Die Koordinierung kann dabei auch über die Grenzen des lokalen Systems auf Basis einer synchronisierten Uhr durchgeführt werden. Die Umsetzung der entwickelten Konzepte wurde in Form eines Frameworks realisiert, was zur Unterstützung für die Umsetzung von Applikationen der mehrstufigen, verteilten Datenverarbeitung dienen kann.

I. Inhaltsverzeichnis

I. Inhaltsverzeichnis i

II. Abbildungsverzeichnis iii

III. Tabellenverzeichnis v

IV. Verzeichnis der Abkürzungen vi

1 Einleitung 1

1.1 Kontext 1

1.2 Motivation und Ziel 2

1.3 Aufgabenstellung 4

1.4 Ergebnisse 5

1.5 Gliederung 6

2 Grundlagen 7

2.1 TAFT-Scheduler 7

2.2 Gruppenkommunikationsprotokoll RGCP 10

2.3 Uhrensynchronisation im lokalen Funknetzwerk 12

2.4 RTLinux 13

2.5 Mehrstufige, verteilte Sensorfusion von Laserscandaten 14

3 Verwandte Arbeiten 18

3.1 Ereignisgesteuerte und Zeitgesteuerte Systeme 18

3.2 Kalman-Filtering und Multi Object Tracking 21

4 Rechtzeitigkeit der Datenverarbeitung 24

4.1 Rechtzeitigkeit und Zeitgarantien für die Bearbeitungsphasen 24

4.2 Erweiterung des Task-Konzeptes 34

5 Zeitliche Kohärenz 38

5.1 Global gültige Zeitstempel der Sensordaten 38

5.2 Globale Zeitgesteuerte Koordinierung der Verarbeitungsphasen 41

5.3 Zeitliche Kohärenz mittels Kalman-Filtering Verfahren 48

5.3.1 Positionsverfolgung der Elemente mittels Kalmanfilter 50

5.3.2 Positionsverfolgung mehrerer Elemente 54

6 Implementierung 58
6.1 Klassen der mehrstufigen, verteilten Datenverarbeitung 58
6.1.1 Umsetzung der mehrstufigen Datenverarbeitung 58
6.1.2 Zeitstempel und globale Zeitgesteuerte Koordinierung 61
6.1.3 Kommunikation und Fusion 63
6.1.4 Implementierungsbeispiel: Schritt für Schritt 65
6.2 Umsetzung des Beispiels mit Hilfe der FFM-Klassen 70
7 Ergebnisse und Messungen 76
8 Zusammenfassung und Ausblick 88
9 Literaturverzeichnis 90

II. Abbildungsverzeichnis

Abbildung 1: Task-Pair Scheduling 9
Abbildung 2: Aufbau eines RGCP Netzwerkes 11
Abbildung 3: Ablauf des RGCP Protokolls [29] 11
Abbildung 4: Protokoll der WLAN Uhrensynchronisation [8] 13
Abbildung 5: RT-Linux Kernel und RT-Module 14
Abbildung 6: Mehrstufige Verarbeitung von Laserscanndaten [15] 15
Abbildung 7: Filter und Fusionsstufen [15] 16
Abbildung 8: Kalman-Filtering Verfahren [20] 22
Abbildung 9: Multiple Hypothesis Tracking Algorithmus [22] 23
Abbildung 10: Mehrstufige und Verteilte Sensordatenverarbeitung 24
Abbildung 11: Perioden- und Phasenfolgen 26
Abbildung 12: Variable Ausführungszeiten des Konturfilters [18] 27
Abbildung 13: Mögliches Auftreten von Exceptions zu unterschiedlichen Zeiten 30
Abbildung 14: Reihenfolgeabhängige Phasenfolge 33
Abbildung 15: Periodenaufteilung durch mehrere TAFT-Threads 34
Abbildung 16: Parallele Verarbeitung innerhalb derselben Phase 35
Abbildung 17: Periodenaufteilung durch mehrere TAFT-Threads (erweitert) 37
Abbildung 18: Verwendung der globalen Uhr 39
Abbildung 19: Antwortzeit eines Laserscanners 40
Abbildung 20: Beispiel: Probleme bei der Fusion zeitlich inkohärenter Daten 42
Abbildung 21: Wechsel von zeitlichen Kohärenzen und Inkohärenzen 43
Abbildung 22: Globale Reihenfolgeabhängigkeit der Fusion und Kommunikation 43
Abbildung 23: Koordinierung der Periodenstartzeiten 45
Abbildung 24: Ermitteln des Korrekturwertes 47
Abbildung 25: Fusion von Sensordaten sortiert nach den Zeitstempeln 49
Abbildung 26: Beispiel Ballposition: Herausfiltern von Positionsfehlern 50
Abbildung 27: Positionsvorhersage 52
Abbildung 28: NN Positionszuordnung 56
Abbildung 29: Elementfusion 57
Abbildung 30: FFM-Klassen 58
Abbildung 31: Funktion des FFMOrganizer Objektes 61

Abbildung 32: Relevanz der Datenobjekte im Verlauf der Berechnungen 62
Abbildung 33: Kommunikationsheader für die Sensordaten 65
Abbildung 34: Anwenderbeispiel 66
Abbildung 35: Klassen der Lasersensordatenverarbeitung 71
Abbildung 36: Kommunikation zwischen den Modulen 75
Abbildung 37: Beispiel: Setup eines Szenarios im Labor 78
Abbildung 38: Abstraktionsstufen der Sensordaten 79
Abbildung 39: Szenario: bewegter Laserscanner 80
Abbildung 40: Phasenaktivitäten der 1. Station 81
Abbildung 41: Phasenaktivitäten der 2. Station 82
Abbildung 42: Beispiel: Fusion inkohärenter und kohärenter Sensordaten 83
Abbildung 43: Zeitstempelunterschiede der Sensordaten 84
Abbildung 44: Szenario der 3. Messung 85
Abbildung 45: Messung der Positionsverfolgung 86

III. Tabellenverzeichnis

Tabelle 1: Vergleich Ereignisgesteuerte vs. Zeitgesteuerte Systeme 20
Tabelle 2: Verarbeitungsphasen bei unterschiedlichen Fusionsstufen 73
Tabelle 3: Modulabhängigkeiten 75
Tabelle 4: Systemressourcen für die Messungen 78
Tabelle 5: ECET,WCET und resultierende Deadlines 80

IV. Verzeichnis der Abkürzungen

AP	Access Point (des RGCP)
CL	Client (des RGCP)
CPU	Central Processing Unit
ECET	Expected Case Execution Time
EDF	Earliest Deadline First
EP	Exception Part
ET	Event-Triggered (Ereignisgesteuert)
FFM	Filter-Fusion-Modul
LRT	Latest Release Time
MHT	Multiple-Hypothesis-Tracking Algorithmus
MP	Main Part
NN	Nearest Neighbor
OD	Omission Degree
RGCP	Real-time Group Communication Protocol
RM	Rate Monotonic
TAFT	Time Aware Fault Tolerant
TP	Task Pair
TT	Time-Triggered (Zeitgesteuert)

1 Einleitung

1.1 Kontext

Das Forschungsgebiet der Mobile Robotik beschäftigt sich mit der Suche nach Möglichkeiten, mobile Robotersysteme zum Erfüllen bestimmter Aufgaben einzusetzen. Die Gründe dafür können unterschiedlich sein. Zum Beispiel, weil der Einsatz von Robotern sich als effizienter erweist, als der Einsatz von Menschen. Oder weil der Einsatz von menschlichen Arbeitskräften zu riskant oder aufwändig wäre. Als Beispiel seien hier die letzten Marsmissionen genannt, bei denen zum Teil erfolgreich mobile Roboter („SPIRIT“, „OPERTUNITY“) eingesetzt wurden, um die Marsoberfläche zu erkunden. Unbemannte Marsmissionen sind um einiges wirtschaftlicher und weniger gefährlich als Bemannte. Aber auch auf der Erde bieten sich viele Einsatzmöglichkeiten an. Die Arbeitsgruppe für Echtzeitsysteme und Kommunikation des Instituts für Verteilte Systeme an der Otto-von-Guericke-Universität Magdeburg beschäftigt sich unter anderem mit diesem Thema.

Mobile Roboter bewegen sich in einer realen Umwelt. Daher müssen sie auch in der Lage sein, mit der Selbigen zu interagieren. Das heißt, sie bewegen sich durch die Umwelt und führen Aktionen aus, die zur Erfüllung ihrer gestellten Aufgaben notwendig sind. Damit diese Interaktion möglich ist, ist es nötig, die umgebende Umwelt wahrzunehmen. Dieses ist aber in der Regel keine einmalige Aktion, da es sich meist bei der Umgebung, in der sich die Roboter befinden, um eine dynamische Umwelt handelt. Das heißt, es treten zeitlich bedingte Veränderungen auf. Daher erfolgt die Wahrnehmung der Umgebung in der Regel periodisch, um diese Dynamik aufzunehmen und damit das Weltbild des Roboters regelmäßig zu aktualisieren. Für die Wahrnehmung sind Sensoren verantwortlich, von denen es eine Menge unterschiedlicher Arten gibt. Beispielsweise Infrarot, Laserscanner, Kamera usw. Viele dieser Sensoren liefern aber eine große Datenmenge, was bedeutet, dass die Daten noch verarbeitet und abstrahiert werden müssen, bis verwertbare Informationen erhalten werden. Das kann auch in mehreren Schritten notwendig sein. Dieses wird bezeichnet als mehrstufige Sensordatenverarbeitung. Des Weiteren liefern Sensoren meist kein vollständiges Weltbild. Daher ist es sinnvoll, auch mehrere Roboter zu verwenden, die die ihnen gestellte Aufgabe im Team lösen. Ein Teil der Teamarbeit ist es dann auch, die eigenen Sensordaten im Team untereinander auszutauschen und zu fusionieren, um ein genaueres Weltbild zu erhalten. Dafür wiederum ist eine Kommunikationsart nötig, die sich mit der Anforderung der Mobilität verträgt. Die Drahtlose Kommunikation ist dabei eine Art, die diese Verträglichkeit liefert. Auch hier wurden in der Arbeitsgruppe Echtzeit und Kommunikation Lösungen entwickelt.

Mobile Roboter, die im Team arbeiten, die Umwelt über Sensoren wahrnehmen, und diese Daten im Team austauchen, müssen also folgende Aktionen durchführen: mehr-

stufige Datenverarbeitung, Kommunikation und Datenfusion. Diese Aktionen müssen unter Echtzeitbedingungen ausgeführt werden. Das heißt, es müssen dabei Zeitschranken eingehalten werden, um jederzeit ein möglichst korrektes und aktuelles Weltbild aufrecht zu erhalten, worauf die Roboter ihre Interaktion mit der Umwelt begründen, um ihre Aufgaben zu erfüllen oder das Überleben des eigenen Systems zu gewährleisten. In diesem Kontext der verteilten mehrstufigen Sensordatenverarbeitung unter Echtzeitbedingungen ist diese Arbeit anzusiedeln.

1.2 Motivation und Ziel

Die verteilte mehrstufige Sensorfusion lässt sich in zwei wesentliche Hauptaspekte unterteilen, die Fusion und die mehrstufige Verarbeitung der Sensordaten. Durch die mehrstufige Verarbeitung wird versucht, aus den rohen Sensordaten die relevanten Daten zu extrahieren oder neue Informationen abzuleiten. Durch die Fusion von Sensordaten, die von mehren Robotern gewonnen wurden, oder allgemeiner von mehreren Sensoren gewonnenen Sensorinformationen, kann eine genauere und vollständigere Wahrnehmung der Umwelt erfolgen. In der Arbeitsgruppe für Echtzeitsysteme und Kommunikation wurden bereits Arbeiten zur Sensordatenfusion gemacht, wie zum Beispiel die verteilte Fusion von Lasersensordaten [15] oder die Stereobildverarbeitung [13]. Für Beide wurden Algorithmen, mit Anytime Eigenschaften entwickelt, womit sie auf einem Task-Pair (TP) Scheduler wie dem TAFT Scheduler [1] anwendbar sind. Dieser erlaubt einen flexiblen Umgang mit stark variablen Laufzeiten der verwendeten Algorithmen.

Bei der Sensorfusion werden die Daten von mehreren Sensoren zusammengeführt. Dabei kann es sich um homogene Sensoren, das heißt, Sensoren der gleichen Art, oder um heterogene Sensoren, also Sensoren unterschiedlicher Art handeln. Das Ergebnis sind dann redundante Umgebungsdaten, was eine höhere Ausfalltoleranz bedeuten kann, oder sich ergänzende Umgebungsdaten mit entsprechend höherer Informationsmenge. Die mehreren verwendeten Sensoren, deren Daten fusionieren werden, können sich alle lokal auf dem gleichem System befinden, oder verteilt über mehrere Systeme. Beispielsweise, wenn die Sensorfusion auf mobilen Robotern, die im Team agieren, einsetzt werden soll. Im Falle der verteilten Sensoren muss beachtet werden, dass diese die Umgebung unabhängig von einander beobachten. Dabei kann es sich um eine dynamische Umwelt handeln, was bedeutet, dass zeitlich bedingte Veränderungen auftreten können durch zum Beispiel sich bewegende Objekte. Auf Grund der unabhängigen Aufnahme durch die verteilten Sensoren kann es dann vorkommen, dass zeitliche Diskrepanzen zwischen deren Daten auftreten können, da sie auf zu unterschiedlichen Zeitpunkten beobachteten Zuständen der Umwelt basieren. Sind diese zu groß als dass sie von den Fusionsalgorithmen toleriert werden könnten, werden eventuell zu ungenaue oder sogar falsche Informationen über den Zustand der Umwelt daraus abgeleitet.

Ein weiteres Problem der Fusion ist die Abhängigkeit von den Sensordaten der verschiedenen verteilten Sensoren und damit die Abhängigkeit von entfernten Systemen oder Robotern. Damit die Fusion auf einem System stattfinden kann müssen die Daten der verteilten Sensoren lokal verfügbar sein. Es muss also eine Kommunikation der Fusion voran gehen. Beide Aktionen erfolgen unter Echtzeitbedingungen. Dass heißt, die Fusion kann nicht beliebig lange verzögert werden, um auf die Daten der beteiligten Systeme zu warten. Sind diese aber nicht verfügbar, kann jedoch auch keine Fusion stattfinden.

Das Problem der mehrstufigen Verarbeitung der Sensordaten durch Filter, Fusionsstufen und andere, ist der Zeitverbrauch durch die einzelnen Verarbeitungsschritte. Die führt zu einer zeitlichen Verzögerung zwischen der Aufnahme der rohen Daten durch die Sensoren und der Verfügbarkeit der daraus extrahierten Umgebungsinformation für die davon abhängige Applikation. Dieses bedeutet, dass diese Informationen sich auf einen Zustand der Umwelt aus der Vergangenheit beziehen. Je älter sie sind, desto weniger wert sind sie. Wenn sie zu alt sind, können sie sogar wertlos für die Applikation sein, da sie auf bestimmte für sie relevante Gegebenheiten in der Umwelt eventuell nicht mehr rechtzeitig reagieren könnte, was für ein System, dass unter Echtzeitbedingungen läuft, fatal sein kann. Deshalb besteht der Wunsch nach Zeitgarantien, die Einhaltung der so genannten Deadlines. Das sind Zeitschranken die keinesfalls überschritten werden dürfen. Damit kann der Applikation im Fall der mehrstufigen Verarbeitung von Sensordaten ein festgelegtes Minimum an Aktualität der Informationen zugesichert werden.

Der klassische Ansatz, um solch eine Zeitgarantie festzusetzen ist, die maximalen Laufzeiten der Algorithmen, die Worst-Case-Execution-Times (WCET), zu bestimmen. Handelt es sich dabei aber um Algorithmen mit stark variablen Laufzeiten, die Beispielsweise aus Einflüssen einer hohen Dynamik der Umwelt resultieren können, kann dieses sehr schwierig, ja sogar unmöglich sein. Die resultierenden Zeitgarantien, würden dann weit jenseits von vernünftigen tolerierbaren Grenzen und den tatsächlichen durchschnittlichen Laufzeiten liegen. In diesem Fall währen die durchschnittlichen Laufzeiten, die so genannte Expected-Case-Execution-Times (ECET), interessanter. Mit deren Hilfe kann eine Zeitschranke abgeschätzt werden, die in den meisten Fällen nicht überschritten wird und die größenmäßig in einem tolerierbaren Bereich liegt. Jedoch, da die Datenverarbeitung unter Echtzeitauflagen läuft, darf die Notwendigkeit, garantierte Zeitschranken bereitzustellen, nicht ignoriert werden. Task-Pair Scheduling Verfahren [3] sind Verfahren, die mit solchen erwarteten Ausführungszeiten arbeiten.

Diese Probleme der verteilten Daten, der Zeitdiskrepanzen und der mehrstufigen Verarbeitung, unter Verwendung eines Task-Pair Scheduling Verfahrens (TAFT-Scheduler[1]), sind Gegenstand dieser Arbeit. Da der Einsatz der verteilten Sensorfusion im Rahmen eine Gruppe mobiler Systeme und damit unter Echtzeitanforderungen die Absicht ist, ist es Ziel, bei der Ausführung die Einhaltung von Ende-zu-Ende Zeitgarantien zu gewährleisten. Ende-zu-Ende bedeutet hierbei, dass die Einhaltung der Zeitgarantien aus Sicht der Applikation gegeben ist, was alle Aspekte der Sensorfusion

(lokale Verarbeitung und, Kommunikation und Fusion) sowie deren Zusammenspiel beinhaltet. Dabei sind zwei Arten von zeitlichen Garantien zu unterscheiden:

1. **Rechtzeitigkeit:** Es existiert eine bekannte obere Schranke für die Zeitdauer zwischen der Beobachtung der Umgebung durch die Sensoren und dem vorliegen der fusionierten, verarbeiteten Daten für die Applikation. Dabei soll auch garantiert werden, dass die grundlegenden Aktionen jedes Verarbeitungsschrittes durchgeführt werden können.

2. **Zeitliche Kohärenz:** Es werden nur solche Daten fusioniert, die Umgebungsdaten aus einem hinreichend kleinen Zeitfenster beschreiben, so dass nicht Informationen fusioniert werden, die auf Zuständen der Umwelt zu völlig unterschiedlichen Zeitpunkten basieren.

Im Rahmen dieser Arbeit werden Lösungen, die diese Ziele umsetzen, entwickelt und an einem praktischen Beispiel, der verteilten mehrstufigen Datenverarbeitung von Laserscannerdaten, erprobt.

1.3 Aufgabenstellung

Aus diesen im vorherigen Abschnitt gestellten Zielen, ergeben sich damit folgende Teilaufgaben, die zur Umsetzung dieser Ziele gelöst werden sollen:

1. ***Erstellen und realisieren eines Task-Konzeptes für die lokale Verarbeitung der Sensordaten, das die Reihenfolgeabhängigkeit der Tasks berücksichtigt.*** Damit soll das Ziel der Rechtzeitigkeit erreicht werden, und auch die Garantie soll gegeben werden können, dass innerhalb einer Datenverarbeitungsperiode alle notwendigen Schritte ausgeführt werden können und dieses auch in der geforderten Reihenfolge, da Verarbeitungsschritte von vorangegangenen Schritten über Ausgabe-Eingabe Beziehungen (Producer-Consumer) voneinander abhängig sein können.

2. ***Gewährleistung der zeitlichen Kohärenz der fusionierten Daten unter Verwendung einer globalen Zeit, ggf. unter zusätzlicher Verwendung von Verfahren zur zeitlichen Extrapolation der gewonnen Daten (z.B. Kalman-Filter).*** Es soll demgemäß möglich sein, den Sensordaten Zeitstempel zuzuordnen, die deren Erzeugungsdatum, also den Zeitpunkt, zu dem die Umwelt mittels der entsprechenden Sensoren beobachtet wurde, festhalten. Mit Hilfe dieses Zeitstempels soll es möglich sein, die Daten zeitlich zu ordnen und eine Aussage zu machen, über deren zeitliche Kohärenz, um die Fusion zeitlich inkohärenter Daten zu vermeiden. Ferner soll es ggf. möglich sein, zeitliche Kohärenz der Daten zu erzeugen, durch geeignete Extrapolationsverfahren. Ein solches Ver-

fahren, ist zum Beispiel das Kalman-Filtering Verfahren, was sich in vielen verschiedenen Anwendungsgebieten der Rechentechnik, Navigation, Raum- und Luftfahrt usw. bewährt hat.

3. ***Zeitlich koordinierte Durchführung der lokalen Berechnungen und der Kommunikation, so dass die Einhaltung der Rechtzeitigkeit Ende-zu-Ende gewährleistet ist.*** Durch die zeitliche Koordinierung bestimmter Aktionen kann ebenfalls zeitliche Kohärenz der Daten erreicht werden. Solche Aktionen währen zum Beispiel alle Aktivitäten, die mit der Beobachtung der Umwelt und Aufnahme der Daten durch die Sensoren zu tun haben. Zeitliche Koordinierung anderer Aktionen kann nötig sein, um Datenabhängigkeiten zwischen Verarbeitungsschritten zu behandeln, wobei die Aktionen dieser Schritte auf entfernte Systeme, wie mobile Roboter zum Beispiel, verteilt ablaufen können. Damit währe die Kommunikation eine Aktivität die eventuell koordiniert werden müsste.

1.4 Ergebnisse

Im Rahmen dieser Arbeit wurden Konzepte entwickelt, mit denen die geforderten zeitlichen Garantien der Rechtzeitigkeit und der zeitlichen Kohärenz der Sensordaten bei der verteilten Verarbeitung und Fusion gegeben werden können. Diese Konzepte wurden in Form von wieder verwendbaren Klassen umgesetzt, die ein Framework bilden, was als Grundlage zur Implementierung von Applikation zur mehrstufige, verteilten Sensordatenverarbeitung dienen soll. Als Grundlage für diese Konzepte dient der TAFT-Scheduler [1], womit die Ausführung der einzelnen Verarbeitungsstufen nicht auf die pessimistischen Laufzeitabschätzungen der maximalen Ausführungszeiten angewiesen ist, sondern auf eine realistischere Schätzung, bei der die zu erwartenden Laufzeiten verwendet werden. Diese Laufzeiten sind oft weit niedriger als die maximalen Ausführungszeiten. Die Folge ist eine bessere CPU-Auslastung und es kann damit auch ein höherer Datendurchsatz erzielt werden.

Die Rechtzeitigkeit der gesamten Datenverarbeitung wird dabei aber immer noch garantiert, durch eine Aufteilung der Verarbeitung, mittels eines geeigneten Task-Konzeptes, in Phasen mit festen Zeitfenstern, worin jeder Phase ein Minimum an Ausführungszeit zugesichert wird. Dieses System bildet außerdem eine globale Zeitgesteuerte Lösung. Global heißt hier, dass die Aktivitäten der Teilenehmer koordiniert werden können, indem die Phasen auf einer globale Uhr basierend aktiviert werden können. Damit werden zwei Möglichkeiten, zum Erzielen zeitlicher Kohärenz, der Verarbeitungsapplikation bereitgestellt. Erstens kann die globale Uhr zur Erzeugung von globalgültigen Zeitstempeln für die Daten herangezogen werden, die dann als Unterstützung dienen, für geeignete Verfahren zum Auflösen zeitlicher Diskrepanzen der Daten. Zweitens können mit global koordinierten Aktivitäten (Bsp. Wahrnehmung, Kommunikation) die Fusion zeitlich kohärenter Daten erzwungen werden.

Um diese Konzepte zu erproben, wurde ein konkretes Anwendungsbeispiel mit Hilfe der hier implementierten Klassen umgesetzt. Als Anwendungsbeispiel diente die verteilte Fusion von Lasersensordaten [15]. Sie wurde erneut Umgesetzt, wobei die Berechnungsalgorithmen größtenteils unverändert übernommen wurden. Um aber beide Möglichkeiten, mit dem Ziel der zeitlichen Kohärenz, testen zu können, wurde die Verarbeitung der Sensordaten um ein Verfahren erweitert, welches das Kalman-Filtering Verfahren [19] benutzt, um zeitlicher Diskrepanzen der Daten aufzulösen, und ist damit auf die globale Koordinierung der Aktivitäten nicht angewiesen. Diese Neuumsetzung der verteilten Fusion von Lasersensordaten wurde anschließend auf die Einhaltung der geforderten Garantien hin untersucht.

1.5 Gliederung

Im folgenden Kapitel wird auf die Grundlagen eingegangen, die für die Entwicklung der Konzepte und der Implementierung der Lösungen von Bedeutung sind. Anschließend werden im dritten Kapitel verwandte Arbeiten zum Thema Ereignisgesteuerte Systeme und das Kalman-Filtering Verfahren vorgestellt. Die in dieser Arbeit entwickelten Lösungen zur Rechtzeitigkeit der Datenverarbeitung und der zeitlichen Kohärenz der Sensordaten werden in Kapitel 4 und 5 erklärt und in Kapitel 6 wird auf die Implementierung dieser Lösungen und des Anwendungsbeispiels eingegangen. Die Ergebnisse der Implementierung und die Resultate der durchgeführten Messungen werden in Kapitel 7 vorgestellt. Abschließend folgt eine Zusammenfassung dieser Arbeit in Kapitel 8.

2 Grundlagen

In diesem Kapitel soll kurz auf die Grundlagen eingegangen werden, die für die Arbeit wichtig sind. Im ersten Abschnitt werden der TAFT-Scheduler [1] und das Task-Pair-Scheduling [3] erläutert. Im folgenden Abschnitt wird auf, das in dieser Arbeit verwendete Kommunikationsprotokoll, RGCP [4][5] eingegangen, gefolgt von der Uhrensynchronisation im lokalen Funknetzwerk [7][8] im dritten Abschnitt. Im vierten Abschnitt wird das Betriebssystem RTLinux [26] vorgestellt, was als Laufzeitumgebung für die Implementierung der hier entwickelten Lösungen diente. Im letzten Abschnitt wird dann ein Anwendungsbeispiel vorgestellt, die mehrstufige, verteilte Sensorfusion von Laserscandaten [15], was zum Zweck der Überprüfung der entwickelten Lösungen im Rahmen dieser Arbeit erneut umgesetzt wurde.

2.1 TAFT-Scheduler

In vielen Systemen laufen mehrere Prozesse parallel ab. Dabei konkurrieren sie eventuell um vorhandene Systemressourcen, die sie zur Erfüllung ihrer Aufgaben benötigen. Auf Grund dieser Konkurrenz existiert in der Regel eine zentrale Instanz, die für die Zuteilung dieser Ressourcen verantwortlich ist, das Betriebssystem. Eine wichtige Ressource ist die CPU-Zeit. Meist existieren mehr Prozesse auf dem System als CPUs. Daher müssen sich mehrere Prozesse die CPU-Zeiten teilen. Der Scheduler ist dabei der Teil des Betriebsystems, der für das Management dieser Ressource verantwortlich ist. Es existieren unterschiedliche Strategien, wie die Zuteilung erfolgt, je nach art des Betriebsystems. Für Echtzeit-Betriebsysteme werden Strategien verwendet, die es erlauben, für die Prozesse zeitliche Garantien festzulegen, so dass sie ihre Aufgaben innerhalb gesetzter Zeitschranken erfüllen können. Solche Echtzeit-Scheduler sind zum Beispiel der RM-Scheduler (Rate Monotonic) und der EDF-Scheduler (Earliest Deadline First). Sie erstellen Ausführungspläne für eine Menge von periodischen Tasks auf der Basis von Zeitparametern, wie der Periode, Deadline und den maximale Ausführungszeit WCET (Worst Case Excecution Time). Die WCET ist dabei die Zeit, die jedem Prozess pro Periode zugesichert wird und die er aber auch nie überschreiten darf. Der Scheduler ist dafür verantwortlich, dass die Prozesse stets ihre Deadlines einhalten, vorausgesetzt, ihm wurden korrekte Zeitparameter übergeben. Nicht immer existiert ein möglicher Ausführungsplan bzw. der Scheduler findet ihn nicht. Zu diesem Zweck gibt es zu den Echtzeit-Schedulern in der Regel einen Feasibility Test, der eine Menge von Tasks mit ihren Zeitparametern auf die Existenz eines Ausführungsplans untersucht. Erzielt dieser Test ein positives Ergebnis, dann findet der entsprechende Scheduler garantiert einen Ausführungsplan vorausgesetzt, die WCET Zeiten der Tasks wurden korrekt ermittelt.

Die Ermittlung von WCETs kann aber unter Umständen problematisch sein. Die Algorithmen der Prozesse können maximale Ausführungszeiten besitzen, die weit über den durchschnittlichen Zeiten liegen können, oder es ist gar nicht erst möglich, sie zu bestimmen, womit diese Scheduler keinen akzeptablen Ausführungsplan finden können. Extrem große WCET Zeiten führen zu einer schlechten CPU Auslastung, wenn die durchschnittlichen Zeiten die WCETs in den meisten Fällen weit unterschreiten. Eine bessere Auslastung könnte dadurch erreicht werden, wenn nicht die WCETs, sonder die durchschnittlichen Ausführungszeiten oder die erwarteten Ausführungszeiten ECETs (Expected Case Execution Time) verwendet werden. Damit auch weiterhin die Echtzeitauflagen erfüllt werden können, muss auch eine andere Scheduling-Strategie verwendet werden. Der TAFT-Scheduler [1] ist ein Scheduler, der auf den ECETs basierend arbeitet.

Der TAFT-Scheduler (Time Aware Fault Tolerant) ist ein echtzeitfähiger Scheduler mit dem Ziel einer verbesserten CPU-Auslastung. Er gehört zu der Klasse der TP-Scheduler (Task-Pair) [3]. Die Ausführung der periodischen Tasks ist dabei in zwei Teile gegliedert, der Mainpart MP und der Exceptionpart EP.

Mainpart MP: Im MP eines Threads ist die Hauptfunktionalität implementiert. Die vollständige Ausführung dieser Funktion wird nicht garantiert. Der Scheduler kann die Ausführung unterbrechen, falls sonst die Deadline nicht eingehalten werden kann. Auf diese Weise wird die Einhaltung der Deadline garantiert. Für die Algorithmen bedeutet das, dass sie eine bestimmte Anforderung erfüllen müssen, und zwar die der Anytime-Fähigkeit [28]. Anytime Algorithmen erzeugen von Anfang an ein Berechnungsergebnis, was sie im Laufe der weiteren Berechnung stetig verbessern. Somit steht ab dem Beginn der Ausführung ein gültiges Ergebnis bereit, womit ein Abbruch der Berechnung jederzeit toleriert werden kann. Das wiederum ist es, was sie für Echtzeitanwendungen und speziell für die Anwendung im TAFT-Scheduler interessant macht. Die MPs der Threads werden vom Scheduler auf der Basis der ECETs verarbeitet. Die ECET wird dabei als die Mindestausführungszeit garantiert, um für jeden Thread einen Mindestfortschritt der Berechnung gewährleisten zu können. Die Idee dahinter ist, dass eine höhere CPU Auslastung erreicht wird, auch wenn die MPs stark variable Ausführzeiten besitzen, da die ECETs in diesem Fall merklich kleiner sind als die maximal möglichen Ausführungszeiten der MPs. Wird die ECET im Laufe der Berechnung überschritten, so besteht die Möglichkeit eines Abbruchs, womit der Exceptionpart ausgelöst wird.

Exceptionpart EP: Der EP wird nur dann aktiviert, wenn der MP vom Scheduler unterbrochen werden musste, um die Verletzung der Deadline zu vermeiden. Er hat die Funktion, essentielle Aktionen auszuführen, die der MP nicht mehr, auf Grund des Abbruchs, durchführen konnte. Solche Aktionen sind Beispielsweise die Ausgabe der Ergebnisdaten, Freigabe von belegten Ressourcen oder andere Aufräumarbeiten, um die Berechnung der nächsten Periode vorzubereiten. Da diese Aktionen wichtig sind, verarbeitet der Scheduler sie basierend auf den maximalen Ausführungszeiten WCET der EPs. Daher sollten die EPs möglichst klein gehalten werden, so dass die WCET des

EPs sehr viel kleiner ist als die ECET des MPs. Die vollständige und rechzeitige Ausführung der EPs vor der Deadline kann garantiert werden, solange eine korrekte WCET angegeben wurde. Der Aufbau einer Thread-Funktion ist hier grob angedeutet.

```
void * thread_function(void *arg) {
      pthread_make_periodic_tp (pthread_self(), release);

      while (1) {
            pthread_wait_np_ex();

            BEGIN_MAIN_PART // Beginn des MP
            ...
            END_MAIN_PART // Ende des MP

            BEGIN_EXCEPTION_PART // Beginn des EP
            ...
            END_EXCEPTION_PART // Ende des EP
      }
      return 0;
}
```

Der TAFT-Scheduler verwendet eine mehrstufige Scheduling-Strategie, um die MPs und EPs zu verarbeiten. Da den MPs möglichst viel CPU-Zeit zur Verfügung gestellt werden soll, muss die Ausführung der entsprechenden EPs so weit wie möglich hinausgezögert werden. Jedoch müssen sie unbedingt vor der Deadline ausgeführt werden. Daher werden sie mittels LRT-Strategie (Latest Release Time) oder auch Reverse-EDF behandelt. Das heißt, es wird alles so spät wie möglich ausgeführt. Außerdem werden die EPs mit höherer Priorität behandelt, als die MPs. Die MPs dagegen werden nach EDF-Strategie (Earliest Deadline First) behandelt. Dass heißt, es wird alles so früh wie möglich erledigt. Dabei werden die MPs selbst in zwei Prioritätsklassen unterteilt. Die höchste Priorität unter den MPs besitzen alle, die in ihrer aktuellen Ausführung ihre ECET noch nicht überschritten haben. Sobald sie jedoch ihre ECET überschreiten, verlieren sie ihre Priorität. Dadurch wird vermieden, dass unerwartet lange Laufzeiten eines MPs die CPU-Zeit konsumiert, die die anderen MPs benötigen, um wenigstens ihre ECET als Mindestausführungszeit zu erhalten. Somit ist die ECET von allen Threads auch die garantierte Ausführungszeit. Abbildung 1 zeigt die zeitliche Aufteilung der Threadausführung des Task-Pair Schedulings mit dem TAFT-Scheduler.

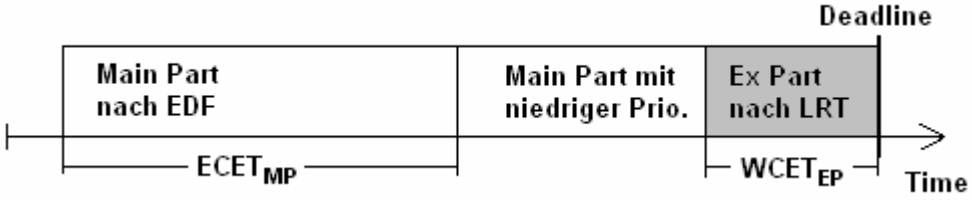

Abbildung 1: Task-Pair Scheduling

Damit stellt der TAFT-Scheduler eine ideale Umgebung für periodische Threads mit variablen Laufzeiten bereit. Es existiert eine Implementierung des TAFT-Schedulers

für RTLinux, die den Standard-Scheduler ersetzt. Dieser dient mit als Grundlage für die Umsetzung der im Rahmen dieser Arbeit entwickelten Konzepte.

2.2 Gruppenkommunikationsprotokoll RGCP

Mobile autonome Roboter oder Systeme, die in einer Gruppe agieren und kooperieren, müssen sich untereinander verständigen können. Dazu sind sie auf ein geeignetes Kommunikationsmedium und Protokoll angewiesen. Bedingt durch die Mobilität und Autonomie der Systeme eignet sich vor allem ein drahtloses Kommunikationsmedium, wie das WLAN [6]. Da die Roboter unter harten Echtzeitbedingungen agieren, stellt das die Anforderung der Echtzeitfähigkeit an das Kommunikationsprotokoll. Das RGCP-Protokoll (Real-Time Group Communication Protocol) [4][5] für das lokale Funknetz WLAN erfüllt diese Anforderungen. Es wurde für diese Zwecke entwickelt.

Das Ziel des RGCP ist es, zeitliche Vorhersagbarkeit und Zuverlässigkeit der Kommunikation zu gewährleisten. Das Problem dabei sind die hohen Verlustraten im Funknetzwerk. Die Zuverlässigkeit wird daher durch redundantes versenden der Datenpakete erreicht, bis alle Gruppenteilnehmer sie erhalten haben. Um zeitliche Vorhersagen gewährleisten zu können, wird ein maximaler Wert angenommen, der Omission Degree OD, wie oft ein Datenpaket beim Senden verloren gehen kann. Bei einem Verlust wird das Senden maximal OD mal wiederholt. Es ist daher möglich, eine obere Zeitschranke festzulegen, innerhalb derer ein Datenpaket garantiert an alle Teilnehmer übertragen wurde, wenn es nicht mehr als OD-mal verloren gegangen ist. Wenn ein Datenpaket nach (OD+1)-mal Versenden immer noch nicht von allen Teilnehmern empfangen wurde, so gilt es als endgültig verloren. Die Übertragung der Datenpakete erfolgt nach dem „Alle oder keiner“ Prinzip. Dass heißt, es muss von allen Teilnehmern empfangen worden sein oder es ist für alle Teilnehmer verloren gegangen. Somit wird garantiert, dass allen Teilnehmern die gleichen Daten zur Verfügung stehen, was für die Kooperation wichtig ist, da sie so Entscheidungen treffen können, die mit der gemeinsamen Teamstrategie konform sind.

Ein RGCP Netzwerk besteht aus einem Accesspoint AP und mehreren mobilen Stationen (Abbildung 2). Der AP besitzt dabei eine zentrale Rolle. Er ist dafür verantwortlich, dass alle Teilnehmer die Datenpakete erhalten. Diese bestätigen ihm den Empfang durch Acknowledges Acks. Die Teilnehmer müssen sich im Sendebereich des APs aufhalten.

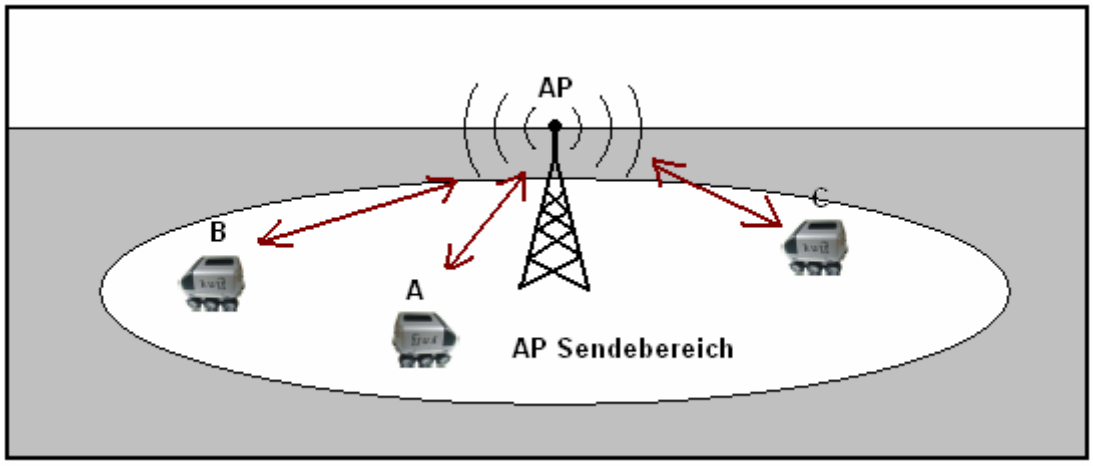

Abbildung 2: Aufbau eines RGCP Netzwerkes

Die Kommunikation läuft Rundenweise (Abbildung 3) ab. In je einer Runde wird jede Station einmal vom AP gepollt. Die betroffene Station sendet daraufhin ihre Daten zum AP, vorausgesetzt es liegen Daten zum versenden bereit. Dazu kommt noch ein Feld mit Acks, mit denen der Empfang der Daten der vorangegangen Runde bestätigt werden. Schließlich broadcastet der AP die Daten an alle Teilnehmer.

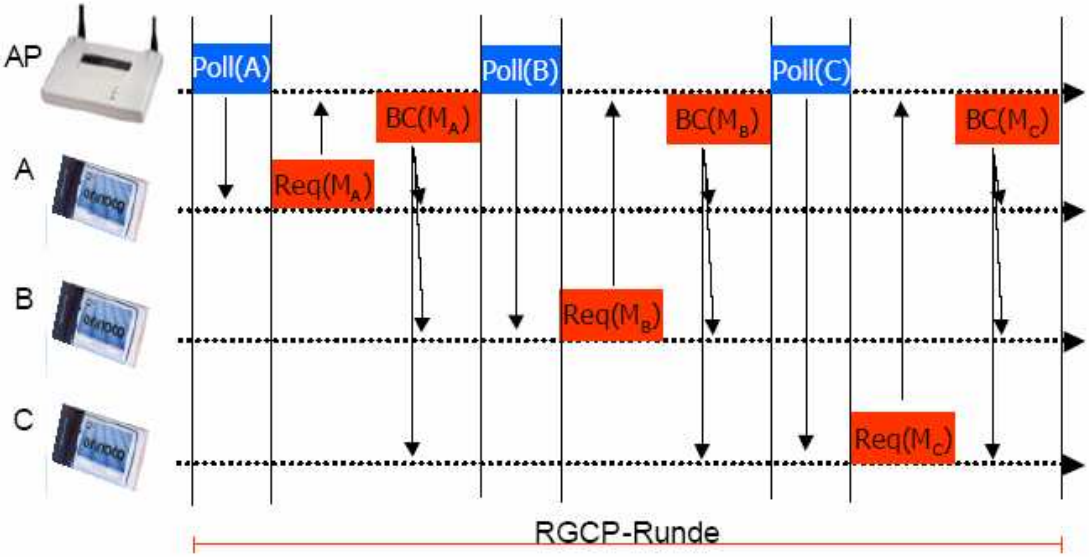

Abbildung 3: Ablauf des RGCP Protokolls [29]

Auf Grund der Eigenschaften des RGCPs kommt es im Rahmen dieser Arbeit für die Umsetzung der Kommunikation in der mehrstufigen, verteilten Datenverarbeitung zum Einsatz. Dafür steht eine Implementierung des RGCPs für RTLinux zur Verfügung.

2.3 Uhrensynchronisation im lokalen Funknetzwerk

Echtzeitsysteme agieren mit der Auflage, zeitliche Schranken einhalten zu müssen. Zu diesem Zweck verwenden sie Uhren mit ausreichender Genauigkeit. In der Regel existieren auf den Systemen bereits solche Uhren. Diese Systemuhren sind allerdings nur lokal gültig. Verteilte Echtzeitsysteme, die ihre Aktionen untereinander zeitlich koordinieren müssen, benötigen dafür jedoch global gültige Uhren. Solche Uhren können mittels Uhrensynchronisation bereitgestellt werden. Die Synchronisation setzt ein vorhandenes Kommunikationsmedium voraus. Mobile Roboter, die über ein Funknetzwerk kommunizieren, verwenden daher die Uhrensynchronisation im Funknetzwerk. Für ein lokales Funknetzwerk WLAN wurde demgemäß ein Echtzeitfähiges Protokoll zur Uhrensynchronisation [7][8] entwickelt.

Die Uhrensynchronisation im lokalen Funknetzwerk stellt eine virtuelle globale Uhr zu Verfügung, die eine garantierte Genauigkeit besitzt. Das heißt, die virtuellen Uhren auf den verschiedenen verteilten Systemen weichen maximal um einen festen Wert von einander ab. Während der Synchronisation werden rückwärtige Sprünge in den Zeitwerten der Uhren und große Sprünge vorwärts vermieden, indem die Uhrenkorrektur über das gesamte Synchronisationsintervall, durch Anpassung der Zählraten der Uhren, verteilt wird. Eine weitere Herausforderung ist die hohe Störanfälligkeit des Funknetzwerkes. Die Uhrensynchronisation muss daher Nachrichtenverluste tolerieren können. Diese Toleranz wird durch Redundanzen in mehreren Nachrichtenpaketen erzielt.

Für das Uhrensynchronisationsprotokoll (Abbildung 4) ist eine Rollenverteilung festgelegt. Es existiert darin ein Master und beliebig viele Slaves. Die Synchronisation ist in Runden aufgeteilt. In jeder Runde sendet der der Master eine Indikationsnachricht an alle Slaves in Form eines Broadcasts. Alle Teilnehmer (Slaves und Master) nehmen beim Empfang dieser Nachricht lokale Zeitstempel. Daraufhin sendet der Master diesen Zeitstempel an alle Slaves, die diesen mit ihren eigenen Zeitstempeln vergleichen und passen darauf basierend ihre Raten der virtuellen Uhren an. Um Bandbreite zu sparen, und den Aufwand durch senden der Nachrichten zu verringern, dient diese Nachricht schon als Indikation für die nächste Synchronisationsrunde. In jeder Indikationsnachricht werden also die Zeitstempel von den letzten n Runden (n=OD+1) übertragen. Auf Grund dieser Redundanz können damit bis zu OD aufeinander folgende Nachrichtenverluste toleriert werden. OD (omission degree) ist die obere Grenze der Anzahl von aufeinander folgenden Nachrichtenverlusten auf dem Kommunikationsmedium, die auftreten können.

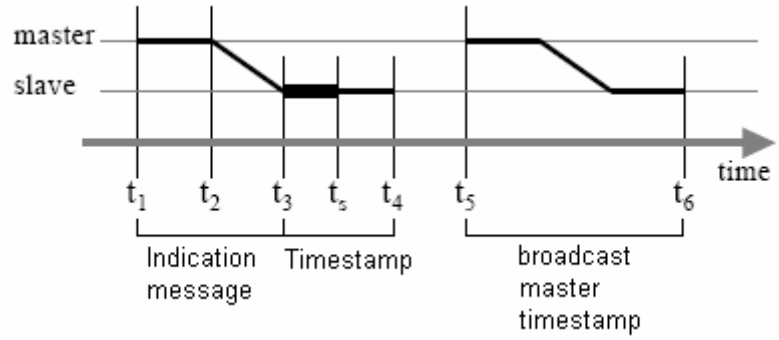

Abbildung 4: Protokoll der WLAN Uhrensynchronisation [8]

Für diese Arbeit steht eine Implementierung dieser Uhrensynchronisation für RTLinux bereit. Diese ist in den Treiber der WLAN Karte rt_orinoco integriert. Da dieser Treiber auch für das RGCP verwendet wird, läst sich die Uhrensynchronisation und das RGCP problemlos auf den Zielsystemen vereinen.

2.4 RTLinux

RTLinux (Realtime-Linux) [26] ist ein Echtzeit-Betriebssystem, was eine Laufzeitumgebung für harte Echtzeitsysteme mit POSIX Unterstützung bereitstellt. Der Kernel von RTLinux ist dabei ein Linux-Kernel, der mittels RTLinux-Patch zum RTLinux-Kernel erweitert wird. Der Patch wird von FSMLabs bereitgestellt. Die Funktionalität des RTLinux Betriebsystems wird mittels Module dynamisch in den Kernel eingebunden. Der Vorteil ist dabei, dass die Quelltexte von RTLinux frei zugänglich sind. Das erhöht die Transparenz des Betriebssystems. Der Datenfluss wird besser sichtbar. Es können Fehler korrigiert, Erweiterungen vorgenommen oder Komponenten entfernt werden. Das Betriebssystem kann auf diese Weise optimal an die Bedürfnisse der jeweiligen Echtzeitanwendung angepasst werden.

Der RT-Kernel (Abbildung 5) teilt sich in mehrere, austauschbare Komponenten auf. Zum Start des Kernels werden diese Komponenten als Kernel-Module eingebunden. Der Start des RT-Kernels erfolgt durch die Initialisierung und die Übernahme der Hardware und Interrupt-Behandlung. Der bisherige Linux-Kernel wird als Echtzeitprozess mit niedrigster Priorität weitergeführt. Die Prozesse innerhalb des Linux-Kernels sind im RTL-Kontext und nach außen nicht sichtbar. Das Scheduling dieser Prozesse übernimmt wie bisher der interne Linux-Scheduler. Interrupts des Linux-Kernels werden erst bearbeitet, wenn die Echtzeit-Threads sowie Echtzeit-Interrupts bearbeitet wurden. Der Linux-Kernel kann bei Bedarf komplett abgeschaltet werden. Die Komponenten des RTLinux Betriebssystems stellen Funktionalitäten bereit, die die folgenden Bereiche abdecken:

- Starten und Beenden von Prozessen
- Interruptbehandlung und Interruptsteuerung

- Scheduling und Zeitfunktionen
- Interprozesskommunikation der Echtzeitprozesse

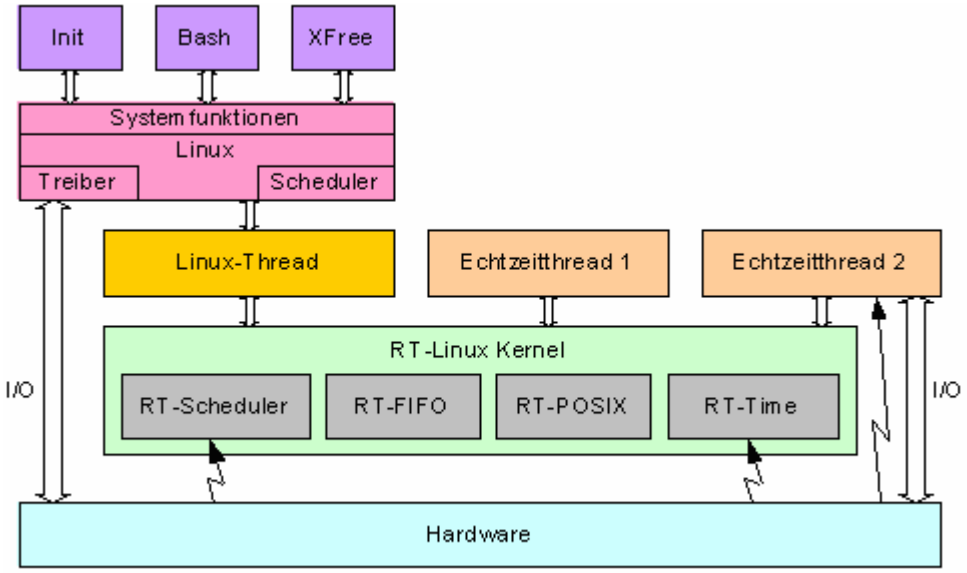

Abbildung 5: RT-Linux Kernel und RT-Module

Für diese Arbeit wird die Version 3.1 des RTLinux Betriebsystems als Grundlage für die Implementierung der hier entwickelten Konzepte verwendet. Ein Grund dafür ist unter anderem, dass weitere Grundlagen, wie das RGCP, die Uhrensynchronisation und der TAFT-Scheduler als RTLinux-Versionen zur Verfügung stehen. Der TAFT-Scheduler ersetzt den Standardscheduler von RTLinux. RGCP und die WLAN Uhrensynchronisation werden durch Einbindung der entsprechenden Module hinzugefügt.

2.5 Mehrstufige, verteilte Sensorfusion von Laserscandaten

Die Mehrstufige, verteilte Sensorfusion von Laserscandaten [15] wurde entwickelt, um Sensordaten unter Echtzeitbedingungen zu verarbeiten, die von Laserscannern mobiler Roboter erhalten werden. Das Anwendungsszenario ist dabei die Teamarbeit mehrerer Roboter im Robocup. Das Ziel ist es, aus den rohen Sensordaten ein Weltbild abzuleiten, auf dessen Basis das System agieren und reagieren kann. Das geschieht durch eine mehrstufige Verarbeitung (Abbildung 6), bei der die Daten mehrere Filter durchlaufen und dabei jedes Mal weiter abstrahiert werden, um die Datenmenge zu verringern und die relevanten Informationen zu extrahieren. Die Filter, die dabei durchlaufen werden sind in dieser Reihenfolge der Konturfilter, der Objektfilter und der Elementfilter. Es existieren damit vier Datenformate, die nacheinander erstellt werden.

1. **Punkte:** Dieses Datenformat wird vom Laserscanner geliefert. Es beschreibt die Umgebung mit einer begrenzten Anzahl von Punkten. Jeder Punkt markiert da-

bei die Position, an der der Laserimpuls des Laserscanners auf ein Hindernis gestoßen ist und reflektiert wurde.

2. **Konturen:** Dieses Datenformat beschreibt die Umgebung durch eine Menge von Kanten und Bögen. Kanten repräsentieren die Existenz von glatten, geraden Oberflächen. Ihre Position wird durch je einen Anfangspunkt und einen Endpunkt festgehalten. Bögen repräsentieren die Existenz von gebogenen Oberflächen. Sie besitzen einen Radius, einen Anfangs- und Endwinkel und die Position des Bogenmittelpunktes wird festgehalten. Bögen und Kanten werden aus den Punktdaten extrahiert, indem der Konturfilter Punkte zu Gruppen zusammenfasst, die zu einer Kante oder einem Bogen gehören.

3. **Objekte:** Die Kanten und Bögen werden vom Objektfilter nach Zugehörigkeit, zu in der Umgebung befindlicher Objekte, sortiert und zusammengefasst. Bögen bilden kreisförmige Objekte und Kanten bilden ein Polygon, was den Umriss kantiger Objekte beschreibt.

4. **Elemente:** Die Objekte werden vom Elementfilter als bekannte Elemente identifiziert. Für das Robocup-Szenario wären das die Spielfeldbegrenzung, der Ball oder die Spielerroboter. Dieses Datenformat stellt das endgültige Weltbild dar.

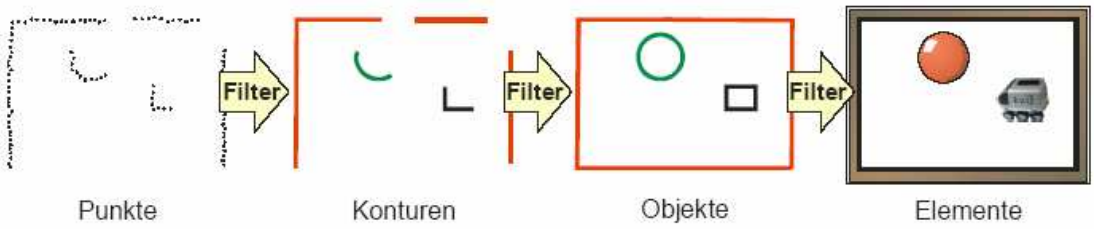

Abbildung 6: Mehrstufige Verarbeitung von Laserscanndaten [15]

Die Wahrnehmung der Sensoren und das resultierende Weltbild können unvollständig sein, bedingt durch den beschränkten Blickwinkel, die begrenzte Auflösung und Reichweite der Laserscanner oder durch die Wahrnehmungsschatten, die durch Verdeckungen, verursacht durch Hindernisse, entstehen können. Daher wurde die Idee verfolgt, die Daten mehrer Roboter zu fusionieren, um die Daten zu vervollständigen und allen Robotern die gleichen Sensordaten zur Verfügung zustellen. Das soll ihnen ermöglichen, eine gemeinsame Teamstrategie aufzustellen. Die Durchführung der Fusion (Abbildung 7) kann dabei auf jeder beliebigen Abstraktionsstufe der Daten gewählt werden. Dabei gilt in der Regel, je niedriger die Abstraktionsstufe, auf der fusioniert wird, also je eher die Fusion durchgeführt wird, desto genauer ist schließlich das resultierende Weltbild. Jedoch ist der Preis dafür die höhere Laufzeit der Fusions- und Filterstufen und die größeren Datenmengen, die zum Zweck der Fusion übertragen und verarbeitet werden müssen. Da es vier Abstraktionsstufen oder auch Datenformate gibt, existieren auch vier Fusionsstufen, die Punktfusion, Konturfusion, Objektfusion und die Elementfusion.

1. **Punktfusion:** Auf dieser Stufe werden die Punktdaten von allen beteiligten Laserscannern vermischt und so sortiert, als würden sie von einem Laserscanner stammen.

2. **Konturfusion:** Auf dieser Stufe werden die Konturen fusioniert. Die Kanten und Bögen werden zusammengemischt. Bögen, die an denselben Positionen liegen, werden zu einem Bogen zusammengefasst. Kanten, die sich überlappen, werden zu einer größeren Kante fusioniert.

3. **Objektfusion:** Hier werden die Objekte fusioniert. Sie werden dabei auf Überlappung untersucht. Existiert ein gewisser Grad an Überlappung, werden die entsprechenden Objekte zu einem zusammengefasst.

4. **Elementfusion:** Auf dieser Ebene werden die Elemente fusioniert. Befinden sich zwei Elemente der gleichen Art annähernd an der gleichen Position, so werden sie zu einem Element fusioniert.

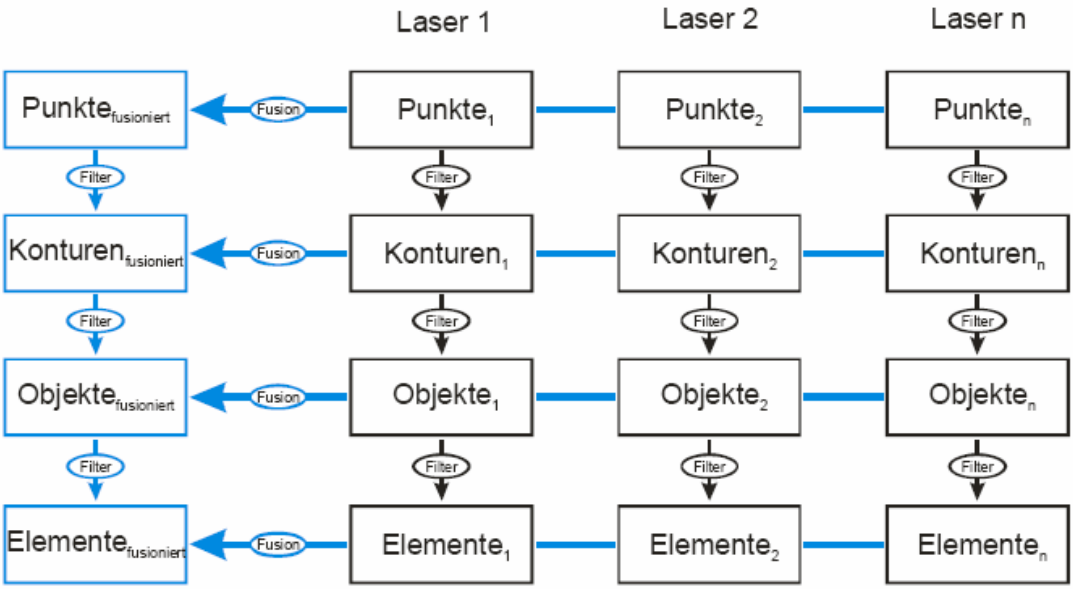

Abbildung 7: Filter und Fusionsstufen [15]

Alle Algorithmen der Fusionsstufen und Filter wurden als Anytime-Algorithmen konzipiert. Dass heißt, sie stellen ein Berechnungsergebnis bereits am Anfang ihrer Ausführung bereit, welches sie im Laufe der Ausführung stetig verbessern. Der Grund dafür sind die stark variablen Ausführungszeiten der Algorithmen. Die Laufzeiten haben daher meist sehr hohe maximale Ausführungszeiten WCET, die aber weit über den im durchschnitt tatsächlich auftretenden Zeiten liegen. Durch die Konzipierung als Anytime-Algorithmen ist es jedoch möglich, der Berechnung Zeitschranken aufzulegen, die weit unter den WCET Zeiten liegen. Droht nun die Berechnungsdauer die Zeitschranke zu überschreiten, kann sie abgebrochen werden und das bis dahin erreichte Teilergebnis kann ausgegeben werden. Für den eventuellen rechtzeitigen Abbruch ist in der mehrstufigen, verteilten Sensorfusion von Laserscandaten der TAFT-Scheduler verantwortlich.

Aus der mehrstufigen, verteilten Sensorfusion von Laserscandaten wurden die möglichen Probleme (zeitliche Inkohärenzen der Sensordaten, Verarbeitung unter harten Echtzeitbedingungen) abgeleitet, die allgemein bei der mehrstufigen, verteilten Datenverarbeitung auftreten können. Diese Probleme dienten daher mit als Grundlage für die Motivation dieser Arbeit und der darin entwickelten Lösungen. Außerdem wird sie daher auch als Anwendungsbeispiel zur Überprüfung der entwickelten Lösungen dieser Arbeit herangezogen.

3 Verwandte Arbeiten

In diesem Kapitel werden wissenschaftliche Forschungsarbeiten vorgestellt, die sich mit ähnlichen Problematiken beschäftigen, wie sie dieser Diplomarbeit zu Grunde liegen. Im ersten Abschnitt werden die zwei Typen von System, die Ereignisgesteuerten Systeme und die Zeitgesteuerten Systeme, vorgestellt, verglichen und es wird auf die Eignung als Lösung, für die in dieser Arbeit gestellten Aufgaben, eingegangen. Im zweiten Abschnitt wird das Kalman-Filtering Verfahren [19] vorgestellt, was im Anwendungsbeispiel für die entwickelten Konzepte umgesetzt wurde, mit dem Ziel, zeitliche Diskrepanzen der Sensordaten auflösen zu können.

3.1 Ereignisgesteuerte und Zeitgesteuerte Systeme

Verteilte Echtzeitsysteme lassen sich in zwei Typen unterteilen, die Ereignisgesteuerten Systeme(event-triggered) und die Zeitgesteuerten Systeme(time-triggered). Die Aktivitäten der Ereignisgesteuerten Systeme werden initiiert, wenn ein signifikantes Ereignis eintritt. Im Zeitgesteuerten System werden dagegen die Aktivitäten periodisch zu vorbestimmten Zeitpunkten ausgeführt. Im Forschungsbericht „Event-Triggered versus Time-Triggered Real-Time Systems“ [11] wurden die Eigenschaften zwischen diesen beiden Systemarten verglichen. Der Vergleich konzentriert sich besonders auf die Punkte der Vorhersagbarkeit, Testbarkeit, der Auslastung der Ressourcen, der Erweiterbarkeit und den Folgen bei Verletzungen der Systemannahmen.

Ereignisgesteuerte Systeme begründen ihre Aktivitäten auf das Auftreten eines signifikanten Ereignisses. Das kann zum Beispiel eine Änderung des Zustandes eines beobachteten Objektes in der Umgebung sein, oder das Empfangen von Daten über ein Ereignisgesteuertes Kommunikationsprotokoll, bei dem der Empfänger den Zeitpunkt nicht vorhersehen kann, wann der Sender Daten überträgt. Es werden zwei Arten von Ereignissen unterschieden, die vorhersehbaren und die zufälligen Ereignisse. Vorhersehbar bedeutet, dass die Zeitpunkte des Auftretens der Ereignisse bekannt sind. Somit können notwendige Systemressourcen im Voraus reserviert werden, um die rechtzeitige Ausführung der entsprechenden Aktionen zu garantieren. Das Auftreten zufälliger Ereignisse kann nicht genau vorhergesagt werden. Es kann lediglich mittels Wahrscheinlichkeit geschätzt werden. Es ist möglich, dass Ereignisse über einen langen Zeitraum nicht eintreten. Jedoch kann es genauso passieren, dass sie nahezu zeitgleich in großer Zahl gehäuft (Peakload) auftreten. Möglicherweise können die Ereignisse nicht alle rechtzeitig verarbeitet werden, auf Grund begrenzter Systemressourcen, die im Ereignisgesteuerten System in der Regel mittels dynamischer Schedulingverfahren zugeteilt werden. Mit Flow Control Mechanismen können solche Überlastungen vermieden werden. Dabei wird die Frequenz der Ereignisse an die Verarbeitungsgeschwindigkeit des Systems angepasst. Zum Beispiel, wenn der Sender die Datenübertragungsraten an

die des Empfängers anpasst. Meist ist es jedoch nicht möglich, auf die Häufigkeit von zufälligen Ereignissen Einfluss zu nehmen. Jedoch können Überlastungen dann vermieden werden, wenn zu häufig auftretende Ereignisse, teilweise oder ganz, ignoriert werden.

In Zeitgesteuerten Systemen werden die Aktivitäten zu festen bekannten Zeiten ausgeführt. Diese werden meist periodisch wiederholt, um Zustandsänderungen eines beobachten Objektes der Umgebung wahrzunehmen. Damit auf diese Zustandsänderungen rechtzeitig reagiert werden kann, muss die Pollrate der Dynamik des Objektes entsprechen. Alle Aktionen im System werden nach einem vorbestimmten Zeitplan aktiviert, in dem die jeweiligen nötigen Systemressourcen reserviert werden, so dass eine rechtzeitige Abarbeitung aller Aufgaben stets garantiert werden kann. Zeitgesteuerte Systeme besitzen implizite Flow Conrol Mechanismen, da Objektzustände der Umwelt mit monoton gleich bleibenden Raten gepollt werden. Dadurch zeichnen sie sich durch einen hohen Grad an Vorhersagbarkeit aus. Jedoch können kurzlebige Zustände des beobachteten Objektes übersehen werden, wenn eine Situation eintritt, bei der es zu vermehrten Zustandsänderungen kommt. Kommunikationsprotokolle sind in Zeitgesteuerten Systemen in der Regel ebenfalls Zeitgesteuerter Art. Die Sender bekommen nach vorherbestimmten Zeitplänen den Zugang zum Kommunikationsmedium (Beispielsweise durch ein „synchronous time division multiple access“ STDMA Protokoll), um rechtzeitige Übertragung aller Daten im System gewährleisten zu können.

In Tabelle 1 sind die Vergleiche in den einzelnen Punkten zwischen Ereignisgesteuerten und Zeitgesteuerten Systemen aus [11] zusammengefasst aufgelistet.

	Ereignisgesteuerte Systeme	**Zeitgesteuerte Systeme**
Vorhersagbarkeit	• Geringere Vorhersagbarkeit • Überschreitung der Deadline bei Peakload Situationen möglich	• Präzise Vorhersagbarkeit • Rechtzeitigkeit durch statische „Schedules“
Testbarkeit	• Extensive Systemtests erforderlich • Reproduktion von Applikationsszenarien ist schwierig • Es bleibt immer eine Restunsicherheit	• Geringerer Testaufwand • Eingabefälle können leicht reproduziert werden
Ressourcenauslastung	• Bessere Auslastung bei niedrigen und durchschnittlichen Belastun-	• Schlechtere Auslastung bei niedrigen und durchschnittli-

	gen • Überlastung in Peakload Situationen	chen Belastungen • Geplant für Peakload Situationen
Erweiterbarkeit	• Erweiterungen fordern meist erneute extensive Systemtests	• Neuberechnung des statischen Ausführungsplans (Schedul)
Verletzung der Systemannahmen	• Verletzung der Systemannahmen können zum Verlust der Rechtzeitigkeit führen	• Kenntnisse über kurzlebige Zustände des externen Systems können verloren gehen

Tabelle 1: Vergleich Ereignisgesteuerte vs. Zeitgesteuerte Systeme

In den meisten Fällen ist der Grund für die Wahl einer Zeitgesteuerten Lösung die Vorhersagbarkeit, während die Wahl für ein Ereignisgesteuertes System vorwiegend auf Grund der besseren Ressourcenauslastung geschieht.

Die Vorhersagbarkeit war Beispielsweise auch der Grund für die Wahl einer Zeitgesteuerten Lösung für das Sensorfusionssystem in der Arbeit „Sensorfusion in Zeitgesteuerten Systemen“[10]. Die Schwerpunkte liegen dabei darin, Die Zuverlässigkeit durch die Sensorfusion zu erhöhen und die Fusion unter Echtzeitbedingungen auszuführen. Das entwickelte System wurde dann an einem selbstfahrenden Roboter mit Infrarot und Ultraschallsensoren getestet, wobei eine höhere Zuverlässigkeit, durch die Zusammenführung der Daten vieler Sensoren, erreicht wurde.

Jedoch schließt die Verwendung von Ereignisgesteuerten Systemen nicht grundsätzlich eine gute Vorhersagbarkeit aus. Es ist auch nicht immer notwendig, sich entweder ausschließlich für eine Zeitgesteuerte oder ausschließlich für eine Ereignisgesteuerte Lösung zu entscheiden. Es existieren bereits eine Menge von Forschungsarbeiten, die sich mit der Entwicklung von hybriden Systemen, also Systeme, die Zeitgesteuerte und Ereignisgesteuerte Lösungen vereinen, beschäftigen. In der Arbeit „Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems“ [12] Beispielsweise wurde eine Lösung eines hybriden Systems entwickelt, in der Ereignisgesteuerte Tasks und Zeitgesteuerte Tasks über ein Busprotokoll kommunizieren, was in dynamische (feste Prioritäten) und statische Phasen (zyklischer statischer Plan) unterteilt ist.

Im Rahmen dieser Diplomarbeit fiel die Wahl auf eine Zeitgesteuerte Herangehensweise für die Lösung der gestellten Aufgaben. Der Grund ist zum einen die hohe Vorhersagbarkeit, die bei einer Zeitgesteuerten Lösung leicht erzielt werden kann und für Echtzeitsysteme wichtig ist, zum anderen ist es die Verwendung des TAFT-Schedulers,

der selbst schon ein Zeitgesteuertes System darstellt, da die Task-Paare darin periodisch aktiviert werden.

3.2 Kalman-Filtering und Multi Object Tracking

Das Kalman-Filtering [19] Verfahren wurde 1960 als rekursive Lösung für ein Problem entwickelt, was „discrete data linear filtering problem“ genannt wird. Dieser Filter wurde für die Erfüllung von Aufgaben konzipiert, wie das Vorhersagen von Zufallssignalen, die Trennung von Zufallssignalen und zufälligem Rauschen und das Erkennen von Signalen bekannter Form (Sinus, Puls). Das Kalman-Filtering Verfahren hat seit seiner Entwicklung in vielen Bereichen Anwendung gefunden, unter anderem in der autonomen oder unterstützen Navigation. Folgendes ist eine kurze Einführung [20] in die Funktionsweise dieses Verfahrens.

Das Kalman-Filtering Verfahren beschäftigt sich mit dem allgemeinen Problem, den Zustand $\mathbf{x} \in \mathbf{R}^n$ eines kontrollierten Prozesses zu diskreten Zeitpunkten zu bestimmen. Der Prozess kann mit folgender Gleichung dargestellt werden,

$$\mathbf{x}_k = \mathbf{A}\,\mathbf{x}_{k-1} + \mathbf{w}_{k-1} \qquad \textbf{(3.1)}$$

mit der Messung $\mathbf{y} \in \mathbf{R}^m$.

$$\mathbf{z}_k = \mathbf{H}\,\mathbf{x}_k + \mathbf{v}_k \qquad \textbf{(3.2)}$$

Die Zufallsvariablen $\mathbf{w}_k$ und $\mathbf{v}_k$ repräsentieren das Prozessrauschen und das Rauchen, das die Messung beeinflusst. Sie werden als von einander unabhängige Normalverteilte Zufallsgrößen angenommen. Die $\mathbf{n} \times \mathbf{n}$ Matrix **A** bildet einen Zustand des vorherigen Schrittes **k-1** auf den Zustand des aktuellen Schrittes **k** ohne Einbezug des Prozessrauschens ab. Die $\mathbf{m} \times \mathbf{n}$ Matrix **H** stellt die Relation von den Zuständen zu den Messungen dar.

Der Zustandsvektor $\mathbf{x}_k$ ist unbekannt. Ziel ist es nun, eine möglichst genaue Schätzung dieses Vektors abzugeben. Diese Schätzung wird einmal mit Kenntnissen über den Prozess durchgeführt, wobei dieser Vektor $\tilde{\mathbf{x}}_k \in \mathbf{R}^n$ „priori state estimate“ genannt wird, und zum zweiten mit Hilfe des gegeben Messvektors $\mathbf{z}_k$. Dieser Vektor $\hat{\mathbf{x}}_k \in \mathbf{R}^n$ wird „posteriori state estimate“ genannt. Die Schätzung $\hat{\mathbf{x}}_k$ wird aus einer Linearkombination aus dem Vektor $\tilde{\mathbf{x}}_k$ und der gewichteten Differenz zwischen den aktuellen Messungen $\mathbf{z}_k$ und den vorhergesagten Messungen $\mathbf{H}\tilde{\mathbf{x}}_k$ berechnet.

$$\hat{\mathbf{x}}_k = \tilde{\mathbf{x}}_k + \mathbf{K}\left(\mathbf{z}_k - \mathbf{H}\tilde{\mathbf{x}}_k\right) \qquad \textbf{(3.3)}$$

Die Differenz $(\mathbf{z_k} - \mathbf{H}\tilde{\mathbf{x}}_\mathbf{k})$ ist der so genannte „residual“ und ist die Diskrepanz zwischen den aktuellen und den vorhergesagten Messwerten. Die Matrix **K** ist der so genannte „gain“.

Der Ablauf des Kalman-Filtering Verfahren besteht aus beliebig vielen Zyklen, wobei in jedem Zyklus ein Prediction Schritt und ein Update Schritt ausgeführt wird. Im Prediction Schritt wird aus dem Zustand der Vergangenheit der aktuelle Zustand vorhergesagt. Es wird also der Vektor $\tilde{\mathbf{x}}_\mathbf{k}$ ermittelt. Mit Hilfe der aktuellen Messungen $\mathbf{z_k}$ wird anschließend im Update Schritt der Vektor $\hat{\mathbf{x}}_\mathbf{k}$ berechnet. Diese beiden Schritte sind in Abbildung 8 mit den entsprechenden Gleichungen dargestellt.

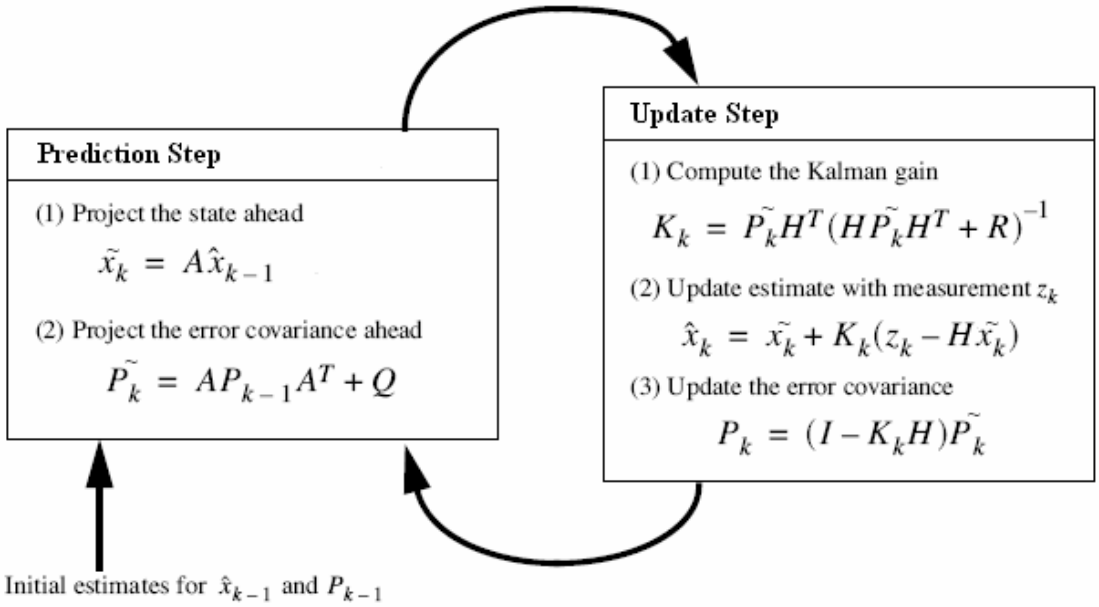

Abbildung 8: Kalman-Filtering Verfahren [20]

Die Matrix $\tilde{\mathbf{P}}_\mathbf{k}$ und die Matrix $\mathbf{P_k}$ sind die Kovarianzen. Die Kovarianz $\tilde{\mathbf{P}}_\mathbf{k}$ wird als „priori estimate error covariance“ bezeichnet. Die Kovarianz $\mathbf{P_k}$ wird als „posteriori estimate error covariance“ bezeichnet und wird mit Hilfe der Matrix **K** minimiert. Mit den Tuning Parametern **Q** und **R** ist es möglich, den Einfluss der Messwerte und des Prozessmodels im Kalman-Filtering Verfahren zu kalibrieren.

Das Kalman-Filtering Verfahren wurde häufig in Applikationen eingesetzt, in denen Objekte der Umwelt mittels Sensoren periodisch wahrgenommen werden, um deren Positionen über einen längeren Zeitraum verfolgen zu können. Beispielsweise wurden die Positionen von Roboter und anderen Objekten im Robocup [21][22] auf diese Weise verfolgt. Dabei existiert jedoch noch ein Problem, denn es existieren in diesem Szenario mehrere gleichartige Objekte, deren Positionen verfolgt werden sollen. Über die Sensoren werden nun mehrere Positionen wahrgenommen, die im jeden Update Schritt je einem Objekt zugeordnet werden müssen. Diese Problematik ist Gegenstand des Forschungsthemas „Multi Object Tracking“. In der Forschungsarbeit „Watch their moves: applying probabilistic multiple object tracking to autonomous robot soccer“ [22]

wurde ein Tracking Algorithmus (MHT Algorithmus) vorgestellt, der erfolgreich im Robocup 2001 world championship eingesetzt wurde.

Der Multiple Hypothesis Tracking Algorithmus (Abbildung 9: MHT Algorithmus) verfolgt die Positionen gegnerischer Roboter durch die Aufstellung mehrerer Hypothesen der Positionszuordnung. Jede Hypothese repräsentiert eine mögliche Assoziation von gemessen Positionen zu den Objekten. In jeder Iteration des Algorithmus wird eine Menge von Hypothesen aufgestellt, zu denen je ein Kalman Filter unterhalten wird. Es werden dann die Hypothesen mittels Prediction auf den aktuellsten Stand gebracht. Darauf aufbauend wird die Hypothesenmenge der nächsten Iteration erstellt, indem ein Updateschritt mit den aktuell gemessenen Positionen durchgeführt wird. Die Zuordnung von Messungen zu den Objekten erfolgt auf Basis einer statistischen Distanzmessung (Mahalanobis Distanz). Da die Hypothesen einen Baum aufspannen, der mit jeder Iteration wächst, wird außerdem ein Pruning Verfahren angewandt, um dieses Wachstum zu begrenzen. Dabei werden unwahrscheinliche Hypothesen verworfen und gehen nicht in die weitere Verarbeitung mit ein.

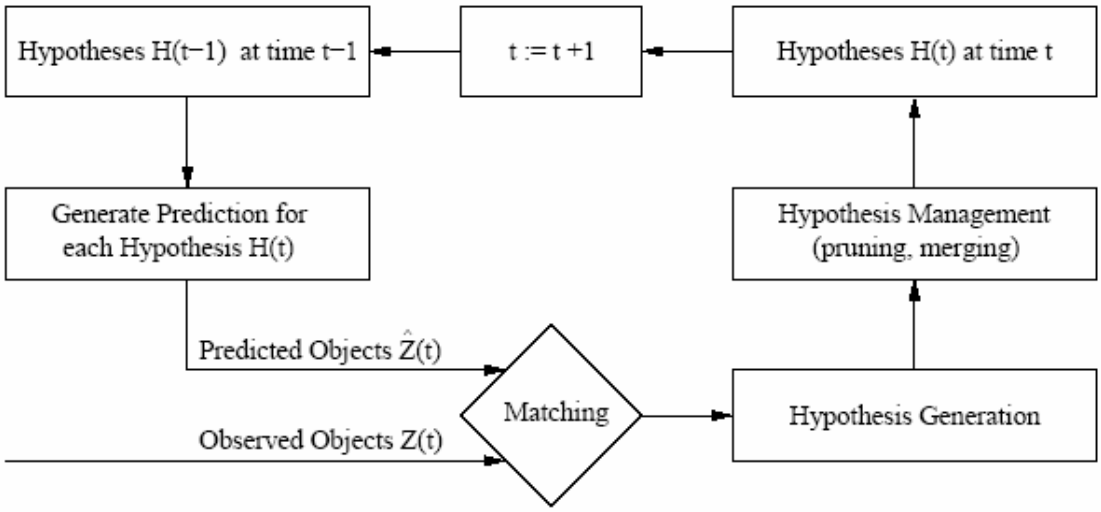

Abbildung 9: Multiple Hypothesis Tracking Algorithmus [22]

Für ein ähnliches Problem wurde im Rahmen der Arbeit „Data association in stochastic mapping using the joint compatibility test" [23] ein Verfahren angewandt, was „Joint Compatibility Test" genannt wird. Es wurde experimentell mit einer klassischen Nearest Neighbor Methode verglichen. Es erwies sich dabei als ein effizientes Verfahren, um die beste Lösung der Datenassoziation zu finden.

Das Kalman-Filtering Verfahren wird im Rahmen dieser Arbeit für das Anwendungsbeispiel der mehrstufigen, verteilten Sensorfusion von Laserscandaten [15], genauer für die Elementfusion, angewandt. Dabei werden besonders die Eigenschaften des Prediction Schritts genutzt, um zeitliche Diskrepanzen zwischen den Daten der einzelnen Teilnehmer aufzulösen. Für das „Multi Object Tracking" Problem wurde ein eigenes, einfacheres Verfahren entwickelt, um den inhaltlichen und zeitlichen Rahmen dieser Diplomarbeit nicht zu sprengen.

4 Rechtzeitigkeit der Datenverarbeitung

Dieses Kapitel beschäftigt sich mit den konzeptionellen Lösungen, mit deren Hilfe die zeitliche Garantie der Rechtzeitigkeit gewährleistet werden kann. Der erste Abschnitt beschäftigt sich mit dem Task-Konzept, welches in der ersten Teilaufgabe gefordert wird. Dieses Task-Konzept teilt die Berechnung einer Periode in Phasen auf, wobei die Reihenfolge dieser Phasen eingehalten wird und der rechtzeitige Abschluss der Bearbeitung der Daten durch alle Phasen gesichert ist. Im zweiten Abschnitt wird diesem Task-Konzept noch eine Erweiterung zugefügt, die es erlaubt, innerhalb der Phasen voneinander unabhängige Aktionen in parallelen Threads auszuführen.

4.1 Rechtzeitigkeit und Zeitgarantien für die Bearbeitungsphasen

Roboter sind auf Daten, basierend auf Sensoren, die ihre Umwelt beschreiben, angewiesen. Diese Daten benötigen sie jedoch nicht nur einmalig, sondern regelmäßig, um dynamische Prozesse und Veränderungen wahrzunehmen. Daher werden diese Aufgaben, der Aufnahme und Verarbeitung dieser Daten, von periodischen Tasks durchgeführt, deren rechtzeitige Ausführung von Echtzeit-Schedulern [27] garantiert wird. Diese benötigen meist dafür als Parameter die maximale Laufzeit **(WCET)** der Ausführungsfunktion.

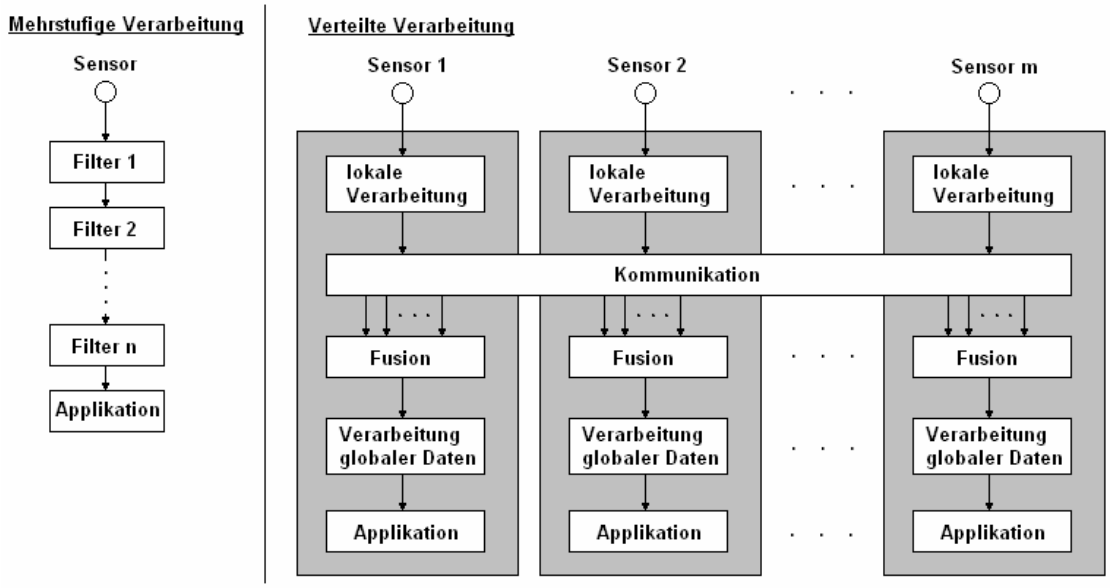

Abbildung 10: Mehrstufige und Verteilte Sensordatenverarbeitung

In der Mehrstufigen Datenverarbeitung von Sensordaten ist die Ausführung in mehrere Phasen oder auch Stufen (Abbildung 10 links) unterteilt, bei denen die Daten von ei-

nem Format in ein anderes umgewandelt werden, um bestimmte relevante Informationen aus ihnen zu extrahieren, oder es werden bestimmte Aktionen mit ihnen ausgeführt. In der verteilten Verarbeitung der Sensordaten (Abbildung 10 rechts) zum Beispiel gibt es unter anderem die Phasen der Kommunikation der Daten über ein bestimmtes Kommunikationsmedium mittels einem geeignetem Protokoll (wie Beispielsweise die Kommunikation via WLAN mittels RGCP [5]) und die Phase der Fusion der Daten aller, an der verteilten Verarbeitung beteiligten Teilnehmer. Jede Phase benötigt dabei ganz bestimmte Daten bzw. ein bestimmtes Format als Eingabe, und liefert wiederum bestimmte Daten als Ausgabe. Die Eingabedaten können dabei die Ausgabe einer anderen Phase sein und die Ausgabedaten können die Eingabe einer weiteren Verarbeitungsphase sein. In diesem Fall entsteht eine Abhängigkeit zwischen den Verarbeitungsphasen, die damit auch eine strikte Verarbeitungsreihenfolge bedingt. Aber nicht nur durch die Ein- und Ausgabebeziehungen entsteht solche Verarbeitungsreihenfolge. Sie kann auch rein aus der Logik heraus entstehen. Beispielsweise muss der Fusion von Sensordaten von unterschiedlichen Stationen unmittelbar eine Kommunikation voran gehen. Es entsteht damit eine Kette von Verarbeitungsschritten, an deren Anfang die rohen Sensordaten stehen, die durch diese Kette hindurch laufen, und am Ende werden der Anwenderapplikation die daraus gewonnenen Informationen, basierend auf den Sensordaten aller Teilnehmer im Falle der verteilten Verarbeitung, zur Verfügung gestellt. Damit dieser Mechanismus funktioniert, wird also eine Lösung in Form eines geeigneten Task-Konzeptes benötigt, welche die folgenden Garantien bereitstellt.

1. Die Verarbeitung der Sensordaten und damit die Abarbeitung der Phasenkette muss periodisch erfolgen, um stets ein möglichst aktuelles Weltbild aufrecht zu erhalten.

2. Die Verarbeitung der aktuellen Periode muss vor dem Ende der Selbigen und dem Anfang der folgenden Periode (rechtzeitig) beendet sein.

3. Innerhalb einer jeden Periode muss jede Phase genau einmal ausgeführt werden, damit die Daten bis zum Ende der Periode das gewünschte Datenformat besitzen, bzw. aus ihnen die gewünschten Informationen extrahiert werden können.

4. Die Ausführungsreihenfolge der Phasen muss strikt eingehalten werden, damit jeder Phase die gewünschten Daten bereitgestellt werden können. Nur dann ist eine erfolgreiche periodische Verarbeitung der Sensordaten durch alle Phasen möglich.

Wichtig ist, dass alle vier Garantien gegeben werden müssen. Kann nur eine Garantie nicht gegeben werden, dann kann auch insgesamt die Garantie nicht gegeben werden, dass in jeder Periode die Sensordaten erfolgreich verarbeitet werden können. Abbildung 11 zeigt grob die Zeitliche Abfolge von Perioden und Phasen.

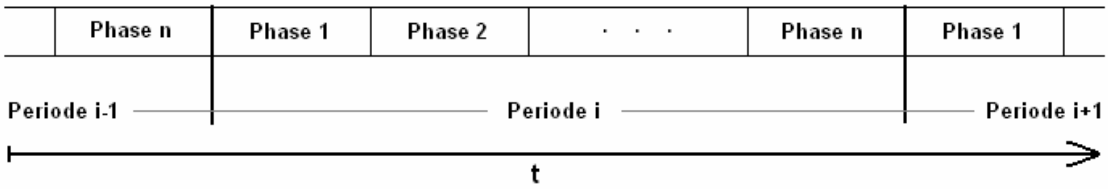

Abbildung 11: Perioden- und Phasenfolgen

Eine einfache Lösung kann schnell gefunden werden, wenn ein Echtzeit-Scheduler verwendet wird, der auf WCET Zeiten basiert, wie zum Beispiel EDF. Angenommen es gibt eine Kette von **n** Verarbeitungsphasen **{$P_1 \ldots P_n$}**, wobei die Phase **P_i** mit **$1 > i > n$** in der Ausführungsreihenfolge der unmittelbare Nachfolger von Phase **P_{i-1}** und der unmittelbare Vorgänger der Phase **P_{i+1}** ist. Unmittelbar heißt hier, es wird keine andere Phase zwischen einer Phase und ihren unmittelbaren Nachfolger bzw. Vorgänger ausgeführt. Eine Phase **P_i** gilt als Nachfolger von einer Phase **P_j**, wenn folgendes gilt:

$$\mathbf{T_START(P_i) \geq T_END(P_j)} \qquad \mathbf{(4.1)}$$

Eine Phase **P_i** gilt als Vorgänger von einer Phase **P_j**, wenn folgendes gilt:

$$\mathbf{T_END(P_i) \leq T_START(P_j)} \qquad \mathbf{(4.2)}$$

Wobei **T_START(P_i)** die Zeit ist, zu der die Ausführung von **P_i** beginnt, und **T_END(P_i)** die Zeit, bei der sie endet.

Die Ausführung dieser Phasen kann dann mit einem Thread organisiert werden. Der Scheduler sorgt dafür dass dieser Thread periodisch ausgeführt wird. In der Thread-Funktion werden die Phasen nacheinander in entsprechender Reihenfolge aufgerufen, wie es hier mit Pseudocode angedeutet ist.

```
Thread-Function:
{
      for(∞){
            call P1;
            call P2;
              :
              :
            call Pn;
            wait_for_next_period;
      }
}
```

Da jede Phase nacheinander im Kontext ein und desselben Threads ausgeführt wird, sind daher automatisch auch die vorher beschrieben Vorgänger und Nachfolger Bedingungen erfüllt. Die Garantien Nummer 3. und 4. sind damit gegeben. Da dieser Thread periodisch ist, ist auch die Garantie Nummer 1. gegeben. Die Garantie Nummer 2. kann ebenfalls gewährt werden, wenn dem Scheduler eine korrekte **WCET** als Parameter für die Berechnungszeit der Thread Funktion übergeben wird. Korrekt heißt, dass diese

Zeit in 100% aller Fälle nicht überschritten wird. Die **WCET** ist in diesem Fall die Summe der maximalen Laufzeiten der einzelnen Phasen **WCET(P_i)**.

$$\mathbf{WCET} = \sum_{i=1}^{n} \mathbf{WCET}(P_i) \qquad \textbf{(4.3)}$$

Wenn diese **WCET** dem Scheduler zur Verfügung steht, dann ist die rechtzeitige Beendigung der gesamten Verarbeitung der Sensordaten vor dem Ende der aktuellen Periode gewährleistet. Insgesamt sind damit alle vier Garantien gegeben.

Obwohl diese Lösung, basierend auf den maximalen Bearbeitungszeiten der einzelnen Phasen, ihren Zweck erfüllt, hat sie dennoch Nachteile. Die WCET ist zum einen die garantierte maximale Zeitdauer zwischen dem Aufnahmezeitpunkt der rohen Sensordaten und dem Zeitpunkt der Bereitstellung der fertig verarbeiteten Daten. Außerdem beschränkt sie auch die Wahl der Periodendauer nach unten. Genauer gesagt, die Periode kann nie kleiner als die WCET sein. Damit ergibt sich die Aussage, dass je größer die WCET, desto kleiner ist die maximale Rate pro Zeiteinheit, mit der ständig neue Sensordaten verarbeitet werden können. Die WCET kann jedoch größenmäßig weit über denen, in den meisten Fällen tatsächlich auftretenden Berechnungszeiten, liegen. Als Beispiel sei hier der Konturfilter der mehrstufigen, verteilten Sensorfusion von Laserscandaten genannt. An ihm wurden verschiedene Messungen [18] durchgeführt, um Unterschiede in den Laufzeiten, bei unterschiedlichen Gegebenheiten, in der durch die Laserscanner wahrgenommenen Umwelt, zu untersuchen. Dabei zeigten sich bei bestimmten Umweltsituationen starke Unterschiede in den Laufzeiten, besonders beeinflusst durch die Bogensuche. Bei der Filterung der Daten von einem Laserscanner lag die längste Bearbeitungsdauer bei ca. 140 ms, wo gegen 90% aller Aufgezeichneten Laufzeiten (Abbildung 12) unter 50 ms lagen. Daher sollte eine Lösung für ein Task-Konzept gefunden werden, das auf durchschnittlichen Laufzeiten der Phasen basiert, oder auf Zeitschranken, die in einem bestimmten Anteil aller Fälle nicht überschritten werden. Dennoch soll dieses Konzept auch weiterhin die vier Garantien geben können.

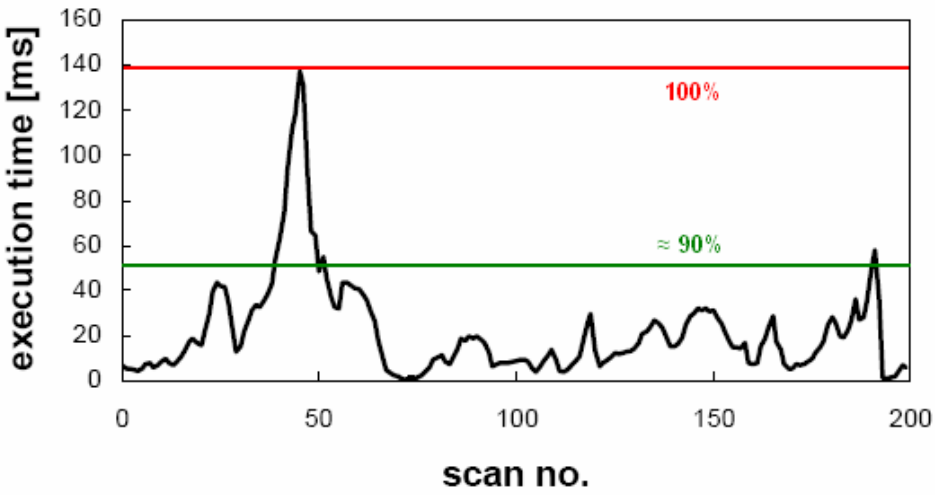

Abbildung 12: Variable Ausführungszeiten des Konturfilters [18]

Hier kommt der TAFT-Scheduler [1], der in Kapitel 2 beschrieben wurde, zum Einsatz. Eine Thread-Funktion besteht darin aus einem Mainpart (MP) und einem Exception Part (EP), wobei der Mainpart die Hauptfunktionalität enthält und periodisch immer ausgeführt wird, und der Exception Part, der nur ausgeführt wird, wenn der Mainpart in der aktuellen Periode durch eine zu lange Ausführungsdauer die Deadline nicht einhalten würden und aus diesem Grunde vom Scheduler abgebrochen wird. Auf Grund dieser Fähigkeit des TAFT-Schedulers benötigt dieser als Parameter nicht zwingen die WCET des MPs, sondern kann mit einer erwarteten Ausführungszeit (ECET) arbeiten, die unterhalb der WCET liegt, womit mehr Flexibilität und eine bessere Prozessorauslastung erreicht werden kann. Die ECET ist dabei die Mindestzeit, die einem Thread vom Scheduler garantiert zugesichert wird. Vorraussetzung dafür ist aber, dass der MP Abbrüche der Ausführung von Seiten des Schedulers tolerieren kann, zum Beispiel, dass er anytime-fähig ist. Dass heißt, er liefert ein partielles Ergebnis seiner Berechnung im Falle eines von Seiten des Schedulers erfolgten Abbruchs zurück.

Damit ist es möglich, ein Task-Konzept zu finden, dass auf den ECET Zeiten basiert, durch die Verwendung von periodischen Task-Pair Threads. Die Thread Funktion für den TAFT-Scheduler nach dem vorherigen Beispiel würde dann wie folgt aussehen:

```
Thread-Function:
{
      for(∞){
            wait_for_next_period;
      BEGIN_MAIN_PART
            call P1;
            call P2;
              :
              :
            call Pn;
      END_MAINPART

      BEGIN_EXCEPTION_PART
            call Exception;
      END_EXCEPTION_PART
      }
}
```

Die einzelnen Phasen werden innerhalb des MPs in entsprechender Reihenfolge nacheinander aufgerufen. Sollte dabei die ECET des MPs überschritten werden und ein Abbruch Seitens des Schedulers auftreten, wird der EP aufgerufen, um eventuell noch letzte notwendige Aktionen auszuführen. Der TAFT-Scheduler benötigt also als Parameter die ECET des Mainparts und die WCET des Exception Parts. Die WCET des Exception Parts ist die maximale Ausführungszeit. Wenn die WCET korrekt ist, wird die komplette Ausführung des Exception Parts garantiert, wenn eine Exception auftritt. Die Menge der Aktionen, die der EP ausführt, sollte dabei möglichst klein sein. Die Aktionen könnten zum Beispiel das Speichern von Zwischenergebnissen oder die Ausgabe der Ergebnisse sein. Die **ECET** des MPs kann durch Aufsummieren der **ECETs** aller Phasen ähnlich wie die **WCET** der vorherigen Thread-Funktion berechnet werden.

$$ECET = \sum_{i=1}^{n} ECET(P_i) \qquad (4.4)$$

Da die **ECETs** der einzelnen Phasen in der Regel kleiner als ihre **WCETs** sind, dann ist auch die gesamte **ECET** kleiner als die gesamte **WCET**. Dadurch ist die Größe der minimalen Periodendauer ebenfalls kleiner und damit kann schließlich eine höhere Sensordatenverarbeitungsrate erreicht werden. Die Größe der **WCET** des EPs sei dabei als vernachlässigbar klein **($WCET_{EP} << WCET_{MP} - ECET_{MP}$)** angenommen, damit der Einsatz des TAFT-Schedulers für das System überhaupt Nutzen bringt.

Da der Aufbau der Thread-Funktion ähnlich ist, wie in der vorher beschriebenen Lösung, basierend auf den WCET Zeiten, verhält es sich auch ähnlich mit den vier Garantien, die gefordert sind. Durch die Nacheinanderausführung der einzelnen Phasen werden die Vorgänger und Nachfolgerbedingungen eingehalten. Damit sind auch wieder die Garantien Nummer 2. und 4. gegeben. Da der Thread periodisch ist, ist auch die Garantie Nummer 1. gegeben. Jedoch die dritte Garantie kann leider nicht gegeben werden. Dies wird klar, wenn ein Szenario angenommen wird, bei dem jede Phase ihre ECET überschreitet. Der Scheduler merkt davon nichts, da er lediglich die ECET des gesamten Threads betrachtet. Er weiß jedoch nicht, dass innerhalb des MPs mehrere Phasen ausgeführt werden, mit je eigenen ECET Zeiten. Er garantiert daher nur die Verfügbarkeit der geforderten gesamten ECET, jedoch nicht die, jeder einzelnen Phase. Das bedeutet, dass die Zeit, die eine Phase über ihre ECET Zeit verbraucht, den anderen Phasen von ihren ECETs gestohlen wird. Den letzten Phasen könnten damit sogar ihre gesamte Ausführungszeit von den Vorhergehenden aufgebraucht werden. Tritt während der Berechnung nun ein Abbruch des MPs auf, dann während der Ausführung einer Phase, die nicht am Ende der Kette steht. Damit haben die Daten nicht den Berechnungsstand, der von der Benutzerapplikation gefordert ist. Nun gäbe es noch zwei Möglichkeiten damit umzugehen. Die Verarbeitung dieser Daten kann in der folgenden Periode fortsetzt werden, was aber bedeutet, dass neue Sensordaten dann nicht verarbeitet werden können und der Applikation werden Daten am Ende zur Verfügung gestellt, die zwei Perioden alt sind, oder diese Daten werden einfach verworfen und in der folgenden Periode wird erneut mit der Verarbeitung aktueller Sensordaten begonnen, was bedeutet, dass alle Berechnungen umsonst waren. In beiden Fällen kann dann aber auch die erste Garantie nicht mehr gegeben werden, da jeweils bei beiden in einer Periode der Applikation keine Daten geliefert werden konnten. Aber auch, wenn der Abbruch in der letzten Phase stattfindet, kann es Probleme geben, wenn die aktuelle Nettoausführungszeit dieser Phase weit unter ihrer ECET liegt. Obwohl alle vorangegangenen Phasen ihre Berechnungen vollständig durchgezogen haben und das bestmögliche an Informationen aus den Daten herausgeholt haben, bleibt der letzten Phase möglicherweise zu wenig Zeit, um wenigstens eine einigermaßen solide Menge an Informationen daraus zu extrahieren. Die Datenmenge, die dann der Applikation zur Verfügung steht, kann so nun trotzdem ziemlich spärlich ausfallen. Eine Kette ist nun mal nur so stark, wie ihr schwächstes Glied. Abbildung 13 zeigt diese beiden Möglichkeiten von Exceptions. Noch problematischer wird es, wenn der Abbruch währen einer Phase eintritt, die

keine anytime-Fähigkeit besitzt oder wenn elementare Aktionen noch nicht ausgeführt wurden. Zum Beispiel bei der verteilten Verarbeitung währe das die Kommunikation. Wird diese nicht ausgeführt, dann können keine Daten zwischen den Teilnehmern ausgetauscht werden und die Sensordaten entfernter Stationen stehen damit lokal nicht zur Verfügung.

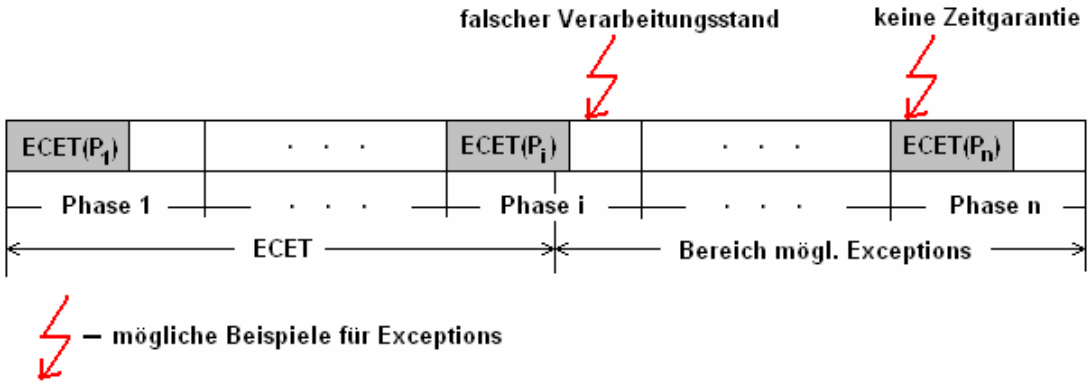

Abbildung 13: Mögliches Auftreten von Exceptions zu unterschiedlichen Zeiten

Um zu verhindern, dass eine Phase die Ausführungszeit der Nachfolgenden stiehlt, müsste zu der Berechnung der **ECETs** für die Phasen, die nicht die letzten in der Kette sind, je die **WCET** herangezogen werden. Lediglich für die letzte Phase kann die **ECET** verwendet werden, falls diese anytime fähig ist.

$$ECET = \sum_{i=1}^{n-1} WCET(P_i) + ECET(P_n) \qquad (4.5)$$

Wird auf diese Weise die **ECET** berechnet, dann ist die vollständige Ausführung der Phasen P_1 bis P_{n-1} garantiert und der Phase P_n wird mindestens deren **ECET** zugesichert. Damit ist nun auch die Garantie Nummer 3. gegeben. Jedoch ist diese Lösung der Ersten sehr ähnlich und bringt gegenüber ihr nur wenig Verbesserung, wenn nicht gerade die letzte Phase gegenüber den anderen Phasen besonders durch hohe Ausführzeiten hervorsticht und diese Ausführzeiten dazu noch stark variieren, da nur die ECET der letzten Phase Beachtung findet.

Um abzusichern, dass in jeder Periode jede Phase genau einmal ausgeführt werden kann, auch unter Heranziehung der ECETs der einzelnen Phasen als Parameter für den Scheduler, wird also eine andere Lösung benötigt. Mit Lösungen, die nur aus einem Thread bestehen, wird dieses aber schwierig. Daher bietet sich eine Lösung an, bei der mehrere Threads verwendet werden. Wird nun jeder Phase mindestens einen Thread zu geordnet, so kann jeder einzelnen Phase eine Mindestzeit zugesichert werden, die ihr pro Periode zur Verfügung steht. Es werden nun für **n** Phasen **n** Thread-Funktionen benötigt, die je aus einem MP und einem EP bestehen. Der EP besitzt dabei die Funktion, im Falle eines Abbruchs, letzte notwendige Aktionen auszuführen, wie Speichern der Zwischenergebnisse oder die Übergabe der Daten an folgende Phasen. Die Thread-Funktionen haben nun den folgenden Aufbau.

```
Thread-Function(Pi):
{
      for(∞){
            wait_for_next_period;
      BEGIN_MAIN_PART
            call Pi;
      END_MAINPART

      BEGIN_EXCEPTION_PART
            call Exception_Pi;
      END_EXCEPTION_PART
      }
}
```

Der Scheduler benötigt nun zu jedem Thread als Parameter die ECET des MPs, die WCET des EPs und die Periodendauer. Die ECET des Threads der Phase P_i ist die **ECET(P_i)**. Die WCET des EPs ist die maximale Ausführungszeit des Exception Parts. Für die Periodendauer muss ein Wert gewählt werden, der klein genug ist um eine akzeptable Verarbeitungsrate der Sensordaten zu erreichen, aber auch groß genug, damit alle beteiligten Threads ihre Zeitgrenzen einhalten können. Außerdem muss für alle Threads die gleiche Periode gewählt werden. Die untere Grenze für die Periodendauer ergibt sich damit wie folgt:

$$Period \geq \sum_{i=1}^{n} \left(ECET(P_i) + WCET(EP_i) \right) \qquad \textbf{(4.6)}$$

EP_i ist der Exception Part der Phase P_i. Wenn für die WCETs der Exception Parts wieder gilt, dass sie vernachlässigbar klein **($WCET_{EP} << WCET_{MP} - ECET_{MP}$)** sind, dann liegt diese untere Grenze auch unterhalb derer, die der Periodendauer der ersten Lösung mit nur einem Thread gesetzt ist und dazu ist in dieser Lösung nun auch noch die dritte Garantie gegeben. Jedoch wird diese Lösung jetzt auf die anderen Garantien hin untersucht, dann wird ersichtlich, dass nun die vierte nicht mehr gegeben werden kann. Dass liegt daran, dass der TAFT-Scheduler die Abhängigkeiten zwischen den Phasen nicht kennt und sie daher auch nicht beachtet.

Damit diese Abhängigkeiten dennoch beachtet werden, muss der Scheduler mittels bestimmter Parameter dazu gebracht werden. Diese Parameter sind zum einen die Release Time und zum anderen die Deadline. Die Release Time ist der absolute Zeitpunkt, zu dem ein Thread, nachdem er gestartet wurde, seine aller erste Periode startet. Die Startzeiten der folgenden Perioden, relativ von der Release Time gesehen, sind alle ganzzahlige Vielfache der Periodendauer. Mit ihr können also die einzelnen Threads zeitversetzt gestartet werden. Dadurch sind auch die Startzeiten der Phasen zeitversetzt. Die Größe dieses Versatzes bleibt in allen Perioden gleich groß, da die Periodendauer für alle Threads gleich groß ist. Wird die Release Time um einen bestimmten Betrag verschoben, so wird bei gleich bleibender Periodendauer auch das Ende der Periode verschoben. Dieses ist standardmäßig auch die Deadline des Threads. Da der TAFT-Scheduler nach dem EDF Verfahren die Ausführungsreihenfolge der MPs bestimmt, folgt daraus, dass die Phasen mit den früheren Release Zeiten auch früher zur Ausfüh-

rung kommen. Damit kann nun eine gewünschte Ausführungsreihenfolge erzielt werden, indem folgende Regel bei der Wahl der Release Times **Release** eingehalten wird.

$$\mathbf{Release}(P_i) < \mathbf{Release}(P_j) \quad \text{für} \quad i < j \qquad \textbf{(4.7)}$$

Sind die Release Times nach diesem Muster gesetzt, dann wählt der Scheduler die Phasen in gewünschter Reihenfolge. Jedoch gilt das nur, solange die Phasen bei ihren Berechnungen nicht ihre ECETs überschreiten, denn ab diesen Punkt senkt der TAFT-Scheduler die Priorität des betroffenen Threads, was bedeutet das andere Threads diesem gegenüber bevorzugt werden, ungeachtet dessen, dass deren absolute Deadline eventuell später erreicht wird. Dabei kann es sich nun aber auch um eine nachfolgende Phase handeln, die auf die Daten der aktuellen Phase angewiesen ist. Diese sind jedoch noch nicht verfügbar, da die aktuelle Phase ihre Berechnungen noch nicht abgeschlossen hat. Abgeschlossen bedeutet entweder regulär beendet oder durch einen Abbruch seitens des Schedulers und vollständige Ausführung des EPs. Damit diese Situation nicht eintritt, muss abgesichert sein, dass die Ausführung der aktuellen Phase und die eventuelle Ausführung des dazugehörigen EPs vor dem Release der folgenden Phase unbedingt beendet ist, sprich die absolute Deadline der aktuellen Phase ist gleich der Release Time der folgenden Phase. Standardmäßig ist die Deadline gleich dem Ende der Periode. Aber es besteht beim TAFT-Scheduler die Option, eine Deadline ungleich der Periode zu wählen. Die Deadline, die dem Scheduler als Parameter übergeben wird, ist immer ein Zeitwert relativ von der Release Zeit. Damit erfolgt die Wahl der Deadlines nach folgender Regel.

$$\mathbf{Deadline}(P_i) \leq \mathbf{Release}(P_{i+1}) - \mathbf{Release}(P_i) \quad \text{für} \quad i < n \qquad \textbf{(4.8)}$$

$$\mathbf{Deadline}(P_n) \leq \mathbf{Release}(P_1) + \mathbf{Period} - \mathbf{Release}(P_n) \qquad \textbf{(4.9)}$$

Dabei muss bei der Wahl der Release Times beachtet werden, dass die resultierenden Deadlines folgende Bedingung erfüllen, damit der Scheduler überhaupt einen ausführbaren Plan für die Threads erstellen kann.

$$\mathbf{Deadline}(P_i) \geq \mathbf{ECET}(P_i) + \mathbf{WCET}(EP_i) \qquad \textbf{(4.10)}$$

Dass heißt, dass die Deadline groß genug sein muss, so dass in diesem Zeitraum der Mainpart bis zur ECET und der Exception part vollständig ausgeführt werden kann. Aber sie darf auch nicht zu groß sein, da die Phasen die Periodendauer untereinander aufteilen müssen, und diese Anteile sind gerade die Deadlines der Phasen. Damit gilt.

$$\mathbf{Period} \geq \sum_{i=1}^{n} \mathbf{Deadline}(P_i) \qquad \textbf{(4.11)}$$

Um die Zeit pro Periode bestens auszunutzen, sollte möglichst sogar Gleichheit gelten. Abbildung 14 zeigt das Aufeinanderfolgen der Phasen in sich zeitlich gegenseitig ausschließenden Zeitfenstern.

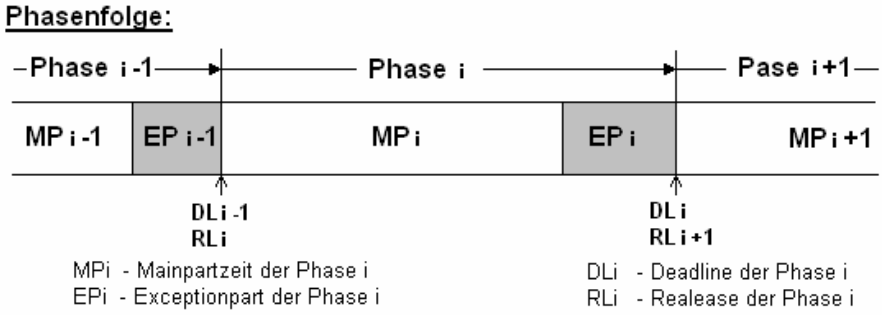

Abbildung 14: Reihenfolgeabhängige Phasenfolge

Nun bleibt noch die Frage, wie die Periodendauer konkret auf die einzelnen Phasen aufgeteilt werden soll. Ein Vorschlag wäre sie fair, abhängig von ihren Erwarteten Laufzeiten aufzuteilen. Also sollte eine Phase solch einen Anteil an der Periodendauer erhalten, der dem Anteil ihrer erwarteten Laufzeit **(ECET + $WCET_{EP}$)** von der Summe der erwarteten Laufzeiten aller Phasen entspricht.

$$\frac{\mathbf{Deadline}(P_i)}{\mathbf{Period}} = \frac{\mathbf{ECET}(P_i) + \mathbf{WCET}(EP_i)}{\sum_{j=1}^{n} \left(\mathbf{ECET}(P_j) + \mathbf{WCET}(EP_i)\right)} \qquad \textbf{(4.12)}$$

Die Release Times ergeben sich damit wie folgt.

$$\mathbf{Release}(P_i) = \mathbf{Release}(P_1) + \sum_{j=1}^{i-1} \mathbf{Deadline}(P_j) \qquad \textbf{(4.13)}$$

Die Release Time **Release(P_1)** ist dabei frei wählbar und ist somit der Startzeitpunkt der gesamten Sensordatenverarbeitung. Durch diese Weise der Aufteilung (Abbildung 15) muss nun auch nicht jede Phase einzeln überprüft werden, ob ihre Scheduler Parameter akzeptabel sind. Es muss nur einmal die Gleichung **(4.6)** überprüft werden. Ist diese erfüllt, dann finden die Gleichungen **(4.12)** und **(4.13)** auch korrekte Release Times und Deadlines.

Dieses Task-Konzept basiert nun, wie gewünscht, auf den ECETs der Phasen. Aber sind nun auch die vier gestellten Garantien gegeben? Die erste Garantie ist durch die Verwendung periodischer Threads gegeben. Der Scheduler sorgt dafür, dass sie, gemäß ihrer Periode, wiederholt ausgeführt werden. Die Periode wird zeitlich von den einzelnen Phasen in feste Zeitfenster aufgeteilt, wobei in jedem nur eine einzige Phase exklusiv ausgeführt wird. Damit muss auch keine extra Synchronisation zwischen ihnen stattfinden, beim Zugriff auf die zu verarbeitenden Daten, durch zum Beispiel Locks oder Semaphoren. Innerhalb dieser Zeitfenster wird seitens des TAFT-Schedulers abgesichert, dass die entsprechenden Phasen ausgeführt werden und ihnen mindestens die ECET als Nettolaufzeit zugesichert, und falls nötig auch noch die WCET des Exception Parts. Außerdem sind die Zeitfenster entsprechend der geforderten Ausführungsreihenfolge zeitlich sortiert. Also sind auch die Garantien Nr. 3 und Nr. 4 gegeben. Und auch

die zweite Garantie ist schließlich gegeben, da alle Zeitfenster der Phasen komplett in der Periode eingeschlossen sind. Somit sind alle vier Garantien gegeben.

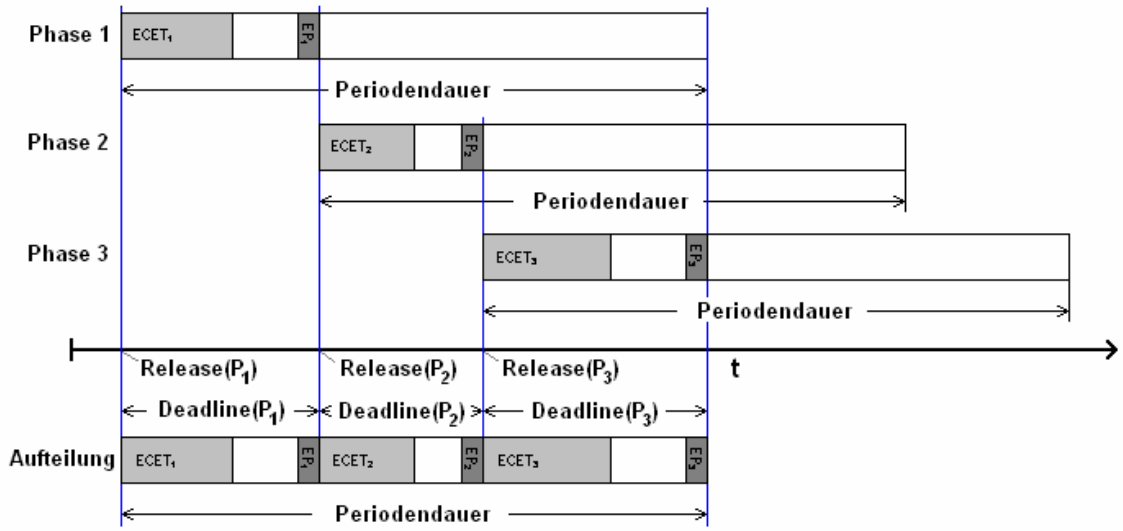

Abbildung 15: Periodenaufteilung durch mehrere TAFT-Threads

Aus der Gegebenheit der Garantien Nr. 1 und Nr. 2 folgt die **Rechtzeitigkeit**. Dass heißt, die Bearbeitung von Sensordaten wird immer innerhalb einer Periode abgeschlossen und der Nutzerapplikation stehen die gewünschten Daten zur Verfügung. Aus den Garantien Nr. 3 und Nr. 4 folgen die **Zeitgarantien für die Bearbeitungsphasen**. Es wird also abgesichert, dass alle Phasen genau einmal pro Periode ausgeführt werden und es wird ein Mindestbetrag an Ausführungszeit jeder Phase gegeben, damit diese die Möglichkeit haben alle notwendigen Aktionen auszuführen, um die Datenverarbeitung erfolgreich zu Ende zu führen. Außerdem werden Reihenfolgeabhängigkeiten zwischen den Phasen beachtet. Damit ist dieses Task-Konzept eine geeignete Lösung für die erste Teilaufgabe, wie sie im ersten Kapitel dieser Arbeit gestellt wurde.

4.2 Erweiterung des Task-Konzeptes

Im vorherigen Abschnitt dieses Kapitels wurde ein Task-Konzept vorgestellt, welches die geforderten Garantien bereitstellt und auf den erwarteten Laufzeiten der Phasen basiert, wodurch höhere Periodizität und engere Zeitgarantien erreicht werden. Das wird durch die Verwendung von mehreren Task-Pair Threads für den TAFT-Scheduler erreicht, wobei jeder Phase genau ein Thread zugeordnet wird. Damit werden je die ECETs der einzelnen Phasen beachtet, was sehr vorteilhaft ist, da jede von ihnen variable Ausführungszeiten besitzen können. Jedoch kann eine Phase selbst ebenfalls aus mehren notwendigen Unterfunktionen bestehen, die wiederum je variable Ausführungszeiten besitzen können, die aber innerhalb der Phase keine feste Ausführungsreihenfolge besitzen. Deshalb soll das Task-Konzept, welches im vorherigen Abschnitt vorgestellt wurde, in diesem Abschnitt dahingehend erweitert werden, dass nun auch die ECETs dieser Unterfunktion einzeln beachtet werden.

In der Mehrstufigen Datenverarbeitung unterteilt sich der Prozess der Datenverarbeitung in mehrere Phasen. Die Phasen besitzen eine feste Reihenfolgeabhängigkeit, die Beispielsweise dadurch bedingt sein kann, dass eine Phase von den Ergebnissen der Berechnung oder den Daten einer anderen Phase abhängig ist. Jedoch kann sich der Prozess auch in Verarbeitungsschritten unterteilen, zwischen denen keine Abhängigkeiten dieser Art existieren. Ein Beispiel lässt sich in der verteilten Fusion von Lasersensordaten finden. Eine Verarbeitungsstufe unter vielen ist dort der Konturfilter. Er benötigt als Eingabe Umgebungsdaten in Form von Punkten, wie sie Laserscanner liefern. Aus denen extrahiert er Bögen und Kanten. Diese Konturen sind die Ausgabedaten und werden wiederum von anderen Filtern oder Fusionsstufen als Eingabe benötigt. Damit kann der Konturfilter als eine Phase mit einer bestimmten ECET im gesamten Verarbeitungsprozess betrachtet werden. Doch wird der Konturfilter nun genauer betrachtet, so lassen sich zwei voneinander unabhängige Aktionen darin finden, und zwar das Extrahieren von Bögen und das Extrahieren von Kanten. Beide extrahieren ihre Informationen aus denselben Daten, nämlich den Punkten, und jeder erzeugt einen separaten Unterteil der Ergebnismenge. Sie sind damit völlig unabhängig von einander und ihre Ausführungsreihenfolge ist aus diesem Grunde egal. Beide Aktionen besitzen jedoch variable Ausführzeiten. Es wäre jedoch wünschenswert, dass beiden Aktionen mindestens je ihre ECETs als Mindestausführungszeit zugesichert werden können, ohne auf ihre WCETs als Grundlage für den Scheduler zurückgreifen zu müssen. Werden beide, in einem Thread ausgeführt, dann ergeben sich die gleichen Probleme mit den Zeitgarantien, wie sie im vorherigen Abschnitt beschrieben wurden, als mehrere Phasen innerhalb eines Threads ausgeführt wurden. Nun könnten die beiden Aktionen je als eine Phase betrachtet werden, wie im vorherigen Abschnitt beschrieben, wodurch ihnen auch je ein TP Thread zugeordnet wird. Jedoch werden dabei auch beide Aktionen in separaten Zeitfenstern ausgeführt. Dieses ist aber, wegen ihrer Unabhängigkeit von einander, überflüssig und schränkt den Scheduler in seiner Flexibilität unnötig ein. Daher wäre es besser, diese von einander unabhängige Aktionen innerhalb desselben Zeitfensters zur Ausführung zu bringen. Der Scheduler kann sie nun flexibler behandeln. Beispielsweise wenn eine Aktion auf Grund variabler Ausführungszeiten weniger Zeit benötigt als erwartet, kann der Scheduler diese Zeit anderen Aktionen im selben Zeitfenster zur Verfügung stellen. Ein Beispiel ist in Abbildung 16 zu sehen. Bei Aktionen, die auf unterschiedliche Zeitfenster verteilt sind, ist das nicht möglich.

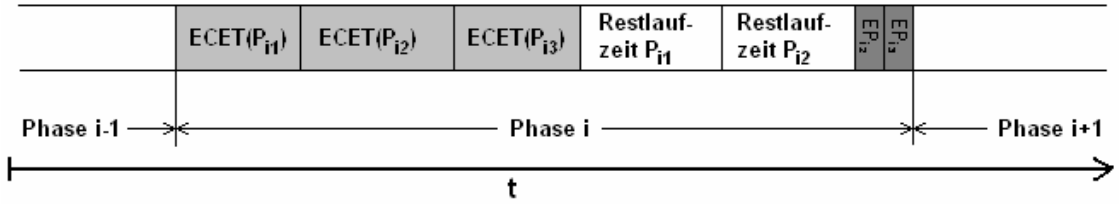

Abbildung 16: Parallele Verarbeitung innerhalb derselben Phase

Damit wird jetzt nicht mehr nur ein Thread pro Phase zugeordnet, sonder mehrere Threads. Also ein Thread pro Verarbeitungsschritt mit variableren Ausführungszeiten. Die Thread-Funktionen haben nun die folgende Form.

```
Thread-Function(Pij):
{
      for(∞){
            wait_for_next_period;
      BEGIN_MAIN_PART
            call Pij;
      END_MAINPART

      BEGIN_EXCEPTION_PART
            call Exception_Pij;
      END_EXCEPTION_PART
      }
}
```

P_{ij} ist dabei die **j**-te Unterfunktion der Phase **P_i** mit dem dazugehörigen Exception Part **EP_{ij}**.

Für die periodischen Threads wird nun wiederum eine Periodendauer benötigt, die groß genug ist, die MPs aller Threads zumindest bis zur ECET auszuführen und die EPs sogar vollständig, falls notwendig. Da nun auch einzelne Phasen aus mehren Aktionen (bis zu **m** Aktionen) bestehen, die jeweils innerhalb eines Threads ausgeführt werden, verändert sich die Gleichung **(4.6)** für die Untergrenze der Periodendauer wie folgt.

$$\mathbf{Period} \geq \sum_{i=1}^{n}\sum_{j=1}^{m}\left(\mathbf{ECET}(P_{ij}) + \mathbf{WCET}(EP_{ij})\right) \qquad \textbf{(4.14)}$$

Der Wert **m** ist hier das Maximum der Anzahl der Unterfunktionen aller Phasen. Es gilt **$ECET(P_{ij})=0$** und **$WCET(EP_{ij})=0$**, wenn **j** gleich oder größer ist, als die Anzahl der Unterfunktionen der Phase **P_i**. Um die Aktionen, die der gleichen Verarbeitungsphase angehören, in dasselbe Zeitfenster zu bringen, müssen die entsprechenden Threads die gleichen Release Zeiten und Deadlines verwenden.

$$\mathbf{Release}(P_{ij}) = \mathbf{Release}(P_i) \qquad \textbf{(4.15)}$$

$$\mathbf{Deadline}(P_{ij}) = \mathbf{Deadline}(P_i) \qquad \textbf{(4.16)}$$

Die einzelnen Deadlines müssen nun ebenfalls groß genug sein um jeweils die Zeitgarantien für alle Aktionen in einer Phase geben zu können. Die Gleichung **(4.10)** ändert sich damit auf diese Weise.

$$\mathbf{Deadline}(P_i) \geq \sum_{j=1}^{m} \mathbf{ECET}(P_{ij}) + \mathbf{WCET}(EP_{ij}) \qquad \textbf{(4.17)}$$

Die Gleichung **(4.12)** zur Berechnung der Deadlines ändert sich nun auf diese Weise.

$$\frac{\text{Deadline}(P_i)}{\text{Period}} = \frac{\sum_{k=1}^{m} \text{ECET}(P_{ik}) + \text{WCET}(EP_{ik})}{\sum_{j=1}^{n}\sum_{k=1}^{m} \left(\text{ECET}(P_{jk}) + \text{WCET}(EP_{ik})\right)} \qquad \textbf{(4.18)}$$

Die Art und Weise der Berechnung der Release Times **(4.13)** verändert sich nicht. Abbildung 17 zeigt nun eine Beispielaufteilung der Periode mit der Erweiterung.

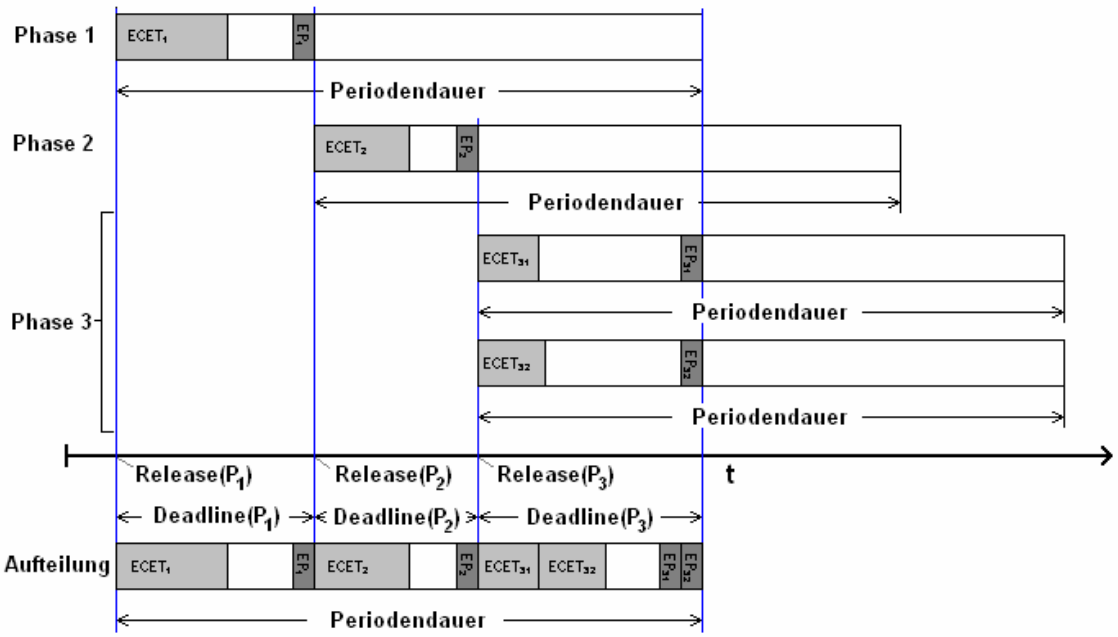

Abbildung 17: Periodenaufteilung durch mehrere TAFT-Threads (erweitert)

Mit dieser Erweiterung ist es nun auch möglich, einzelne Phasen in mehrere Einzelaufgaben zu unterteilen, indem ihnen je ein Thread zugeordnet wird. Die Folge ist, dass nun auch diesen Unterfunktionen je eine Mindestausführungszeit garantiert werden kann, die niedriger ist als die entsprechende maximale Ausführungszeit. Um auf das Beispiel des Konturfilters zurück zukommen, kann dieser nun die Extraktion von Bögen und Kanten in jeweils getrennten Threads ausführen. Damit werden ihnen in jeder Periode mindestens ihre ECETs garantiert. Dabei ist es möglich, dass der eine Thread dem jeweils anderen, bedingt durch variable Ausführungszeiten, nicht benötigte Prozessorzeit zur Verfügung stellt.

5 Zeitliche Kohärenz

In diesem Kapitel werden konzeptionell die Lösungen zur zeitlichen Kohärenz der Sensordaten vorgestellt. Im ersten Abschnitt wird die Verwendung von global gültigen Zeitstempeln eingeführt, womit am Beispiel der Elementfusion die Datenfusion mittels Kalman-Filtering Verfahrens durchgeführt werden kann, um zeitliche Diskrepanzen der Daten aufzulösen. Dieses Beispiel wird im dritten Abschnitt vorgestellt. Im zweiten Abschnitt wird das Konzept erläutert, mit einer Zeitgesteuerten Strategie die Aktivitäten der verteilten Systeme global zu koordinieren, um zeitliche Kohärenz der Daten während der Fusion zu gewährleisten.

5.1 Global gültige Zeitstempel der Sensordaten

Sensordaten werden durch die Wahrnehmung der Umgebung durch Sensoren gewonnen. Diese Wahrnehmung findet zu bestimmten Zeitpunkten statt. Damit beschreiben die Daten den Zustand zu genau diesen Zeitpunkten. Die Sensordaten müssen nun aber noch weiterverarbeitet und ausgewertet werden. Dieser Datenverarbeitungsprozess benötigt Zeit, und um genau diese Zeit altern die Daten und verlieren damit mehr und mehr an Aktualität, bedingt durch dynamische Veränderungen in der Umwelt. Für die Benutzerapplikation, die auf die verarbeiten Sensordaten angewiesen ist oder auch schon für den Verarbeitungsprozess der Sensordaten selbst könnte es daher wichtig sein, diesen Zeitpunkt zu kennen. Damit könnte die Aktualität der aus den Sensordaten gewonnen Informationen abgeschätzt werden und es könnte Beispielsweise auch eine Extrapolation der Daten durchgeführt werden mit Hilfe der Zeitinformation und möglichen zusätzlichen Kenntnissen über in der Umwelt ablaufender Prozesse, die deren Dynamik ausmachen. Das heißt, dass von dem Umweltzustand aus, der aus den Daten extrahiert wurde und ein Zustand der Vergangenheit ist, begründet durch den Zeitkonsum des Datenverarbeitungsprozesses, auf einen aktuellen oder möglicherweise auch zukünftigen Zustand geschlossen wird. Zu diesem Zweck muss zusammen mit den Daten auch die Zeitinformation festgehalten werden.

Eine Möglichkeit, die Zeitinformation festzuhalten, ist es, den Daten einen Zeitstempel beizulegen. Ein Zeitstempel repräsentiert den Zeitpunkt, zu dem der Zustand, den die Daten beschreiben, zu beobachten war, ist oder sein wird. Der Zeitstempel wird bei der Erzeugung der Daten (durch Wahrnehmung der Sensoren) erstellt, und begleitet sie während ihrer gesamten Lebensdauer. Dabei wird er in der Regel nicht mehr verändert, auch wenn die Daten verarbeitet werden und in andere Formate überführt werden. Ausgenommen bei der Extrapolation der Daten, wobei sie anschließend einen Umgebungszustand zu einem anderen Zeitpunkt repräsentieren. In diesem Fall muss der Zeitstempel entsprechend aktualisiert werden.

Ein Zeitstempel ist ein Wert, der von einem Zeitgeber geliefert wird, welcher ausreichende Genauigkeit liefert. Rechnersysteme besitzen in der Regel eine physikalische Uhr, die diese Funktion erfüllt. Mit ihr erstellte Zeitstempel können untereinander verglichen und zeitlich geordnet werden. Jedoch funktioniert das nicht mit verteilt gewonnen Daten, wobei auch deren Zeitstempel, auf unterschiedlichen Systemen erzeugt wurden. Die physikalischen Uhren sind jedoch nur lokal gültig da sie zu gleichen Zeit auf verschiedenen Systemen unterschiedliche Zeitwerte besitzen. Dazu existieren noch geringe Unterschiede zwischen den Taktraten der Uhren, wodurch ihre Zeitwerte ständig auseinanderdriften. Daher wird eine global gültige Uhr benötigt, mit deren Hilfe alle Zeitstempel kreiert werden.

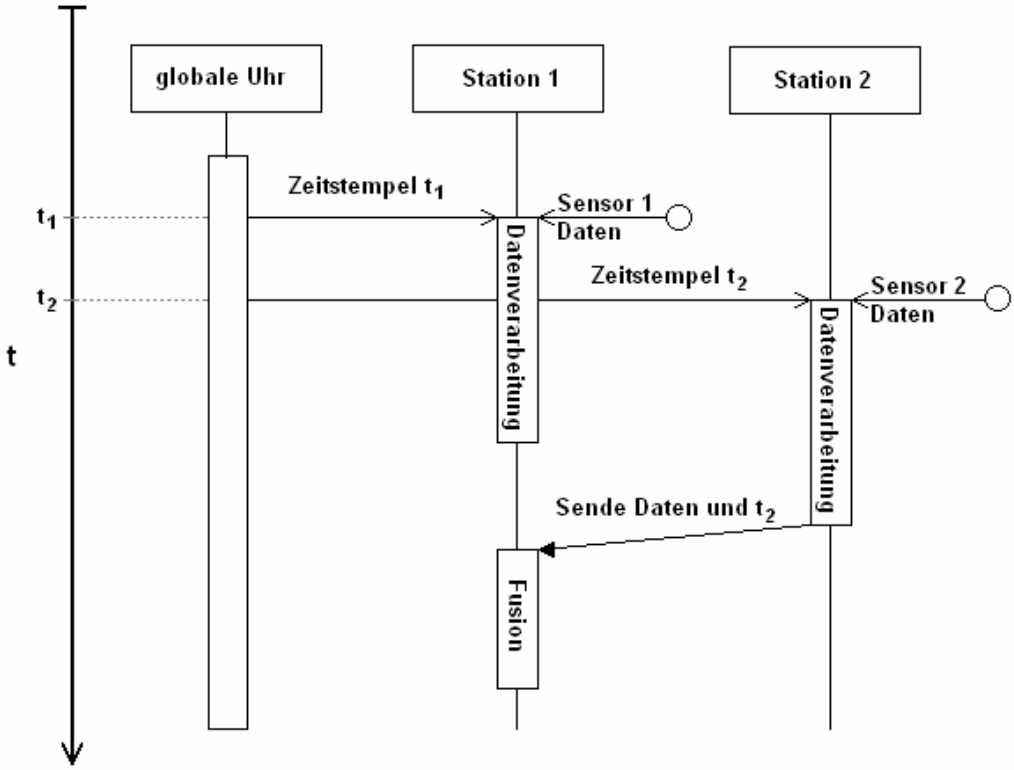

Abbildung 18: Verwendung der globalen Uhr

Um eine gültige globale Uhr zu erhalten, ist eine Synchronisation dieser Uhren zwischen allen beteiligten Stationen notwendig. Eine solche führt die WLAN Uhrensynchronisation [7] aus, wie sie in Kapitel 2. erwähnt wurde. Diese Uhrensynchronisation wird für diese Arbeit verwendet, da sie auf einem drahtlosen Kommunikationsmedium basiert, wie es autonome mobile Robotersysteme verwenden können. Die einzelnen Zeitstempel werden nun ausschließlich mit Hilfe dieser Uhr erstellt und den Daten zugeordnet. Beim Austausch der Daten zwecks Fusion, muss dabei beachtet werden, dass auch die Zeitstempel mit ihnen übertragen werden. Diese können nun verwendet werden, um die zeitliche Kohärenz der Daten zu überprüfen (Abbildung 18). Das ist wichtig, wenn die Daten auf den Wahrnehmungen von Sensoren zu möglicherweise unterschiedlichen Zeiten basieren und die beobachtete Umwelt dynamischer Natur ist. Damit können zeitliche Diskrepanzen der Daten unter anderem zu widersprüchlichen Umwelt-

informationen führen. Als zeitlich Kohärent gelten die Daten, wenn der Zeitunterschied zwischen ihren Zeitstempeln unter einem bestimmten Schwellenwert, liegt. Der Schwellenwert **SW** ist dabei die maximale Toleranz der Datenfusionsalgorithmen der Zeitdiskrepanzen zwischen den Daten. Dabei muss aber auch bedacht werden, dass die Größe dieses Schwellenwertes nach unten begrenzt ist, begründet durch einen geringen unvermeidbaren Zeitfehler in den Zeitstempeln. Dieser ist die Summe aus den folgenden zwei Fehlern.

1. ***Fehler* (ERROR$_{ts}$) *in den Zeitstempeln selbst.*** Dieser Fehler basiert auf Ungenauigkeiten der Uhrensynchronisation. Diese besitzt aus technischen Gründen nur eine begrenzte Präzision. Beispielsweise durch die ständige Driftrate, mit der die Uhren auf den einzelnen Stationen auseinander laufen. Die Korrektur dieses Fehlers ist nur in bestimmten Zeitabständen möglich. Daher entsteht ein Fehler, der beachtet werden sollte. Aber dieser Wert ist durch einen maximalen Wert begrenzt.

2. ***Fehler* (ERROR$_{cr}$)*beim Erzeugen der Zeitstempel.*** Dieser Fehler entsteht durch den Prozess der Erzeugung der Sensordaten bzw. der Wahrnehmung der Umwelt. Dieser benötigt ebenfalls eine gewisse Zeit, und läuft in den Sensoren intern ab. Diese Zeit müsste eigentlich beachtet werden, bei der Erzeugung des Zeitstempels. Doch oft sind Informationen über diese Zeit nicht verfügbar. Die maximale Größe dieser Zeit wird als Antwortzeit der Sensoren bezeichnet. Diese geht dann als möglicher Fehler mit in den Zeitstempel mit ein. Ein Laserscanner LMS 200 [24] zum Beispiel besitzt einen Rotierenden Spiegel, um den Laserstrahl abzuleiten, und somit die Umgebung in einem bestimmten Winkelbereich abzuscannen. Trifft nun eine Anforderung an den Scanner ein, Daten zu erzeugen, dann kann es je nach aktueller Position des Spiegels eine gewisse Zeit dauern, bis der Spiegel den Anfangswinkel des Scannbereichs erreicht, womit erst dann die eigentliche Aufnahme beginnt. Abbildung 19 zeigt die Funktionsweise des LMS 200 und wodurch die Daten nach Anforderung zeitlich verzögert zurückgeliefert werden.

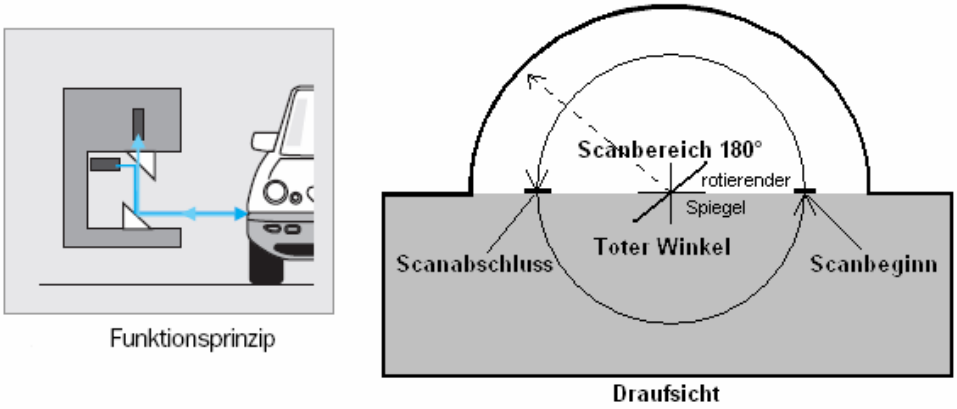

Abbildung 19: Antwortzeit eines Laserscanners

Die Summe der maximalen Werte dieser beiden Fehler ist der maximal zu erwartende Gesamtfehler bei der Verwendung der Zeitstempel, und sollte als Untergrenze für den Schwellenwert **SW** dienen.

$$SW \geq MAX(ERROR_{ts}) + MAX(ERROR_{cr}) \qquad \textbf{(5.1)}$$

Somit kann die globale Uhr zur Erzeugung von Zeitstempeln mit einem gewissen Grad an Genauigkeit herangezogen werden. Diese Zeitstempel können verwendet werden, um die Daten von unterschiedlichen Sensoren auf **zeitliche Kohärenz** zu untersuchen, und gegebenenfalls eine Extrapolation der Daten durchzuführen, um Kohärenz zwischen ihnen zu erreichen.

5.2 Globale Zeitgesteuerte Koordinierung der Verarbeitungsphasen

Wenn die Verarbeitung der Sensordaten auf verteilten Systemen abläuft, kann es notwendig sein, bestimmte Aktionen zeitlich zu koordinieren. Bei der Erzeugung neuer Sensordaten durch die Wahrnehmung der Sensoren können, wie in vorangegangenen Abschnitten dieses Kapitels bereits erwähnt, zeitliche Diskrepanzen zwischen den Daten verschiedener Sensoren entstehen, wenn diese auf je unterschiedlichen Systemen existieren. Im vorherigen Abschnitt wurde bereits die Möglichkeit vorgestellt, die Daten mit auf einer global gültigen Uhr basierenden Zeitstempeln zu versehen. Diese können herangezogen werden, um die Existenz solcher Diskrepanzen festzustellen und die Größe der Abweichung abzuschätzen. Tritt der Fall nun ein, dass zeitlich nicht kohärente Daten vorliegen, so dass eine Fusion der gleichen nicht ohne weiteres möglich wäre, so müssen die Daten der Art behandelt werden, dass sie schließlich dennoch die Fusion stattfinden kann. Es wurde bereits die Möglichkeit der Datenextrapolation erwähnt, um zeitliche Diskrepanzen aufzulösen. Abbildung 20 zeigt Beispielhaft das Problem der Mehrdeutigkeit, bei der Interpretation von fusionierten Daten, basierend auf der Fusion zeitlich inkohärenter Sensordaten.

Abbildung 20: Beispiel: Probleme bei der Fusion zeitlich inkohärenter Daten

Doch nicht immer existieren solche Verfahren, die das mit ausreichender Qualität ausführen. Dann müsste durchgesetzt werden, dass nur zeitlich kohärente Daten fusioniert werden, indem bestimmte Daten davon ausgeschlossen werden, was aber wiederum ein Informationsverlust bedeutet. Außerdem ist das Auftreten von zeitlichen Diskrepanzen kein gleich verteiltes Zufallsereignis, wenn die Sensordatenverarbeitung auf allen Teilnehmern mit der gleichen Periodendauer durchgeführt wird. Denn so ist zu erwarten, dass Stationen, die in der aktuellen Periode zueinander zeitlich inkohärente Sensordaten liefern, dies auch in den zukünftigen Perioden tun. Zwar verhält es sich analog dazu auch so mit zeitlich kohärenten Daten, denn die sich periodisch wiederholende Erzeugungszeitpunkte der Daten auf den verteilten Systemen haben zu einander feste Zeitunterschiede, bedingt durch die gleiche Periodendauer, aber die periodische Ausführung der Verarbeitung, einschließlich der Erzeugung der Sensordaten wird auf den einzelnen Systemen durch den lokalen Scheduler organisiert, der wiederum zu diesem Zweck auf die lokale Systemuhr zurückgreift. Dabei muss beachtet werden, dass die Zeitwerte der Uhren aus technischen Gründen ständig mit einer festen Rate auseinanderdriften. Das hat zur Folge, dass trotz vorgegebener fester Periodendauer für alle Teilnehmer, die realen Periodendauern von einander geringfügig abweichen, womit sich über die Zeit auch die Zeitunterschiede zwischen den Erzeugungszeitpunkten linear ändern. Somit ist irgendwann bei einer Folge von mehren Perioden, in denen zwei Teilnehmer kohärente Daten lieferten, der Zeitpunkt erreicht, ab dem in allen folgenden Perioden nun zueinander zeitlich inkohärente Daten erzeugt werden, bis zu einem weiteren Zeitpunkt, in dem das Spiel von neuem beginnt. Damit entstehen im Wechsel große Zeiträume der zeitlichen Kohärenz und Inkohärenz, was ein Problem sein kann. In Abbildung 21 ist das Auftreten dieser Zeiträume über eine große Zeitspanne skizziert. Soll nun aber immerzu zeitliche Kohärenz der Daten gewährleistet sein, so müssten die Erzeugungen der Sensordaten, also die Sensoraktivitäten, auf den verteilten Systemen miteinander koordiniert werden.

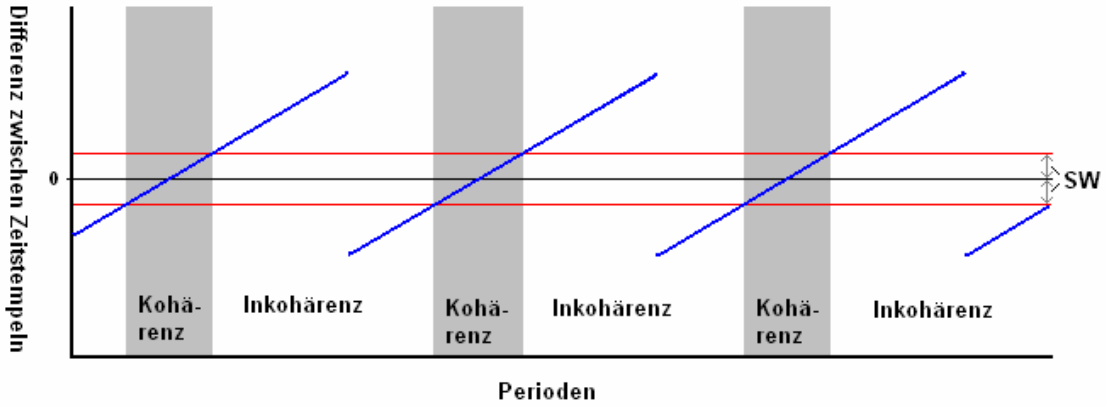

Abbildung 21: Wechsel von zeitlichen Kohärenzen und Inkohärenzen

Aber nicht nur die Erzeugung der Sensordaten ist während der verteilten Sensordatenverarbeitung eine Problemstelle. Im vierten Kapitel wurden die Reihenfolgeabhängigkeiten zwischen den einzelnen Phasen erwähnt und ein Task-Konzept entworfen, das diese Abhängigkeiten beachtet. Jedoch werden dort diese nur lokal beachtet. Sollen aber die Sensordaten von verteilten Systemen fusioniert werden, so existieren nun auch Reihenfolgeabhängigkeiten zwischen den Phasen über die Systemgrenzen hinaus. Die Fusionsphase erhält ihre Daten von der Kommunikationsphase, die die Daten von allen beteiligten Systemen während deren Kommunikationsphasen empfängt. Im Gegenzug sendet sie die lokalen Daten und stellt sie damit den jeweils anderen Teilnehmern zur Verfügung. In der Kommunikationsphase müssen dann alle zueinander zeitlich kohärenten Daten rechtzeitig empfangen werden, bevor die Phase der Fusion beginnt, die darauf angewiesen ist (Abbildung 22). Damit ist die Kommunikation ebenfalls eine zu koordinierende Aktion zwischen allen teilnehmenden Systemen. Zusammengefasst existieren also zwei Gründe, die Phasen der verteilten Verarbeitung zu koordinieren. Zum Ersten, um zeitlich kohärente Sensordaten zu erzeugen und zum Zweiten, um globale Reihenfolgeabhängigkeiten zwischen den Verarbeitungsphasen zu garantieren.

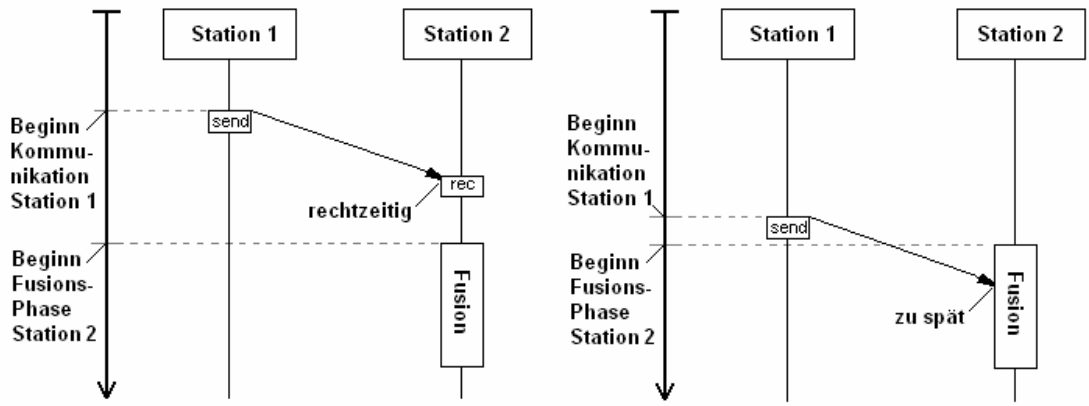

Abbildung 22: Globale Reihenfolgeabhängigkeit der Fusion und Kommunikation

In Kapitel 3 wurden bereits zwei Strategien zur Koordinierung vorgestellt. Die Ereignisgesteuerte (event triggered) Aktivierung und die Zeitgesteuerte (time triggered) Ak-

tivierung. Für die Ereignisgesteuerte Aktivierung wird ein Ereigniserzeuger benötigt. Damit könnte dann die Ausführung bestimmter Verarbeitungsphasen, wie die Kommunikation und die Erzeugung der Sensordaten gesteuert werden. Die Datenverarbeitung läuft jedoch unter harten Echtzeitanforderungen. Das bedeutet, dass die Phase nach ihrer Aktivierung durch ein Ereignis bis zu einer Deadline ihre Aufgaben erfüllten haben muss. Da die Aktivierung jedoch asynchron und zu einem gewissen Grad unvorhersehbar erfolgt, kann auf dem betroffenen System keine sichere Garantie über die Verfügbarkeit erforderlicher Ressourcen (Bsp.: CPU, Daten) gegeben werden, die nötig sind, um die Ausführung zu beginnen. Aus diesem Grund wird eine Zeitgesteuerte Koordinierung bevorzugt, da sie sich in der Regel durch bessere Vorhersagbarkeit auszeichnet und damit für ein Echtzeitsystem geeigneter ist.

Das im vierten Kapitel vorgestellte Task-Konzept garantiert die periodische Ausführung der einzelnen Phasen in einer festen Reihenfolge innerhalb festgelegter Zeitfenster. Damit enthält dieses Konzept bereits eine lokale Zeitgesteuerte Koordinierung. Lokal bedeutet hier, dass nur die Phasen auf demselben System miteinander mit Hilfe der lokalen Systemuhren zeitlich koordiniert sind. Es wird jedoch nach einer globalen Koordinierung der Phasen verlangt. Zu diesem Zweck benötigen alle teilnehmenden Systeme eine globale Uhr als Basis dafür.

Im vorhergehenden Abschnitt wurde bereits eine globale Uhr verwendet, um die Sensordaten mit Zeitstempeln zu versehen und zeitlich zu ordnen. Genau diese Uhr kann nun auch für die globale Koordinierung der verteilten Sensordatenverarbeitung und Fusion verwendet werden. Jede Verarbeitungsperiode wird dann auf den einzelnen Systemen zu einer festen bekannten Zeit gestartet. Das heißt, dass es eine vereinbarte Zeit gibt, die alle Teilnehmer ermitteln können. Hier wurden als gültige Startzeiten alle ganzzahligen Vielfachen der Periodendauer festgelegt. Wird zu einem Zeitpunkt $\mathbf{GT_{now}}$ die Aufforderung gegeben, mit der periodischen Sensordatenverarbeitung zu beginnen, dann berechnen alle Teilnehmer nach folgender Gleichung die nächst mögliche gültige Startzeit $\mathbf{GT_{start}}$ der ersten Periode.

$$GT_{start} = GT_{now} + Period - GT_{now} \text{ modulo } Period \qquad (5.2)$$

$\mathbf{GT_{start}}$ und $\mathbf{GT_{now}}$ beziehen sich dabei beide auf Zeitwerte der globalen Uhr. Die Aufforderung zum Beginn der Verarbeitung muss dabei nicht unbedingt auf allen Systemen gleichzeitig erfolgen. Es ist nicht zwingend notwendig, dass alle Teilnehmer die gleiche $\mathbf{GT_{start}}$ ermitteln. Es muss nur eine Zeit sein, die ein ganzzahliges Vielfaches der Periodendauer **Period** ist, was durch die Gleichung **(5.2)** sicher ist. Sobald alle beteiligten Systeme ihre Verarbeitung gestartet haben, werden ihre Perioden zueinander phasensynchron gestartet. Da alle Phasen feste Startzeiten relativ zum Beginn der aktuellen Periode besitzen, können diese nun durch die geeignete Wahl der relativen Startzeiten und Deadlines auch über die Systemgrenzen hinweg miteinander koordiniert werden. Phasen, die im selben Zeitfenster aktiv sein sollen, bekommen daher die gleiche relative Startzeit und die gleiche Deadline.

Die Ausführung der periodischen Threads, die den Phasen zugeordnet sind, werden vom Scheduler organisiert. In dieser Arbeit ist das der TAFT-Scheduler. Dieser verwendet jedoch dazu die lokale Systemuhr, womit er mit dem Zeitwert **GT**$_{start}$ als Startzeit nichts anfangen kann. Aus diesem Grund muss für den Scheduler die zu **GT**$_{start}$ entsprechende Zeit **LT**$_{start}$ der lokalen Uhr ermittelt werden. Dies erfolgt mit der folgenden Gleichung.

$$LT_{start} = LT_{now} + Period - GT_{now} \text{ modulo } Period \qquad (5.3)$$

LT$_{now}$ ist dabei der zu **GT**$_{now}$ entsprechende Zeitwert der lokalen Uhr. Die Ratenunterschiede der Uhren werden hier ignoriert, da der resultierende Fehler, abhängig von **Period** und der maximalen Driftrate der Uhren, begrenzt ist und als vernachlässigbar klein angenommen wird. Das Wertepaar **GT**$_{now}$ und **LT**$_{now}$ kann dadurch erhalten werden, indem die Werte der globalen und der lokalen Uhr möglichst zeitgleich ausgelesen werden. Der Scheduler startet dann die erste Periode der Verarbeitung zum Zeitpunkt **LT**$_{start}$ nach lokaler Uhrzeit, was dem Zeitpunkt **GT**$_{start}$ nach globaler Uhrzeit entspricht. Es existiert zwar eine geringe Ungenauigkeit, die jedoch durch die verwendete WLAN Uhrensynchronisation begrenzt ist und deshalb hier vernachlässigt wird.

Was aber nicht vernachlässigt werden kann, ist das ständige Abdriften der lokalen Systemuhren von der globalen Uhr mit unterschiedlichem Ausmaß. Der Scheduler verwendet die lokale Uhr, um die Länge der Perioden zu messen. Durch diese technisch bedingten Unterschiede zwischen den lokalen Systemuhren hat das zur Folge, dass die Periodenlängen auf allen beteiligten Systemen geringfügig unterschiedlich lang sind. Das wiederum führt dazu, dass die Periodenzeiten mit der Zeit immer weiter auseinander driften. Daher müssen die lokalen Startzeiten aller Phasen je System in regelmäßigen Abständen korrigiert werden. Da die Datenverarbeitung periodisch erfolgt, wird deshalb die Korrektur ebenfalls einmal je Periode ausgeführt. Abbildung 23 zeigt die Koordinierung zwischen zwei Stationen.

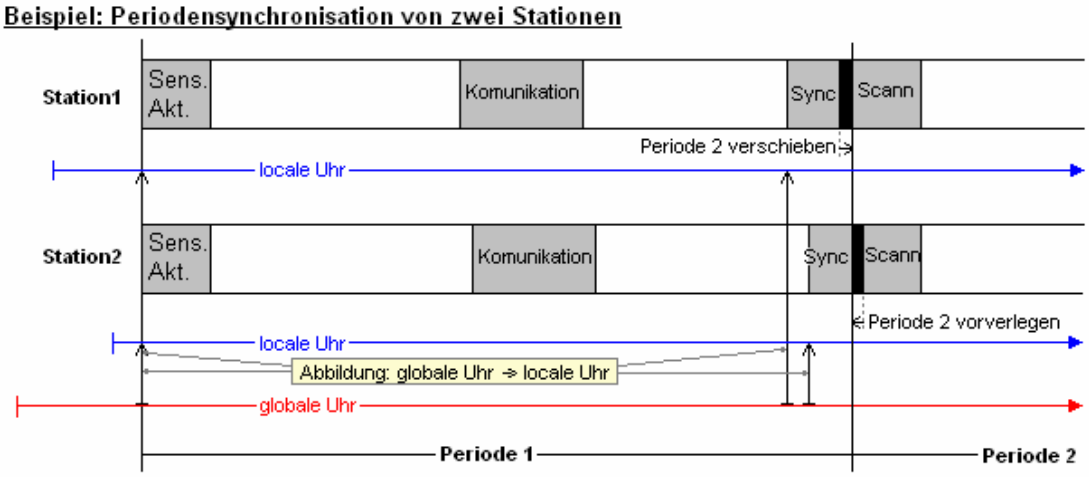

Abbildung 23: Koordinierung der Periodenstartzeiten

Nun muss noch der Wert ermittelt werden, um den die lokale Zeit korrigiert werden muss. Gegeben seien das Wertepaar $\mathbf{g}_1$ und $\mathbf{l}_1$, wobei $\mathbf{g}_1$ der Wert der globalen Uhr zu einem bestimmten Zeitpunkt ist und $\mathbf{l}_1$ der Wert der lokalen Uhr zur selben Zeit. Sei nun angenommen, dass die lokalen Uhren fehlerlos arbeiten. Das heißt, dass ein zweites Wertepaar $\bar{\mathbf{g}}_2$ und $\bar{\mathbf{l}}_2$ wie folgt vorhersagt werden kann, wie es nach einem Zeitraum $\Delta\mathbf{t}$ nach dem Moment, an dem das erste Wertepaar $\mathbf{g}_1$ und $\mathbf{l}_1$ gelesen wurde, von den beiden Uhren erhalten werden kann.

$$\bar{\mathbf{g}}_2 = \mathbf{g}_1 + \Delta\mathbf{t} \quad \textbf{und} \quad \bar{\mathbf{l}}_2 = \mathbf{l}_1 + \Delta\mathbf{t} \qquad \textbf{(5.4)}$$

Da die lokalen Uhren jedoch aus technischen Gründen nicht fehlerlos arbeiten, gilt für das Wertepaar $\mathbf{g}_2$ und $\mathbf{l}_2$, wenn $\mathbf{g}_2 = \bar{\mathbf{g}}_2$, dann ist in der Regel $\mathbf{l}_2 \neq \bar{\mathbf{l}}_2$. Auf diese Weise ist es also möglich, aus zwei solchen Wertepaaren den Korrekturwert zu berechnen. Während der periodischen Startzeitkorrektur wird ein aktuelles Wertepaar $\mathbf{g}_i$ und $\mathbf{l}_i$ ermittelt, und als zweites wird das, aus der vorherigen Periode ermittelte Wertepaar $\mathbf{g}_{i-1}$ und $\mathbf{l}_{i-1}$ verwendet. Die periodische Berechnung des Korrekturwertes $\mathbf{C}_i$ wird mit folgender Gleichung durchgeführt.

$$\mathbf{C}_i = (\mathbf{l}_i - \mathbf{l}_{i-1}) - (\mathbf{g}_i - \mathbf{g}_{i-1}) \qquad \textbf{(5.5)}$$

Die Berechnung ist noch mal graphisch in Abbildung 24 dargestellt. Die Startzeit der nächsten Periode muss nun um diesen Wert verschoben werden. Das wird dadurch erreicht, indem im Scheduler auf die Aktivierungszeiten der nächsten Periode aller periodischen Threads dieser Korrekturwert addiert wird. Dabei sind nicht nur die gemeint, die an der Sensordatenverarbeitung beteiligt sind, sonder ausnahmslos alle periodischen Threads, denn sonst besteht die Gefahr, dass für die resultierende Konstellation der Threads kein realisierbarer Ausführungsplan für den Scheduler auffindbar ist. Werden jedoch alle gleich behandelt, ändert sich die Konstellation nicht und auf eine erneute Überprüfung mittels Akzeptanztest des Schedulers kann verzichtet werden.

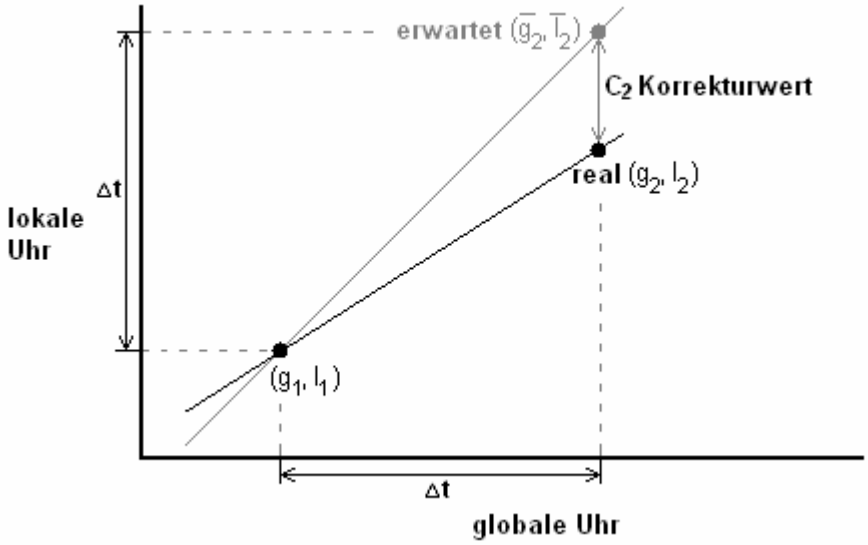

Abbildung 24: Ermitteln des Korrekturwertes

Der Korrekturwert kann positiv oder negativ sein. Ein positiver Wert bedeutet, dass die Periodendauer in Bezug zur globalen Uhr zu kurz ist und die nächste Periode beginnt folglich zu früh. Die Startzeit wird daher um den Korrekturwert zeitlich nach hinten verschoben. Das ist unkritisch, da der CPU keine Zeit verloren geht. Ein negativer Wert dagegen bedeutet, dass die Periodendauer zu lang ist und die nächste Periode beginnt daher zu spät. Die Startzeit wird damit vorverlegt, was kritisch sein kann, da die CPU nun Zeit verliert, die eventuell für die Ausführung anderer Threads verplant wurden. Dies sollte unbedingt vermieden werden, da eventuell zeitliche Garantien seitens des Schedulers nicht mehr gegeben werden können. Verhindern lässt sich das, indem die Zeit für mögliche Startzeitverschiebungen nach vorn pro Periode reserviert wird. Wie groß diese reservierte Zeit sein muss hängt davon ab, wie groß die maximale Abweichung der Startzeiten pro Periode sein kann. Die Startzeitkorrektur wird von einem periodischen Thread ausgeführt, der daher auch eine Maximale Ausführungszeit WCET besitzt, die ihm vom Scheduler garantiert wird. Wird nun zu dieser WCET noch die zu reservierende Zeit dazuaddiert, so wird ihm auch diese garantiert. Die resultierende WCET wird auch als Deadline für diesen Thread verwendet, um zu verhindern, dass er durch andere Threads inmitten seiner Ausführung verdrängt werden kann. Tritt nun der Fall ein, dass die Startzeit der Periode nach vorn verlegt wird, dann beraubt sich dieser Thread gerade um diese Zeit. Andere Threads werden damit nicht beeinflusst. Der Korrektur-Thread selbst erleidet jedoch ebenfalls keinen Schaden, da er diese Zeit ohnehin nicht benötigt. Auf diese Weise reserviert er selbst mindestens die Zeit, um die maximal eine Verschiebung der Periodenstartzeit nach vorn stattfinden kann.

Da nun pro Periode der Korrektur-Thread ausgeführt werden soll, muss dieser auch ein freies Zeitfenster besitzen und muss bei der Berechnung der Release Zeiten und Deadlines der Datenverarbeitungsphasen mit einbezogen werden. Die Gleichung **(4.14)** für die Untergrenze der Periodendauer **Period** ändert sich damit wie folgt.

$$\mathbf{Period} \geq \sum_{i=1}^{n}\sum_{j=1}^{m}\left(\mathbf{ECET}(P_{ij}) + \mathbf{WCET}(EP_{ij})\right) + \mathbf{Deadline}_C \qquad \textbf{(5.6)}$$

$\mathbf{Deadline}_C$ ist dabei die Deadline des Korrektur-Threads, die sich aus der Summe der maximalen Laufzeit der Korrekturaktion und der maximalen Zeitverschiebung ergibt. Die Gleichung **(4.18)** zur Berechnung der Deadlines der Phasen wird auf diese Weise angepasst.

$$\frac{\mathbf{Deadline}(P_i)}{\mathbf{Period} - \mathbf{Deadline}_C} = \frac{\sum_{k=1}^{m}\mathbf{ECET}(P_{ik}) + \mathbf{WCET}(EP_{ik})}{\sum_{j=1}^{n}\sum_{k=1}^{m}\left(\mathbf{ECET}(P_{jk}) + \mathbf{WCET}(EP_{ik})\right)} \qquad \textbf{(5.7)}$$

Die Startzeitkorrektur wird am Ende jeder Periode durchgeführt. Damit wird die Release Zeit des Korrektur-Threads wie folgt ermittelt.

$$\mathbf{Release}_C = \mathbf{Release}(P_1) + \mathbf{Period} - \mathbf{Deadline}_C \qquad \textbf{(5.8)}$$

Zusammenfassend wurde damit in diesem Abschnitt eine Möglichkeit entwickelt, Phasen, auf einer Zeitgesteuerten Strategie basierend, global zu koordinieren. Es können Aktionen koordiniert werden, wie die Erzeugung der Sensordaten mittels Wahrnehmung durch die Sensoren, womit zeitliche Kohärenz zwischen den Daten erreicht werden kann. Es können ebenfalls nun auch Datenabhängigkeiten zwischen den Verarbeitungsphasen über die Systemgrenzen hinaus beachtet werden. Solche Abhängigkeiten existieren zum Beispiel in den Fusionsstufen. Diese benötigen die Sensordaten von verschiedenen verteilten Systemen, die daher rechtzeitig vor Beginn der Fusion lokal bereitliegen sollten. Diese **Rechtzeitigkeit** kann erreicht werden, wenn die Kommunikationsphasen der verteilten Systeme miteinander koordiniert werden.

5.3 Zeitliche Kohärenz mittels Kalman-Filtering Verfahren

Um Sensordaten zu fusionieren, die verteilt aufgenommen wurden, sollte gewährleistet sein, dass diese konsistente Umgebungsinformationen liefern. Eine mögliche Quelle von Inkonsistenzen kann die Wahrnehmung einer dynamischen Umwelt zu unterschiedlichen Zeitpunkten sein. Im ersten Abschnitt dieses Kapitels wurde die Verwendung von Zeitstempeln für die Sensordaten erklärt, die auf einer globalgültigen Uhr basieren. Mit deren Hilfe können die Daten auf zeitliche Kohärenz überprüft werden. Jedoch ist dies allein keine Lösung, um zeitliche Kohärenz der Daten, die zur Fusion herangezogen werden sollen, zu gewährleisten. Im zweiten Abschnitt wurde daher eine Zeitgesteuerte globale Koordinierung als Lösung vorgestellt. Allerdings wurde im ersten Abschnitt auch die Möglichkeit erwähnt, zeitlich inkohärente Sensordaten mittels geeignetem Extrapolationsverfahren durch Auflösung der Zeitdiskrepanzen zu fusionieren, womit es optional auch möglich ist, auf die globale Koordinierung der Systemaktivitä-

ten zu verzichten. Diese soll nun in diesem Abschnitt mittels des Kalman-Filtering Verfahrens [19] am Beispiel der mehrstufigen verteilten Sensorfusion von Laserscandaten gezeigt werden.

Die mehrstufige verteilte Sensorfusion von Laserscandaten [15] wurde entwickelt, damit im Team agierende Roboter beim Robocup ihre Sensordaten austauchen und fusionieren können, um ein vollständigeres und einheitliches Weltbild zu erhalten. Die rohen Daten vom Laserscanner werden stufenweise in andere Datenformate überführt, wobei jedes Mal relevante Merkmale extrahiert werden. Dabei sind besonders die höherstufigen Filter und Fusionsstufen, wie zum Beispiel der Elementfilter und die Elementfusion, speziell auf das Robocup Szenario ausgerichtet. Im Robocup Szenario ist eine dynamische Umwelt gegeben, bedingt durch sich bewegende Objekte bzw. Elemente, wie der Ball oder die Roboter. Dadurch kann es zu mehrdeutigen Interpretationsmöglichkeiten über die, aus den fusionierten Daten gewonnen Elementpositionen, kommen, wenn zeitlich inkohärente Sensordaten fusioniert werden. Daher soll hier die Elementfusion mit Hilfe des Kalman-Filters durchgeführt werden.

Der Kalman-Filter ist ein rekursiver Filter, bei dem in jedem Zyklus die Daten der Vergangenheit mit in den Prozess einfließen. Ein Filterzyklus besteht aus einem Prediction-Schritt, während dem, basierend auf einen Zustand der Vergangenheit, ein aktueller Zustand vorhergesagt wird, und einem Update-Schritt, womit, mit Hilfe des vorhergesagten Zustandes und eines aktuell gemessenen Zustand, der aktuelle Zustand geschätzt wird. Hierbei ist gerade der Prediction-Schritt besonders interessant für die Fusion zeitlich inkohärenter Daten, da er Diskrepanzen dieser Art auflösen kann, vorausgesetzt es liegt ihm ein angemessenes Prozessmodel zu Grunde. In der Anwendung für die Fusion sieht das nun so aus, dass die Daten von unterschiedlichen Systemen nach den Zeitstempeln zeitlich sortiert werden und in der Reihenfolge dann in eine Folge von abwechselnden Prediction-Schritten und Update-Schritten fusioniert werden (Abbildung 25). Am Ende sind nicht nur die Sensordaten der aktuellen Periode fusioniert, sonder es fließen sogar die Daten der vergangenen Perioden mit ein.

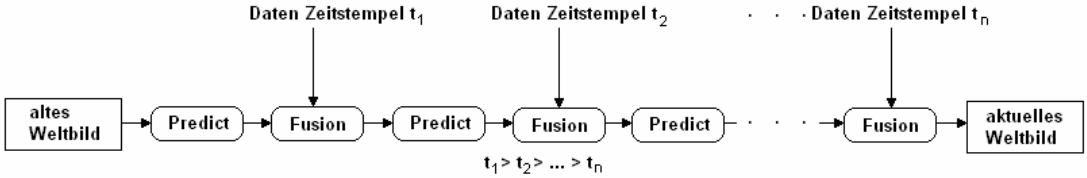

Abbildung 25: Fusion von Sensordaten sortiert nach den Zeitstempeln

Die Stufe der Elementfusion verarbeitet die Daten des Elementfilters. Dieser extrahiert aus seinen Eingabedaten Elemente. Ein Element hat eine bestimmte Position, die in Form kartesischer zweidimensionaler Koordinaten festgehalten wird. Dazu hat es noch eine Größe und ist von einem bestimmten Typ, entweder Ball, Roboter, oder Bande. Während Typ und Größe der Elemente als zeitlich unveränderliche Zustände angenommen werden können, ist die Position die Komponente, auf die sich die Dynamik der Umwelt auswirkt. Das heißt, es treten zeitliche Veränderungen auf, da sich die E-

lemente in der Umgebung bewegen. Daher ist die Position auch die Komponente, auf die das Kalman-Filtering Verfahren in der Fusion hier angewendet werden soll.

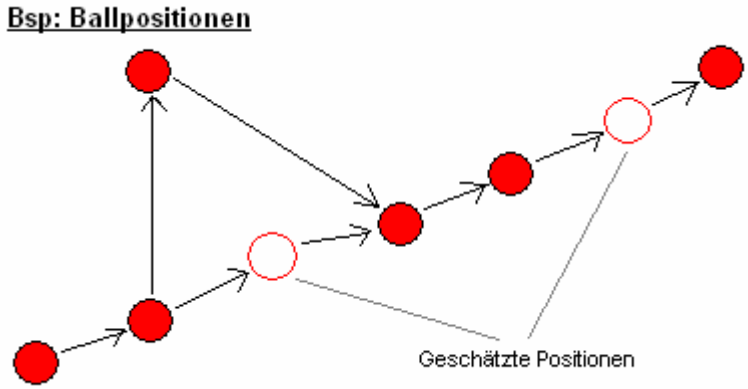

Abbildung 26: Beispiel Ballposition: Herausfiltern von Positionsfehlern

Außer der Möglichkeit, zeitlich inkohärente Daten zu fusionieren, liefert der Kalman-Filter zusätzlich noch einen Bonus (Abbildung 26). Da er ursprünglich entwickelt wurde, um Messfehler herauszufiltern, kann er dies nun auch innerhalb der Elementfusion tun. Messfehler können während der Positionsbestimmung der Elemente entstehen, auf Grund des Einflusses gaußschem Rauschens auf die Sensoren, Informationsverlust in den Filterstufen, oder durch Bedingungen in der Umwelt, die zu einer unvollständigen Wahrnehmung der Umwelt führen. Folgende Messfehler können dabei entstehen:

1. Ständige Abweichung der gemessenen Position von der realen Position
2. Fehlerhafte Erkennung und damit starke Abweichung von der realen Position
3. Nicht Erkennen der Position eines real vorhandenen Objektes

Diese Fehler können auf diese Weise herausgefiltert werden, so dass ihre Auswirkungen auf das Ergebnis, vermindert oder sogar vollständig eliminiert werden.

5.3.1 Positionsverfolgung der Elemente mittels Kalmanfilter

Ein Zyklus des Kalman-Filters besteht aus einem Prediction-Schritt und einem Update-Schritt. Für beide Schritte müssen die entsprechenden Modelle aufgestellt werden. Für den Prediction-Schritt ist es das Prozessmodell, welches die zeitlichen Veränderungen der Position möglichst gut beschreiben kann. Die folgenden drei Prozessmodelle sind mögliche Kandidaten, die für die Positionsvorhersage geeignet währen.

1. Die Elemente in der Umgebung stehen an festen Positionen und ändern sich nicht. Befindet sich ein Element zum Zeitpunkt t_{k-1} an einer bestimmten Position (x_{k-1}, y_{k-1}), dann ist zu erwarten, dass dieses Element sich auch zum Zeitpunk t_k an derselben Position befindet. Das kann mit folgenden Gleichungen ausgedrückt werden.

$$\mathbf{x_k} = \mathbf{x_{k-1}} \quad \textbf{(5.9)}$$

$$\mathbf{y_k} = \mathbf{y_{k-1}} \quad \textbf{(5.10)}$$

Es wird als von der Position $(\mathbf{x_{k-1}}, \mathbf{y_{k-1}})$, die das Element zum Zeitpunkt $\mathbf{t_{k-1}}$ hatte, auf die Position $(\mathbf{x_k}, \mathbf{y_k})$ zum Zeitpunkt $\mathbf{t_k}$ geschlossen. Eventuelles Auftreten von Bewegungen gilt als Abweichung vom Prozessmodell.

2. Die Elemente in der Umgebung bewegen sich. Sie haben dabei eine feste Geschwindigkeit und Bewegungsrichtung. Wenn diese Geschwindigkeit und Richtung bekannt ist, dann kann die Position $(\mathbf{x_k}, \mathbf{y_k})$ zum Zeitpunkt $\mathbf{t_k}$ aus der Position $(\mathbf{x_{k-1}}, \mathbf{y_{k-1}})$ vorhergesagt werden. Daher wird die Geschwindigkeit und die Richtung mit als Teil des Elementzustandes hinzugefügt. Die folgenden Gleichungen beschreiben dieses Modell.

$$\mathbf{x_k} = (\mathbf{t_k} - \mathbf{t_{k-1}})\,\Delta\mathbf{x_{k-1}} + \mathbf{x_{k-1}} \quad \textbf{(5.11)}$$

$$\mathbf{y_k} = (\mathbf{t_k} - \mathbf{t_{k-1}})\,\Delta\mathbf{y_{k-1}} + \mathbf{y_{k-1}} \quad \textbf{(5.12)}$$

$$\Delta\mathbf{x_k} = \Delta\mathbf{x_{k-1}} \quad \textbf{(5.13)}$$

$$\Delta\mathbf{y_k} = \Delta\mathbf{y_{k-1}} \quad \textbf{(5.14)}$$

Der Vektor $(\Delta\mathbf{x_k}, \Delta\mathbf{y_k})$ ist hier der Geschwindigkeitsvektor, der die Bewegungsrichtung anzeigt und dessen Betrag die Geschwindigkeit ist. Als Abweichungen von diesem Modell gelten hier eventuell auftretende Änderungen der Geschwindigkeit oder Richtung.

3. Die Elemente in der Umgebung bewegen sich. Sie können ihre Geschwindigkeit und Richtung mittels Beschleunigung ändern. Die Beschleunigung wird als konstant angesehen. Die Vorhersage basiert nun auch auf der Kenntnis über die Beschleunigung. Daher wird sie dem Zustand des Elementes hinzugefügt. Damit beschreiben die folgenden Gleichungen dieses Modell.

$$\mathbf{x_k} = \frac{\mathbf{ax_{k-1}}}{2}(\mathbf{t_k} - \mathbf{t_{k-1}})^2 + \Delta\mathbf{x_{k-1}}(\mathbf{t_k} - \mathbf{t_{k-1}}) + \mathbf{x_{k-1}} \quad \textbf{(5.15)}$$

$$\mathbf{y_k} = \frac{\mathbf{ay_{k-1}}}{2}(\mathbf{t_k} - \mathbf{t_{k-1}})^2 + \Delta\mathbf{y_{k-1}}(\mathbf{t_k} - \mathbf{t_{k-1}}) + \mathbf{y_{k-1}} \quad \textbf{(5.16)}$$

$$\Delta\mathbf{x_k} = (\mathbf{t_k} - \mathbf{t_{k-1}})\,\mathbf{ax_{k-1}} + \Delta\mathbf{x_{k-1}} \quad \textbf{(5.17)}$$

$$\Delta\mathbf{y_k} = (\mathbf{t_k} - \mathbf{t_{k-1}})\,\mathbf{ay_{k-1}} + \Delta\mathbf{y_{k-1}} \quad \textbf{(5.18)}$$

$$\mathbf{ax_k} = \mathbf{ax_{k-1}} \quad \textbf{(5.19)}$$

$$\mathbf{ay_k} = \mathbf{ay_{k-1}} \quad \textbf{(5.20)}$$

Hier ist der Vektor $(\mathbf{ax_k}, \mathbf{ay_k})$ der Beschleunigungsvektor, der die Wirkrichtung anzeigt und dessen Betrag die Beschleunigung ist. Änderungen der Beschleunigung gelten hier als Abweichungen vom Modell.

Da der Einsatz in mobilen Robotern vorgesehen ist, ist zu erwarten, dass diese sich häufig bewegen. Daher ist das erste Model weniger geeignet. Lediglich für statische Elemente, wie die Spielfeldbegrenzung, eignet es sich gut. Daher währen die letzten beiden Modelle besser geeignet, um die Dynamik anderer Elemente zu beschreiben. Bei der Entscheidung für ein Model sollten außer der Qualität der Prozessbeschreibung auch noch die Komplexität beachtet werden. Hier haben die verwendeten Zustandsvektoren in jedem Modell unterschiedliche Größe. Je größer diese Vektoren sind, desto komplexer sind die Berechnungen des Kalmanfilters. Während im ersten Model **2x2** Matrix Operationen nötig sind, sind es im zweiten Modell schon **4x4** Matrizen, und im dritten sogar **6x6**. Eine weitere Schwierigkeit ist die Initialisierung aller Werte des Zustandes, da für Geschwindigkeit und Beschleunigung keine zusätzlichen Sensoren zur Verfügung stehen, müssen diese ebenfalls mit aus den Daten des Elementfilters extrahiert werden. Für die Initialisierung der Position ist eine gemessene Elementposition nötig. Um die Geschwindigkeit und die Beschleunigung zu initialisieren, sind mindestens zwei bzw. drei Elementpositionen nötig. Für die Elementfusion ist hier schließlich die Entscheidung für das zweite Modell gefallen. Abbildung 27 zeigt grafisch die Positionsvorhersage für ein Element.

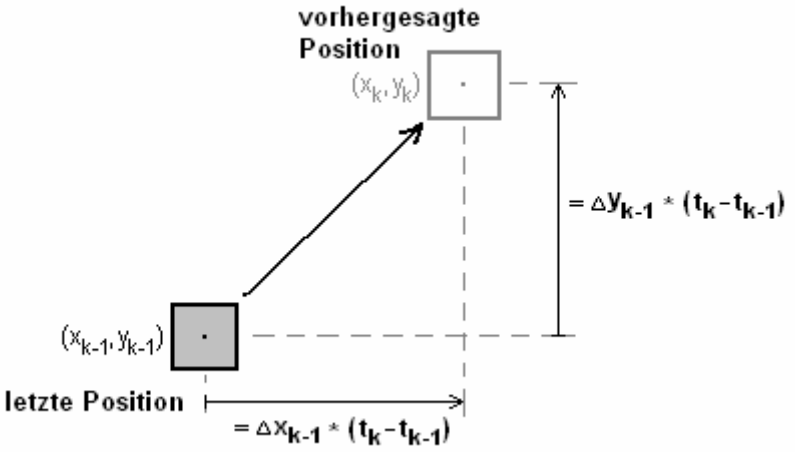

Abbildung 27: Positionsvorhersage

Da ein Prozessmodel vorhanden ist, kann der Prediction-Schritt durchgeführt werden.

$$\tilde{\mathbf{v}}_k = \mathbf{A}\,\hat{\mathbf{v}}_{k-1} \qquad \textbf{(5.21)}$$

$$\tilde{\mathbf{P}}_k = \mathbf{A}\mathbf{P}_{k-1}\mathbf{A}^T + \mathbf{Q} \qquad \textbf{(5.22)}$$

Der Vector $\mathbf{v_k}$ ist der reale Zustandsvektor $(\mathbf{x_k}, \mathbf{y_k}, \mathbf{\Delta x_k}, \mathbf{\Delta y_k})$ des Elements zum Zeitpunkt $\mathbf{t_k}$, der nicht bekannt ist, aber als Vektor $\hat{\mathbf{v}}_k$ geschätzt wurde. Der Vektor $\tilde{\mathbf{v}}_k$ repräsentiert den vorhergesagten Zustand des Elementes, ausgehend vom Vektor $\hat{\mathbf{v}}_{k-1}$. Die Gleichungen des Prozessmodels, hier als Matrix **A**, bilden also den Vektor $\hat{\mathbf{v}}_{k-1}$ auf den Vektor $\tilde{\mathbf{v}}_k$ ab. Die Matrix **A** ist nach dem zweiten Prozessmodel eine **4x4** Matrix und sieht wie folgt aus.

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & (t_k - t_{k-1}) & 0 \\ 0 & 1 & 0 & (t_k - t_{k-1}) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(5.23)}$$

Die Matrix $\tilde{\mathbf{P}}_k$ ist die vorhergesagte Kovarianz, ausgehend von der Kovarianz-Matrix $\mathbf{P}_k$. Die Matrix $\mathbf{Q}$ ist dabei einer der Parameter, mit dem der Kalman-Filtering Prozess beeinflusst werden kann. Mit ihr ist es möglich, den Einfluss der vorhergesagten Zustände auf das Ergebnis zu bestimmen. Dabei gilt, dass je größer $\mathbf{Q}$ ist, desto weniger Einfluss haben die vorhergesagten Zustände zu Gunsten der gemessenen Zustände. Die Gleichungen **(5.21)** und **(5.22)** führen damit eine Prediction durch, von der letzten geschätzten Position zu der Position, an der sich das Element zu dem Zeitpunkt t_k befindet, zu dem eine neue Position gemessen wurde. Die Zeitinformation t_{k-1} und t_k werden mittels Zeitstempel zur Verfügung gestellt, wobei t_{k-1} der Zeitstempel der letzten Messdaten ist, und t_k der der aktuellen Messdaten.

Nachdem ein Prediction-Schritt durchgeführt wurde, muss nun ein neuer Zustand mit Hilfe aktueller Messdaten geschätzt werden. Dies geschieht im Update-Schritt nach folgenden Gleichungen.

$$K_k = \tilde{P}_k H^T (H \tilde{P}_k H^T + R)^{-1} \quad \textbf{(5.24)}$$

$$\hat{\mathbf{v}}_k = \tilde{\mathbf{v}}_k + \mathbf{K}_k (z_k - \mathbf{H}\tilde{\mathbf{v}}_k) \quad \textbf{(5.25)}$$

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k \mathbf{H})\tilde{\mathbf{P}}_k \quad \textbf{(5.26)}$$

Die Matrix $\mathbf{R}$ ist wie $\mathbf{Q}$ ein Parameter, mit dem der Kalman-Filtering Prozess beeinflusst werden kann. Mit ihr ist es möglich, den Einfluss der aktuell gemessenen Zustände auf das Ergebnis zu bestimmen. Dabei gilt analog zu $\mathbf{Q}$, dass je größer $\mathbf{R}$ ist, desto weniger Einfluss haben die gemessenen Zustände zu Gunsten der vorhergesagten Zustände. Die Matrix $\mathbf{K}_k$ wird als Kalman-Gain bezeichnet. Der Vektor $\mathbf{z}_k$ enthält die, für den Update-Schritt notwendigen Messwerte. Für das Prozessmodell wird zusätzlich die Geschwindigkeit benötigt. Da aber, wie schon erwähnt, keine zusätzlichen Sensoren zur Geschwindigkeitsbestimmung zu Verfügung stehen, muss diese aus zwei bekannten Positionen berechnet werden. Der Vektor $\mathbf{z}_k$ enthält daher zwei Positionen und hat die Form $(\mathbf{x}_{k-1}, \mathbf{y}_{k-1}, \mathbf{x}_k, \mathbf{y}_k)$. Wobei die Position $(\mathbf{x}_{k-1}, \mathbf{y}_{k-1})$ aus dem letzten Schätzwert $\hat{\mathbf{v}}_k$ stammt und die Position $(\mathbf{x}_k, \mathbf{y}_k)$ vom Elementfilter erhaltenen wird, der sie wiederum aus den aktuellen Sensordaten extrahiert hat. Mit den folgenden Gleichungen kann nun aus einem Vektor $\mathbf{z}_k$ ein Vektor der Form $\mathbf{v}_k$ berechnet werden.

$$\mathbf{x}_k^{\mathbf{v}} = \mathbf{x}_k^{\mathbf{z}} \quad \textbf{(5.27)}$$

$$\mathbf{y}_k^{\mathbf{v}} = \mathbf{y}_k^{\mathbf{z}} \quad \textbf{(5.28)}$$

$$\Delta x_k^v = \frac{x_k^z - x_{k-1}^z}{t_k - t_{k-1}} \quad \textbf{(5.29)}$$

$$\Delta y_k^v = \frac{y_k^z - y_{k-1}^z}{t_k - t_{k-1}} \quad \textbf{(5.30)}$$

Das **v** und das **z** sind hier nicht als Exponenten gedacht, sondern zeigen lediglich die Herkunft aus dem Vektor $\mathbf{v_k}$ bzw. $\mathbf{z_k}$. In den Gleichungen **(5.24)** bis **(5.26)** führt die Matrix **H** eine Abbildung in die Entgegengesetzte Richtung, also eine Abbildung von $\mathbf{v_k}$ nach $\mathbf{z_k}$. Die Matrix **H** ist damit eine **4x4** Matrix, und sieht wie folgt aus.

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & -(t_k - t_{k-1}) & 0 \\ 0 & 1 & 0 & -(t_k - t_{k-1}) \end{bmatrix} \quad \textbf{(5.31)}$$

Im Update-Schritt kann nun, basierend auf die vorhergesagte Position und der aktuell gemessene Position, die aktuelle Position des Elements geschätzt werden.

Ein Prediction-Schritt und ein Update-Schritt bilden einen Zyklus des Kalman-Filtering Verfahrens. Ein Zyklus wird jedes Mal für ein bestimmtes Element durchlaufen, wenn für dieses Element eine neue Position vom Elementfilter extrahiert wurde. Im Falle der Fusion von verteilt verarbeiteten Sensordaten auf der Stufe der Elemente, kommen diese von den Elementfilter unterschiedlicher Teilnehmer, was für die Elementfusion jedoch keine Bedeutung hat, da sie mittels Kalman-Filtering Verfahren eventuelle zeitliche Diskrepanzen im Prediction-Schritt auflöst. Dafür werden zeitliche Informationen über die Position des Elementes benötigt, die aber gerade durch die globalgültigen Zeitstempel, die im ersten Abschnitt dieses Kapitels vorgestellt wurden, zur Verfügung gestellt werden.

5.3.2 Positionsverfolgung mehrerer Elemente

Die Lösung mittels Kalman-Filtering Verfahren die Positionen eines Elements zu verfolgen, setzt voraus, dass die vom Elementfilter gelieferten Positionen auch zu diesem speziellen Element gehören. Er liefert jedoch mehrere Positionen, wenn mehrere Objekte in der Umgebung observiert wurden. Damit stellt sich die Frage: „Welche Elementposition gehört zu welchem Element?“. Im Elementfilter werden zwar Objekte als ein bestimmter Typ identifiziert, jedoch reicht das für eine eindeutige Positionszuordnung nicht aus, da mehrere Elemente des gleichen Typs wahrgenommen werden können. Zum Beispiel ist im Robocup anzunehmen, dass es nur eine Bande und nur einen Ball gibt, aber es existieren mehrere Roboter, seinen sie aus dem eigenem oder gegnerischem Team, auf dem Spielfeld. Zu ähnlichen Problemen existieren bereits Arbeiten, die sich damit unter der Themenüberschrift „Multiple Object Tracking“ beschäftigen. Einige dieser Arbeiten wurden bereits in Kapitel 3 erwähnt. In diesen Arbeiten wurden unter anderem der „multiple hypothesis tracking“ **MHT** Algorithmus [22] untersucht

und ein klassisches „nearest neighbor“ **NN** Verfahren [23] angewandt, bzw. untersucht. Um aber nicht den Rahmen dieser Diplomarbeit zu sprengen, wird hier für die Elementfusion ein eigenes einfacheres Verfahren angewandt.

Die Elementfusion verfolgt die Positionen von m Elementen, wobei m als fester bekannter Wert angenommen wird und daher unabhängig von der Anzahl der erkannten Elementpositionen des Elementfilters ist, der auch weniger oder auch mehr als m Positionen wahrnehmen kann. Dieser sollte, basierend auf den Regeln des Robocup (Anzahl von Bällen und Robotern auf dem Feld), festgelegt werden. Den m Elementen müssen nun die Positionen zugeordnet werden. Dabei wird immer ein Element ausgewählt, dessen Position wird mittels Prediction-Schritt vorhergesagt, und zwar auf die Zeit des Zeitstempels der Positionen, aus den Positionen wird eine diesem Element zugeordnet und damit der Update-Schritt durchgeführt. Das wird solange wiederholt, bis alle Elemente mit je einer Position aktualisiert wurden, oder keine weiteren Positionen mehr existieren. Diese Folge der Verarbeitung hat den Vorteil, dass die Updates der Elemente ziemlich gleichmäßig über die gesamte Laufzeit verteilt sind, was wiederum von Vorteil ist, falls ein Abbruch Seitens des Schedulers eintritt, da in diesem Fall die Chancen gut stehen, dass zu mindest ein Teil der Elemente aktualisiert wurden. Die einzelnen Schritte der Elementfusion werden hier nun genauer erklärt.

Auswahl der Elemente: Als erstes muss festgelegt werden, in welcher Reihenfolge die Elemente aktualisiert werden. Hier wird immer das aktuellste Element zuerst bearbeitet. Das heißt, das Element, dessen letzte Aktualisierung zeitlich am wenigsten lang zurückliegt. Die Idee dahinter ist, dass die vorhergesagte Position eines Elements präziser ist je aktueller das Element ist. Damit sind Fehler bei der Positionszuordnung weniger wahrscheinlich, womit auch weniger die Gefahr besteht, dass einem Element eine Position zugeordnet wird, die eigentlich zu einem anderen Element gehört, was aber noch nicht bearbeitet wurde. Als Beispiel sei angenommen, es werden seit einem gewissen Zeitraum die Positionen zweier Elemente in der Elementfusion verfolgt. Diese Elemente liegen räumlich relativ eng zusammen, so dass sie gerade noch vom Elementfilter als zwei separate Elemente wahrgenommen werden. Sie bewegen sich außerdem parallel mit gleicher Geschwindigkeit und Richtung. Nun aber extrahiert der Elementfilter aus den folgenden Daten nur noch eine Position von den beiden. Die Folge ist, dass nur einem von beiden Elementen eine Position zugeordnet werden kann. Damit ist dieses Element auch das aktuellere. Angenommen, es würde nun immer das weniger aktuelle Element als erstes ausgewählt, um die Zuordnung durchzuführen, dann würde diesem Element sie Position zugeordnet und das ehemals aktuellere Element wird als Folge zu dem weniger aktuellen. Das bedeutet, dass immer im Wechsel die Positionen mal dem ein und mal dem anderen Element zugeordnet werden. Damit würde in der Elementfusion also dasselbe Objekt doppelt verfolgt, was nicht erwünscht wird. Daher sollte in den folgenden Verarbeitungsschritten immer dem aktuelleren Element diese Positionen zugeordnet werden, während das andere Element seine Aktualität immer mehr verliert, es sei den, der Elementfilter kann irgendwann wieder die zweite Position extrahieren. Die Elemente in der Reihenfolge ihrer Aktualität auszuwählen ist daher sinnvoll.

Kalman Prediction-Schritt: Es wird mir dem ausgewählten Element der Prediction-Schritt durchgeführt, um die Positionen der Elemente zu der Zeit vorherzusagen, zu denen die aktuell gemessenen Positionen wahrgenommen wurden.

Positionszuordnung: Es wird unter den Positionen vom Elementfilter die ausgewählt, die der vorhergesagten Position des Elementes am nächsten liegt (Nearest Neighbor NN). Diese Position wird dem Element zugeordnet (Abbildung 28).

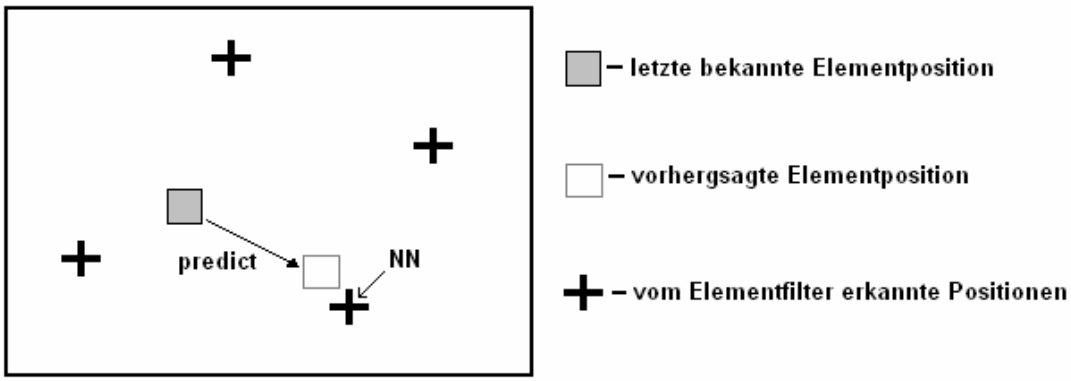

Abbildung 28: NN Positionszuordnung

Überprüfung der Entfernung: Hier wird überprüft, ob die Entfernung zwischen der vorhergesagten Position und der zugeordneten Position unter einem bestimmten Schwellenwert S_k liegt. Der Schwellenwert ist dabei abhängig von dem maximal zu erwartenden Messfehler e_{max} der Position, verursacht durch gaußschem Rauchen während der Sensoraktivität oder Fehler bei der Datenverarbeitung, und von dem maximal zu erwartenden Abweichung vom Prozessmodell, was in diesem Fall also die maximal zu erwartende Beschleunigung a_{max} ist.

$$S_k = \frac{a_{max}}{2}(t_k - t_{k-1})^2 + e_{max} \qquad \textbf{(5.32)}$$

Ist die Entfernung größer als dieser Schwellenwert, dann wird der Update-Schritt nicht durchgeführt. Ist er kleiner, dann wird die Position als zugeordnet markiert, damit Mehrfachzuordnungen vermieden werden. Die Idee dahinter ist, dass Zuordnungen, die als unwahrscheinlich gelten, herausgefiltert werden. Anschließend wird der Update-Schritt des Kalman-Filtering Verfahrens mit der zugeordneten Position durchgeführt.

Kalman Update-Schritt: Hier wird mit Hilfe der vorhergesagten Position und der aktuell gemessenen Position der aktuelle Zustand des Elementes geschätzt.

Der gesamte Ablauf der Elementfusion in einer Periode ist hier in Abbildung 29 noch einmal zusammengefasst.

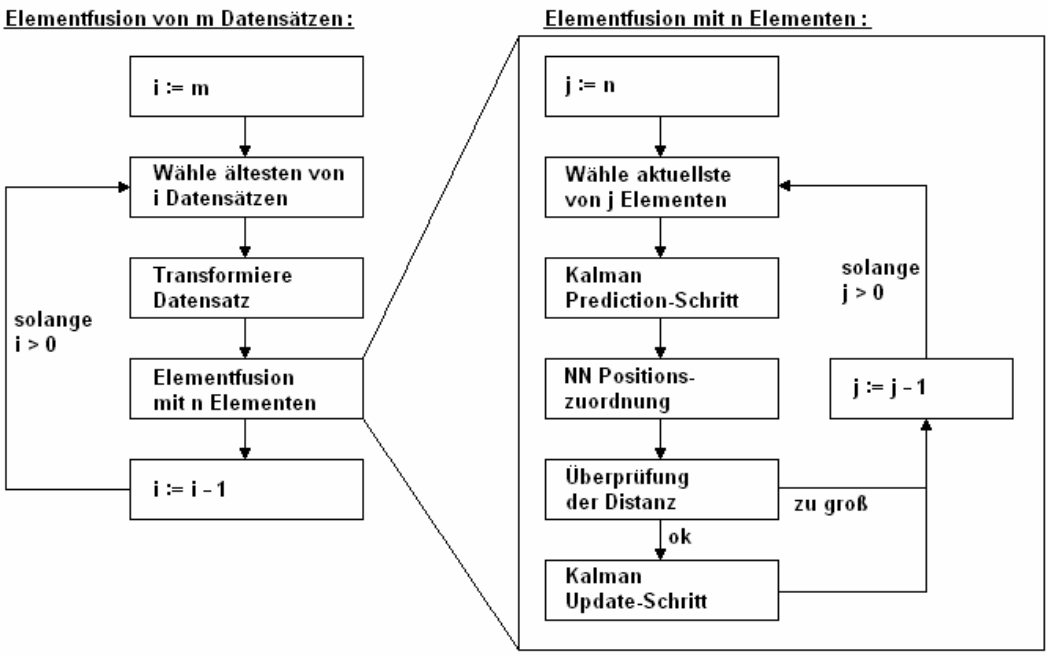

Abbildung 29: Elementfusion

6 Implementierung

Dieses Kapitel beschäftigt sich mit der Umsetzung der, in den beiden vorangegangenen Kapiteln, entwickelten Lösungen. Die Konzepte wurden in Form von wieder verwendbaren Klassen in C++ implementiert, die für den Einsatz in RTLinux Modulen für RTLinux 3.1 [26] gedacht sind. Der erste Abschnitt stellt diese implementierten Klassen vor, während sie im zweiten Abschnitt verwendet werden, um die entwickelten Konzepte auf die verteilte Fusion von Lasersensordaten [15] anzuwenden.

6.1 Klassen der mehrstufigen, verteilten Datenverarbeitung

Die FFM(Filter-Fusion-Module)-Klassen der mehrstufigen, verteilten Datenverarbeitung bilden eine Art Framework, und sind so konzipiert, dass sie mittels Vererbung in die Applikationen der Nutzer eingebunden werden können. In ihnen sind die Konzepte aus Kapitel 4 und 5 umgesetzt. Die Nutzerapplikation ist dabei ein Kernel-Modul für RTLinux [26] und wird gestartet, indem es in den Kernel dynamisch eingebunden wird. Das Klassendiagramm in Abbildung 30 zeigt die entwickelten Klassen und die Verwendung durch den Anwender mittels Vererbung.

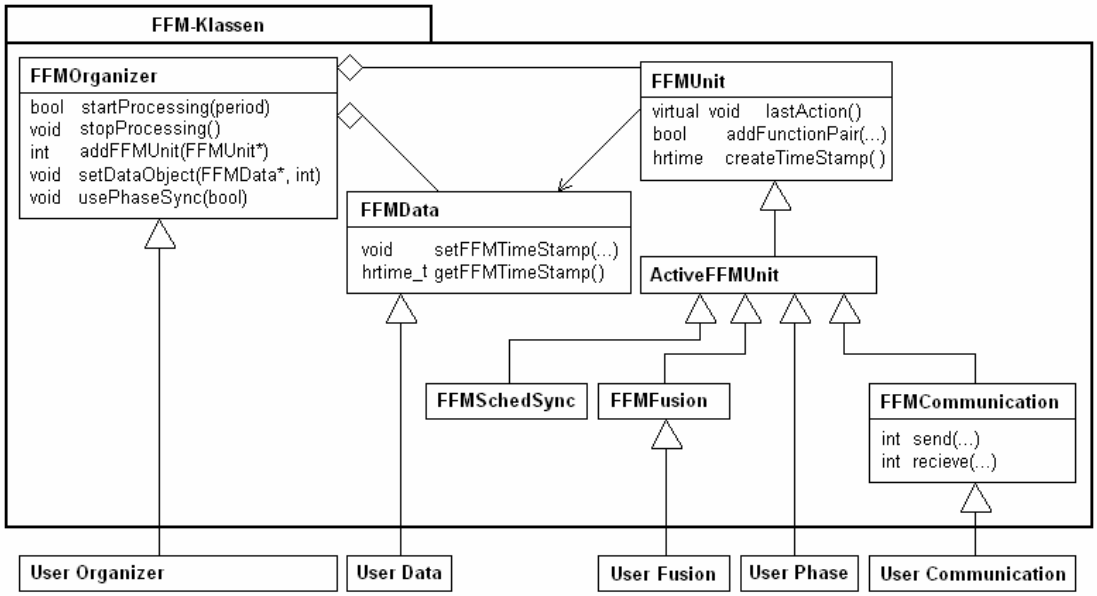

Abbildung 30: FFM-Klassen

6.1.1 Umsetzung der mehrstufigen Datenverarbeitung

Das vorgestellte Task-Konzept ist in den Klassen `FFMUnit`, `ActiveFFMUnit` und `FFMOrganizer` implementiert. Ein Objekt der Klasse `ActiveFFMUnit` bzw. `FFMUnit` repräsentiert dabei eine Phase der Datenverarbeitungskette. Die Datenverarbeitungskette und ihre Phasen werden dabei von einer Instanz der Klasse `FFMOrganizer` verwaltet.

Will nun ein Anwender eine mehrstufige Datenverarbeitungskette erstellen, so muss er als erstes die einzelnen Phasen implementieren. Dies kann er tun, indem er für jede Phase je eine Klasse erstellt, die von der Klasse `ActiveFFMUnit` abgeleitet ist, und diese dann je um die gewünschte Funktionalität erweitert. Die aus diesen Klassen erzeugten Instanzen muss er dann dem `FFMOrganizer` Objekt in der Reihenfolge übergeben, in der auch die entsprechenden Phasen pro Verarbeitungsperiode ausgeführt werden sollen, und ihm dann den Befehl zum Start der Datenverarbeitung geben. Das `FFMOrganizer` Objekt übernimmt die zeitliche Aufteilung der Periodendauer in Phasen. Dazu benötigt es vom Anwender zu jeder Phase die ECETs der Mainparts MPs und die WCETs der Exceptionparts EPs, und es braucht die Dauer einer Periode. Im folgendem werden nun die Aufgaben der einzelnen Klassen ausführlicher erklärt.

`FFMUnit:`

Der Anwender implementiert die Funktionalität der einzelnen Phasen in je einer Klasse. Die Klasse `FFMUnit` dient dabei je als indirekte Basisklasse. Der Anwender legt nun Funktionspaare, als Methoden der entsprechenden Klassen, an. Eine Funktion für den MP und eine für den EP. Er kann beliebig viele Funktionspaare in einer Klasse anlegen, wenn die darin implementierten Aktionen der Phase unabhängig voneinander im selben Zeitfenster ausgeführt werden sollen, wie es im zweiten Abschnitt des vierten Kapitels erwähnt wurde. Diese Funktionen bzw. Methoden haben die folgende Form.

```
int <Klassenname>::<Funktionsname> (void)
{
      …
}
```

Die Funktionsnamen können beliebig gewählt werden. Da die Klasse `FFMUnit` diese Methoden nicht kennt, muss der Anwender sie ihr bekannt machen, durch den Aufruf der folgenden Methode in der Constructor-Funktion, womit er außerdem die dazugehörigen ECETs und WCETs der Funktionen mit übergibt.

```
bool FFMUnit::addFunctionPair(int (FFMUnit::mp)(void),
                              int (FFMUnit::ep)(void),
                              hrtime_t ecet,
                              hrtime_t wcet);
```

Jedes Funktionspaar wird dem `FFMUnit` Objekt auf diese Weise, durch je einen Aufruf dieser Methode, bekannt gemacht. Die MPs werden dann im laufenden Betrieb periodisch aktiviert, wenn Daten zu Verarbeitung anliegen. Die Daten können dabei von einer vorherigen Verarbeitungsphase bereitgestellt worden sein. In diesem Fall muss eine vorhergehende Phase der nachfolgenden dies signalisieren, durch den Aufruf der Methode `makeOutputAviable()`. Der Anwender ist dafür verantwortlich, dass sie aufgerufen wird, wenn Daten für die nachfolgende Phase bereitgestellt werden sollen. Ein möglicher Ort für den Aufruf dieser Funktion ist die Methode `virtual void lastAction()`. Sie wird garantiert genau einmal pro Periode aktiviert, wenn in dieser Periode Daten verarbeitet wurden. Es wird außerdem garantiert, dass sie vor der Deadline aus-

geführt wird und nachdem alle Berechnungen dieser Phase beendet sind. Damit ist es möglich, nach jeder Verarbeitungsphase letzte Aktionen, wie Aufräumarbeiten, Speichern der Ergebnisse usw., auszuführen. Da sie als `virtual` deklariert ist, kann sie vom Anwender in der abgeleiteten Klasse überschrieben werden.

```
void <Klassennahme>::lastAction()
{
      …
      makeOutputAviable();
      …
}
```

Aufgerufen wird diese Methode durch den phaseninternen Thread, der als letztes seine Berechnungen der aktuellen Periode beendet. Die Auswahl dieses Threads erfolgt intern durch ein Verfahren, dass dem Spiel „Reise nach Jerusalem“ ähnelt. In diesem Spiel gibt es n Teilnehmer, aber nur n-1 Stühle. Auf ein Signal hin versuchen alle Teilnehmer einen Stuhl zu besetzen. Der Teilnehmer, der übrig bleibt, ist der Verlierer. In der Klasse `FFMUnit` sind hier die Teilnehmer die internen Threads und statt Stühle werden Mutexes vom Typ `pthread_mutex_t` verwendet. Jeder Thread versucht am Ende seiner Berechnungen solch ein Mutex zu besetzen. Sind jedoch bereits alle belegt, dann ist dieser Thread für die Ausführung der Methode `lastAction()` verantwortlich. Da ein Thread aus einem MP und einem EP besteht, so besteht auch die Möglichkeit, dass diese Methode im Kontext des EPs ausgeführt wird. Daher sollte der Anwender daran denken, die maximale Ausführungszeit der Methode `lastAction()` in die Bestimmung der WCETs der EPs mit einzubeziehen, damit vollständige Ausführung innerhalb der gesetzten Deadline garantiert werden kann. Des Weiteren sollte der Anwender die Richtlinie im Hinterkopf behalten, die Methode, wie die EPs auch, minimal klein zu halten. Das heißt, aufwendige Berechnungen sollten vermieden werden oder in die MPs der Phase verlagert werden.

`ActiveFFMUnit:`

Die Klasse `ActiveFFMUnit` ist von der Klasse `FFMUnit` abgeleitet. Sie dient als direkte oder indirekte Basisklasse bei der Implementierung der Verarbeitungsphasen des Anwenders. Ihre Funktion ist die Verwaltung und Start der Threads, die zu dieser Phase gehören. Sie ist es, die dem TAFT-Scheduler die Parameter Release Time, Dealine und Periodendauer übergibt und damit ein Ausführungszeitfenster für die dazugehörige Verarbeitungsphase schafft. Als Voraussetzung muss auf dem Anwendersystem die Version des TAFT-Schedulers für RTLinux installiert sein.

`FFMOrganizer:`

Die Klasse `FFMOrganizer` ist für die Infrastruktur der Verarbeitungsphasen verantwortlich (Abbildung 31) und verwaltet dazu noch die Datenobjekte, die vom Anwender mit folgender Methode übergeben werden:

```
void FFMOrganizer::setDataObject(FFMData* dat, int dln);
```

Mit je einem Aufruf wird ein Datenobjekt überreicht. Der Anwender muss daher so viele Datenobjekte übergeben, wie Systeme an der verteilten Datenverarbeitung beteiligt sind. Mit dem Parameter `dln` ordnet er jedes Datenobjekt einem Teilnehmersystem zu.

Der Anwender übergibt nun alle Objekte, die die einzelnen Verarbeitungsphasen repräsentieren und indirekt `FFMUnit` Objekte sind, an das `FFMOrganizer` Objekt. Dieses baut daraus die Verarbeitungskette auf, wobei jeder Phase jeweils, soweit vorhanden, der Nachfolger und Vorgänger bekannt gemacht wird. Die Übergabe erfolgt mit der folgenden Methode:

```
int FFMOrganizer::addFFMUnit(FFMUnit* ffmunit);
```

Die Phasen müssen dabei, entsprechend der gewünschten Ausführungsreihenfolge, auch in dieser Reihenfolge auf diese Weise übergeben werden. Wenn dies geschehen ist, kann der Anwender die periodische Datenverarbeitung starten, durch den Aufruf dieser Methode:

```
bool FFMOrganizer::startProcessing(hrtime_t prd);
```

Dabei wird die Periodendauer `prd` geliefert. Der FFMOrganizer überprüft daraufhin die Parameter der einzelnen Phasen auf die Weise, wie sie im vierten und fünften Kapitel in der Gleichung **(4.14)** beschrieben wurde, oder Gleichung **(5.6)** wenn die globale Koordinierung verwendet wird. Anschließend wird die Periode in Zeitfenster, nach den Gleichungen **(4.17)**, **(4.18)** oder **(5.7)**, **(5.8),** aufgeteilt, und alle Phasen werden nacheinander aktiviert. Mit dem Aufruf von `stopProcessing()` kann der Anwender die Ausführung wiederum anhalten.

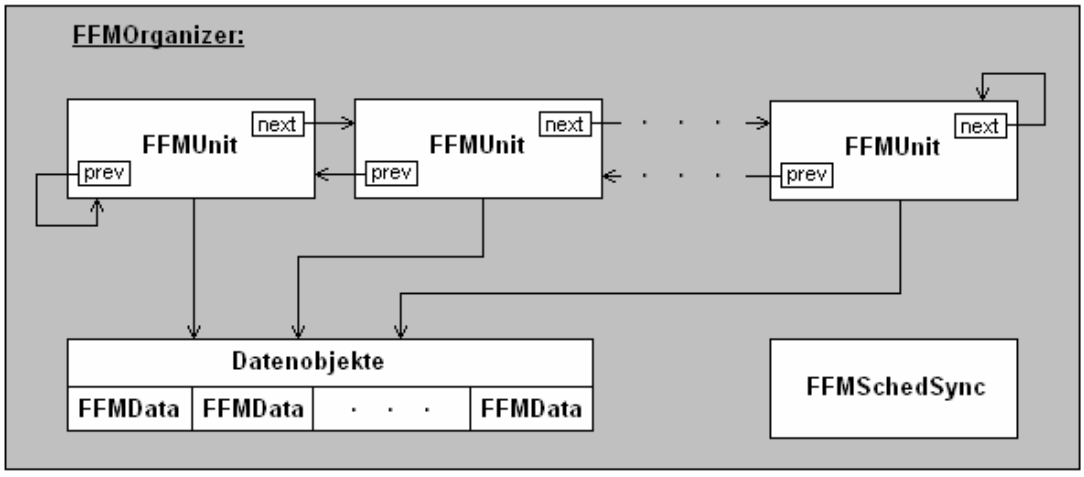

Abbildung 31: Funktion des FFMOrganizer Objektes

6.1.2 Zeitstempel und globale Zeitgesteuerte Koordinierung

Um globale Zeitstempel verwenden zu können, muss auf den Anwendersystemen je eine Orinoco WLAN Karte vorhanden sein. Dazu muss der rt_orinoco Treiber für

RTLinux installiert sein, mit der Erweiterung WLAN Uhrensynchronisation [7]. Durch diese wird eine globale Uhr zur Verfügung gestellt, auf die alle beteiligten Systeme zugreifen können. Die Methode `createTimeStamp()` der Klasse `FFMUnit` erlaubt dem Anwender die aktuelle globale Uhrzeit zu erhalten und einen Zeitstempel zu kreieren. Die Zeitstempel können daraufhin in den Datenobjekten gesetzt werden, wenn Beispielsweise die Daten erzeugt werden. Auf der globalen Uhr basiert ebenfalls die globale Zeitgesteuerte Koordinierung der Periodenstartzeiten. Mit der folgenden Methode der `FFMOrganizer` Klasse kann der Anwender sie aktivieren.

```
void FFMOrganizer::usePhaseSync(bool set);
```

Das `FFMOrganizer` Objekt bestimmt dann beim Beginn der Datenverarbeitung eine statisch vereinbarte Startzeit nach der globalen Uhr (Gleichung **(5.2)** und **(5.3)**), so dass auf allen Systemen die Perioden Phasensynchron aktiviert werden. In der Klasse `FFMSchedSync` ist der Mechanismus implementiert, der verhindert, dass die Periodenstartzeiten im laufenden Betrieb auseinanderdriften. Die Aufgaben der Klassen `FFMData` und `FFMSchedSync` sind hier noch etwas ausführlicher erläutert.

FFMData:

Diese Klasse dient als Basisklasse für die Datenobjekte des Anwenders. Er implementiert in den abgeleiteten Klassen die von ihm benötigte Funktionalität der Datenobjekte, was in der Regel die Speicherung der Sensordaten oder der daraus extrahierten Informationen ist. Der Anwender muss aus diesen Klassen genau so viele Datenobjekte erzeugen, wie Systeme an der verteilten Datenverarbeitung beteiligt sind. Dabei ist eines davon für die Speicherung der Daten des systemlokalen Sensors und den daraus extrahierten Informationen verantwortlich. Die anderen Datenobjekte dienen zur Speicherung der Daten der anderen Systeme und sind daher nur für die Fusionsstufen und die Kommunikationsphasen relevant. Nach diesen Phasen befinden sich alle Daten bzw. Informationen fusioniert in einem Datenobjekt. Das ist auf allen System das Objekt mit der Nummer `dln = 0`. Abbildung 32 zeigt, wann sich im Verlauf der Berechnung relevante Daten in den jeweiligen Datenobjekten befinden.

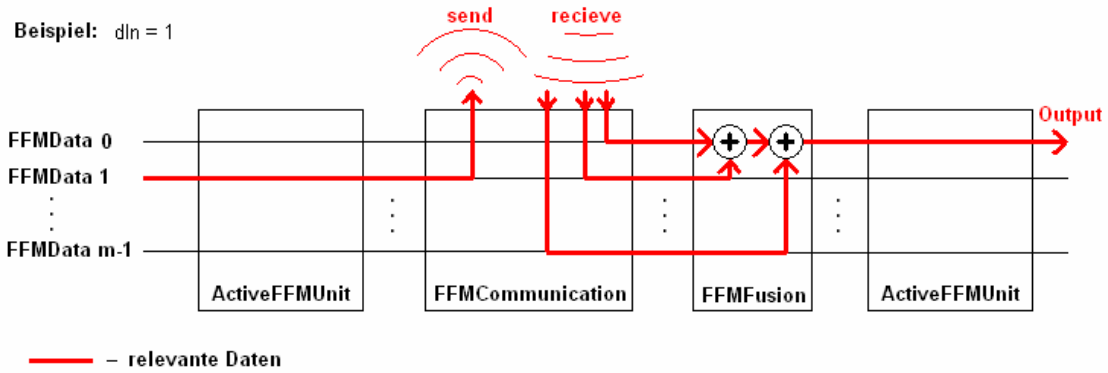

Abbildung 32: Relevanz der Datenobjekte im Verlauf der Berechnungen

In der Klasse `FFMData` sind außerdem die Methoden implementiert, um einen Zeitstempel zu setzen bzw. abzufragen.

```
void FFMData::setFFMTimeStamp(hrtime_t ts);

hrtime_t FFMData::getFFMTimeStamp();
```

Der Anwender kann damit den Daten Zeitstempel zuordnen.

`FFMSchedSync:`

Diese Klasse ist von der Klasse `ActiveFFMUnit` abgeleitet. Sie wird ähnlich wie eine Verarbeitungsphase behandelt. Ihrer Aktivität wird also ebenfalls ein festes Zeitfenster in jeder Periode zugeordnet, obwohl sie nicht zur eigentlichen Datenverarbeitungskette gehört. Die Aufgabe des `FFMSchedSync` Objektes ist es, in regelmäßigen Abständen die Periodenstartzeiten zu korrigieren, wie es im fünften Kapitel mit der Gleichung **(5.6)** beschrieben wurde. Zu diesem Zweck wurde die verwendete Version des TAFT-Schedulers um eine Funktion erweitert, die es erlaubt, alle folgenden Aktivierungszeiten der Threads in der Liste des Schedulers um eine geforderte Zeitspanne `diff` zu verschieben. Daher muss diese erweiterte Version des TAFT-Schedulers ebenfalls auf dem Anwendersystemen vorhanden sein, wenn die globale Koordinierung verwendet werden soll.

```
int rtl_sched_resumetime_move(hrtime_t diff);
```

Diese Funktion verschiebt die Zeiten der nächsten Aktivierungen aller Threads um den gegebenen Korrekturwert. Da dies ausnahmslos mit alle Threads geschieht, die im TAFT-Scheduler verwaltet werden, ist gesichert, dass sie alle geschedult werden können. Es ist daher nicht nötig, den Akzeptanztest des Schedulers erneut durchzuführen, da sich der Ausführungsplan nicht verändert, sonder ledig um einen geringen Zeitwert nach vorn oder nach hinten verschoben wird. Das `FFMSchedSync` Objekt wird nur aktiv, wenn der Anwender die globale Koordinierung im `FFMOrganizer` Objekt aktiviert hat.

6.1.3 Kommunikation und Fusion

Die Klassen `FFMCommunication` und `FFMFusion` stellen spezielle Funktionalität zur Kommunikation bzw. Fusion zur Verfügung. Sie dienen als Basisklassen für die Kommunikations- bzw. Fusionsklassen des Anwenders und sind selbst aus der Klasse `ActiveFFMUnit` abgeleitet, da die Fusion und Kommunikation ebenfalls Verarbeitungsphasen sind. Das spezielle an ihnen ist jedoch, im Gegensatz zu den normalen Verarbeitungsphasen, welche die Eingabedaten von einem Datenobjekt erhalten und die Ausgabedaten in dasselbe Datenobjekt legen, dass die Kommunikation die Eingabe von einem Datenobjekt erhält, aber die Ausgabe auf mehrere Datenobjekte verteilt und die Fusion verarbeitet mehrere Datenobjekte als Eingabe, aber legt die Ausgabe nur in einem Datenobjekt ab. Im Folgenden sind die Klassen `FFMCommunikation` und `FFMFusion` genauer erklärt.

FFMCommunikation:

In der Kommunikationsphase der verteilten Datenverarbeitung werden die lokalen Daten basierend auf den eigenen Sensoren über ein geeignetes Kommunikationsmedium mittels echtzeitfähigen Protokolls versandt und damit den anderen Teilnehmern zugänglich gemacht. Deren Daten werden ebenfalls in dieser Phase empfangen und für die folgende Fusion bereitgestellt. In der Klasse `FFMCommunikation` werden zu diesem Zweck dem Anwender bereits entsprechende Funktionen, wie das Senden und Empfangen von Daten bereitgestellt. Der Anwender implementiert seine eigene Kommunikationsphase, indem er eine Klasse von dieser ableitet. Die folgende Methode ist die Sendefunktion der Klasse `FFMCommunication`.

```
int FFMCommunication::send(unsigned char* buf, int len, FFMData* ts);
```

Der Anwender übergibt einen Zeiger `buf` auf seine Daten und die Länge `len` der Daten in Bytes. Dazu muss er die Daten eventuell in eine flache Struktur überführen. Obendrein liefert er noch eine Referenz `ts` auf das dazugehörige Datenobjekt. Das ist nötig, da darin der Zeitstempel der Daten vorhanden ist, der aber mit versandt werden muss, damit er anschließend auf allen Systemen mit den Daten vorhanden ist. Analog dazu die Empfangsfunktion.

```
int FFMCommunication::recieve(unsigned char* buf, int len, int dln,
FFMData* ts);
```

Hier wird wieder die Referenz `ts` des Datenobjektes übergeben, welches den übermittelten Zeitstempel aufnehmen soll. Mit dem Parameter `dln` wird mitgeteilt, von welchem System die Daten erwartet werden. Zurückgegeben wird die Länge der Daten in Bytes, falls überhaupt welche empfangen wurden.

Als Kommunikationsprotokoll wird hier das RGCP [5] via WLAN verwendet. Auf Grund seiner Echtzeitfähigkeit und der drahtlosen Kommunikation, ist es für das Einsatzgebiet der mobilen Robotik bestens geeignet. Das setzt voraus, dass auf dem Anwendersystem die entsprechende Hardware, also Orinoco WLAN Karte, und die Software, also der rt_orinoco Treiber und die RGCP Bibliothek für RTLinux, vorhanden sind. Die Anwenderdaten werden von dem `FFMCommunication` Objekt an das RGCP weitergereicht. Dabei werden die Daten eventuell aufgeteilt und in mehreren Portionen versandt, da das RGCP in einer Runde nur begrenzt viele Daten (1499 Bytes) versenden kann. Beim Empfangen werden dann die Daten wieder korrekt zusammengesetzt und als Ganzes von der Empfangsfunktion zurückgeliefert. Zu diesem Zweck werden den einzelnen Datenportionen zusätzliche Header (Abbildung 33) vorangestellt, in denen die Information über die Aufteilung mit übertragen wird. Außerdem wird in diesem Header auch der Zeitstempel übermittelt.

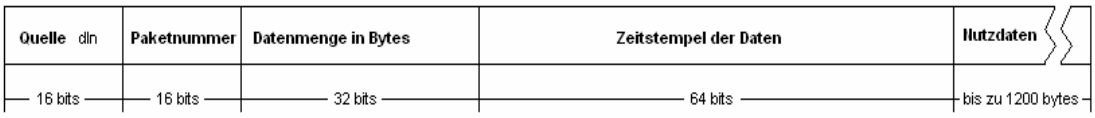

Abbildung 33: Kommunikationsheader für die Sensordaten

`FFMFusion:`

Die Klasse `FFMFusion` ist von der Klasse `ActiveFFMUnit` abgeleitet und repräsentiert ebenfalls wie sie eine Verarbeitungsphase. Sie ist jedoch speziell für die Fusion vorgesehen. Im Gegensatz zu gewöhnlichen `ActiveFFMUnit` Objekten, die nur ein Datenobjekt als Eingabe erwarten, benötigt sie mehrere Datenobjekte mit den Daten aller beteiligten Systeme. Daher werden hier speziell alle Datenobjekte auf Verfügbarkeit von Daten überprüft. Die Klasse `FFMFusion` dient somit als Basisklasse für die Implementierung der Fusionsstufen des Anwenders.

6.1.4 Implementierungsbeispiel: Schritt für Schritt

In diesem Abschnitt soll Schritt für Schritt erklärt werden, wie ein Anwender eine verteilte mehrstufige Sensordatenverarbeitungskette mit Hilfe der FFM-Klassen erstellen kann. Es sei hier angenommen, der Anwender besitzt m Systeme mit Sensoren. Sie sind von **0** bis **m-1** durchnummeriert und auf jedem sind die Hardware und Software Voraussetzungen erfüllt. Auf allen Systemen sollen die Sensordaten stufenweise durch Filter verarbeitet werden. Dabei durchlaufen sie periodisch genau **n** Filter, bis die endgültigen Informationen in jeder Periode als Ausgabe zur Verfügung stehen. Außerdem sollen die Systeme ihre Daten zwischen der **h**-ten Filterstufe und **h+1**-ten Filterstufe **(1 < h < n)** untereinander austauschen und fusionieren. Diese Verarbeitung soll innerhalb eines Moduls laufen, was mittels der FFM-Klassen erstellt wurde. Abbildung 34 zeigt den Aufbau, den das Modul am Ende besitzen soll.

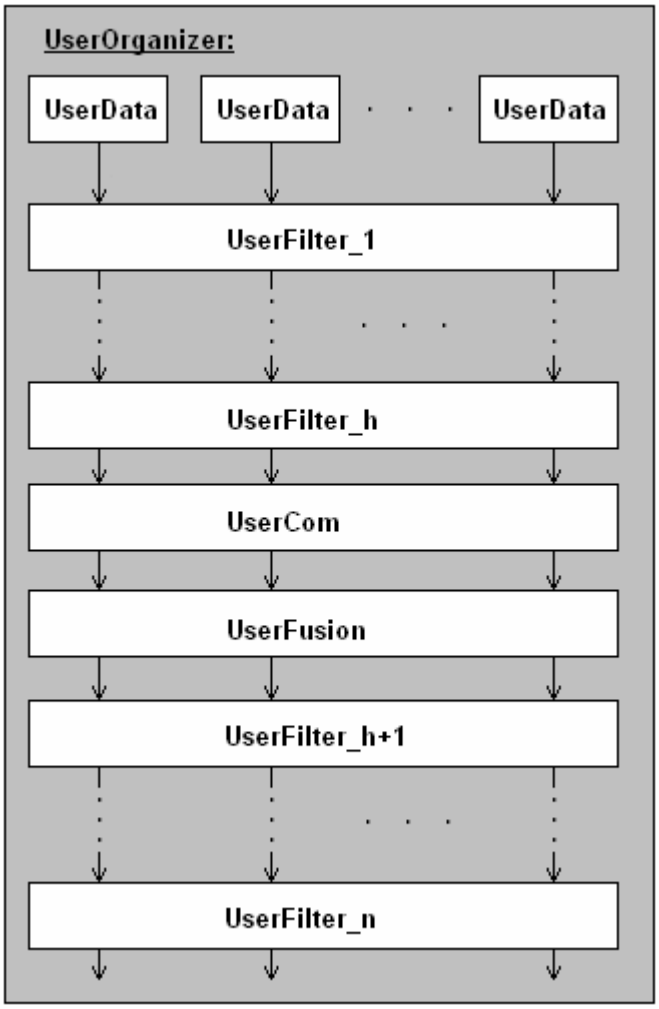

Abbildung 34: Anwenderbeispiel

Die folgenden Schritte führt der Anwender durch, um das Modul zu implementieren.

1. **Klasse der Anwenderdatenobjekte implementieren:**

   Der Anwender implementiert die Klasse, aus denen er seine Datenobjekte erzeugen will. Er leitet sie aus der Klasse `FFMData` ab und fügt die Funktionalität hinzu, die er für seine Zwecke benötigt.

```
class UserData : public FFMData {

      ... // Anwenderdaten
};
```

2. **Klassen der Anwenderfilterstufen implementieren:**

   Der Anwender erstellt für jede Filterstufe von `UserFilter_1` bis `UserFilter_n` eine Klasse, in der er die Funktionen der einzelnen Filter implementiert. Diese Klassen sind je von der Klasse `ActiveFFMUnit` abgeleitet. Nun muss er noch mindestens ein Funktionspaar anlegen, worin einmal die Hauptfilteraktivität (MP) ausge-

führt wird und zum anderen im Falle einer Exception die Ausnahmebehandlung (EP).

```
class UserFilter_i : public ActiveFFMUnit {
public:
      ...

      int mainpart(); // Mainpart
      int excepart(); // Exceptionpart

      ...
};
```

Wichtig ist noch die Constructor-Funktion. Hier übergibt der Anwender an die Klasse `ActiveFFMUnit` als Parameter eine Nummer, die dem Objekt mitteilt, im wievielten Datenobjekt es die Eingabedaten zu erwarten hat und die Ausgabedaten abzulegen hat. Dabei ist zu unterscheiden, ob die Filterstufe vor oder nach der Fusion ausgeführt wird. Für die Filter `UserFilter_1` bis `UserFilter_h` wird die Nummer `dln` übergeben, die auf jedem System einmalig vergeben ist und im Bereich **0** bis **m-1** liegt. Für alle übrigen Filterstufen wird die Nummer `dln = 0` verwendet. Außerdem muss der Anwender die Funktionspaare mit dazugehörigen E-CETs und WCETs im Constructor bekannt geben.

```
UserFilter_i::UserFilter_i(…) : ActiveFFMUnit(dln) {

      ...

      addFunctionPair(
            (int(FFMUnit::*) (void))(UserFilter_i::mainpart),
            (int(FFMUnit::*) (void))(UserFilter_i::excepart),
             ecet, wcet); // Funktionspaar bekannt machen
}
```

3. Kommunikationsphase des Anwenders implementieren:

Die Kommunikationsphase wird ähnlich wie die Filterstufen erstellt. Sie wird jedoch von der Klasse `FFMCommunication` abgeleitet und dem Constructor der Klasse wird ebenfalls eine Nummer `dln` im Bereich von **0** bis **m-1** übergeben. Außerdem werden weitere Parameter übergeben die für das RGCP Protokoll relevant sind, wie die Anzahl der Teilnehmer **m** und die Adressen der Clients und des Access point AP.

```
class UserCom : public FFMCommunication {
public:
      ...

      int mainpart(); // Mainpart
      int excepart(); // Exceptionpart
      bool requestDate(int dline); // Daten empfangen

      ...
};
```

```
UserCom::UserCom(…) : FFMCommunication(dln, m, AP, CLIENTS) {

      ...

      addFunctionPair(
            (int(FFMCommunication::*)  (void))(UserCom::mainpart),
            (int(FFMCommunication::*)  (void))(UserCom::excepart),
             ecet, wcet); // Funktionspaar bekannt machen
}
```

Da die Daten periodisch versandt werden, wird die Sendefunktion innerhalb der Funktion `mainpart()` aufgerufen. Für den Aufruf der Empfangsfunktion ist die virtuelle Methode `requestData()` der Klasse `FFMUnit` vorgesehen, die von der nachfolgenden Phase aufgerufen wird, was im Fall der Kommunikation die Fusion ist. Diese Methode wird hier vom Anwender überladen und implementiert. Die Ausführungszeit sollte daher mit in die ECET der Fusionsphase einbezogen werden.

```
int UserCom::mainpart() {
      /* lokale Sensordaten) */
      UserData* data = (UserData*)(getDataObject());
      unsigned char* db = ... // Daten versandtbereit machen

      ...

      send(db, db_size, data); // Sende Daten
      return 0;
}

bool UserCom::requestData(int dline) {
      /* Datenobjekt dline*/
      UserData* data = (UserData*)(getDataObject(dline));
      unsigned char* db;
      /* empfange Daten von System dline*/
      int rec = recieve(db, max_db_size, dline, data);

      ... // Daten umwandeln und im Datenobjekt speichern
}
```

4. Fusionsstufe des Anwenders implementieren:

Der Anwender erstellt die Fusionsstufe ähnlich wie die Filterstufen. Nur leitet er die Fusionsklasse von der Klasse `FFMFusion` ab und übergibt im Constructor die Anzahl der Teilnehmer **m**.

```
class UserFusion : public FFMFusion {
public:
      ...

      int mainpart(); // Mainpart
      int excepart(); // Exceptionpart

      ...

};
```

```
UserFusion::UserFusion(…) : FFMFusion(m) {

      ...

      addFunctionPair(
            (int(FFMFusion::*) (void))(UserFusion::mainpart),
            (int(FFMFusion::*) (void))(UserFusion::excepart),
             ecet, wcet); // Funktionspaar bekannt machen
}
```

5. Implementierung des Organizer Objektes:

Der Anwender erstellt eine Klasse `UserOrganizer`, die von der Klasse `FFMOrganizer` abgeleitet ist, und fügt gegebenenfalls eigene Funktionalität hinzu.

```
class UserOrganizer : public FFMOrganizer {

      ... // eventuelle Anwenderfunktionalität
};
```

Im Constructor dieser Klasse muss der Anwender nun **m** Datenobjekte erzeugen und anmelden. Anschließend erzeugt und meldet er alle Bearbeitungsphasen, also Filter-, Fusionsstufen und die Kommunikation, in der richtigen Reihenfolge an. Wenn die globale Koordinierung der Phasen verwendet werden soll, so wird sie hier aktiviert. Am Ende gibt er den Befehl zum Start der Verarbeitung mit der Übergabe der Periodendauer. Den Destructor dagegen verwendet er, um die Ausführung zu stoppen und für eventuell weiteren Aufräumarbeiten.

```
UserOrganizer::UserOrganizer(…) : FFMOrganizer() {

      ...

      setDataObject(new UserData(), 0); // Datenobjekte anmelden
      ... // Datenobjekte anmelden
      setDataObject(new UserData(), m-1); // Datenobj. Anmelden

      addFFMUnit(new UserFilter_1(…)); // Filter anmelden
      ... // Filter anmelden
      addFFMUnit(new UserFilter_h(…)); // Filter anmelden
      addFFMUnit(new UserCommunication(…)); // Komm. anmelden
      addFFMUnit(new UserFusion(…)); // Fusion anmelden
      addFFMUnit(new UserFilter_h+1(…)); // Filter anmelden
      ... // Filter anmelden
      addFFMUnit(new UserFilter_n(…)); // Filter anmelden

      usePhaseSync(true); // globale Koordinierung aktivieren
      startProcessing(period); // Verarbeitung starten
}
```

```
UserOrganizer::~UserOrganizer() {
      stopProcessing(); // Verarbeitung anhalten
      clearFFMUnits(); // Verarbeitungskette löschen
      clearData(); // Datenobjekte löschen

      ...
}
```

6. Organizer Objekt im Modul erzeugen:

Der Anwender erzeugt das Organizer Objekt in der Initialisierungsfunktion des Moduls und löscht es in der Aufräumfunktion.

```
int init_module() {

      ...

      UserOrganizer* organizer = new UserOrganizer(…);
}

void cleanup_module() {
      delete organizer;

      ...
}
```

7. Anwendermodul erzeugen und im Kernel einbinden:

Der Anwender übersetzt den Quellcode in ein ausführbares Modul mit einem geeigneten C++ Compiler. Im Rahmen dieser Arbeit wurde der g++ Compiler der gcc lib 2.95 verwendet. Das Modul muss dann auf allen Systemen gestartet werden, durch Einbindung in den RTLinux Kernel. Dazu muss auch der AP des RGCP gestartet werden, damit die Kommunikation erfolgen kann.

```
insmod module.o
```

6.2 Umsetzung des Beispiels mit Hilfe der FFM-Klassen

Um die Funktionalität und das zeitliche Verhalten der Lösungen, die in dieser Arbeit entwickelt wurden, überprüfen und testen zu können, wurde die mehrstufige, verteilte Sensorfusion von Laserscandaten hier mit Hilfe der FFM-Klassen implementiert. Das Ziel dabei ist, in dieser Anwendung die zeitlichen Garantien bereitzustellen und die Fusion von zeitlich kohärenten Sensordaten in einer dynamischen Umwelt zu gewährleisten. Dabei werden die originalen Fusions- und Filteralgorithmen größtenteils unverändert aus der ursprünglichen Implementierung übernommen. Was getan wurde, ist lediglich eine Umstrukturierung des Quellcodes. Die Ausnahme ist die Elementfusion, da sie hier das Kalman-Filtering Verfahren zur Fusion anwenden soll, wurde sie entsprechend neu umgesetzt. Die Implementierung der mehrstufigen, verteilten Sensorfusion von Laserscandaten wurde nach dem Muster durchgeführt, wie es im vorherigen Abschnitt Schritt für Schritt erklärt wurde. In Abbildung 35 ist das Klassendiagramm

der hier neu umgesetzten mehrstufigen, verteilten Sensorfusion von Laserscandaten dargestellt.

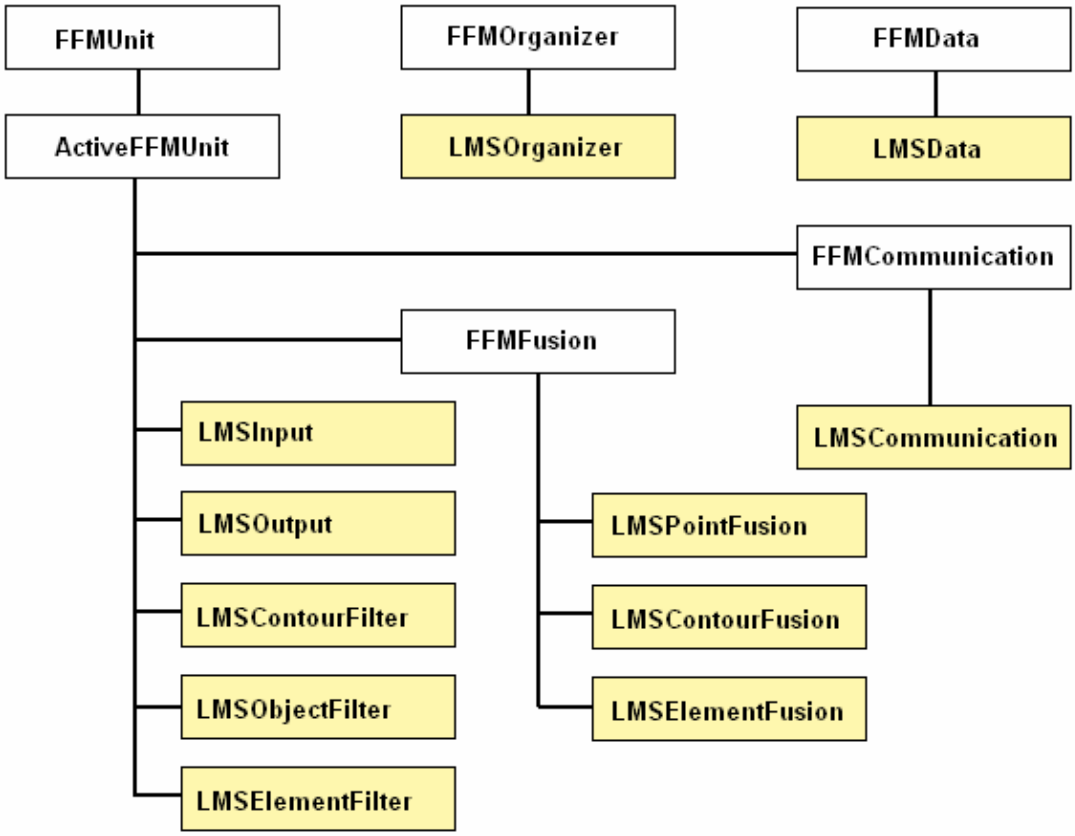

Abbildung 35: Klassen der Lasersensordatenverarbeitung

Das Datenobjekt:

Die Datenobjekte werden aus der Klasse `LMSData` erzeugt. In der neuen Implementierung wird sie aus der Klasse `FFMData` abgeleitet und erbt damit die Zeitstempelmethoden.

Die Verarbeitungsphasen und Filterstufen:

Die Filterstufen der mehrstufigen, verteilten Sensorfusion von Laserscandaten sind vorher alle in der Klasse `LMSFilter` implementiert worden. Sie wurden während der Datenverarbeitung innerhalb desselben Threads nacheinander ausgeführt. In der neuen Implementierung soll, gemäß den in dieser Arbeit entwickelten Lösungen, die Verarbeitung in Phasen durch Verwendung mehrerer Threads aufgeteilt werden, um zeitliche Garantien über die Ausführung der einzelnen Phasen und der Rechtzeitigkeit geben zu können. Daher wurden die Algorithmen der `LMSFilter` Klasse nach Zugehörigkeit zu den Filterstufen sortiert und in getrennten Klassen implementiert. Die Kontur-, Objekt- und Elementfilter sind in den Klassen `LMSContourFilter`, `LMSObjectFilter` und `LMSElementFilter` umgesetzt. Sie sind alle direkt von der Klasse `ActiveFFMUnit` abgeleitet und enthalten je ein Funktionspaar(Mainpart und Exceptionpart). Die Ausnah-

me ist hier der Konturfilter. Er wurde in zwei von einander unabhängige Verarbeitungsschritte aufgeteilt, in Bogensuche und Kantenextraktion. Daher existieren in der Klasse `LMSContourFilter` dem entsprechend zwei Funktionspaare, und die beiden Aktionen werden damit unabhängig voneinander im selben Zeitfenster ausgeführt.

Zusätzlich zu den Filterstufen wurden noch zwei weitere Verarbeitungsphasen in der neuen Implementierung hinzugefügt. Einmal für die periodische Eingabe von aktuellen Sensordaten und einmal für die Ausgabe der Ergebnisdaten über eine RT-FIFO, so dass sie darüber von den Anwenderapplikationen oder der Kontrollapplikation ausgelesen werden können. Die Kontrollapplikation wurde aus der alten Implementierung übernommen und geringfügig angepasst. Diese beiden Phasen sind in den Klassen `LMSInput` und `LMSOutput` implementiert, die beide ebenfalls direkt von der Klasse `ActiveFFMUnit` abgeleitet sind. Das `LMSInput` Objekt ist für die Kommunikation mit dem Laserscanner über das Laser-Modul verantwortlich und erzeugt auch die Zeitstempel zu den Sensordaten. Das Folgende zeigt die Methode, die den MP implementiert:

```
int LMSInput::mainpart() {
      LMSData* data (LMSData*)(getDataObject());

      hrtime_t ts = get_scan_lms(
                        (char*)(dataref->GetActualDataPointer()),
                        getDataLineNumber(),
                        createTimeStamp()); // neue Daten anfordern

      if(ts != 0) {
            dataref->setFFMTimeStamp(stime); // Zeitstempel setzen
            makeOutputAviable();
      }
}
```

Da die Ausführungszeit kaum variable ist, wird als ECET in dieser Klasse die maximale Ausführungszeit verwendet, womit der EP eigentlich keine Funktion besitzt. Das `LMSOutput` Objekt legt periodisch die Ergebnisdaten in die FIFO. Dabei ist es egal, welches Format sie besitzen. Es ist daher Möglich, zu Testzwecken die Daten auch in einem Format (Punkte, Konturen, Objekte) für die Kontrollapplikation, durch gezieltes Überspringen der letzten Verarbeitungsphasen, auszugeben, was nicht den Endverarbeitungsstand der Daten repräsentiert. Wie auch im `LMSInput` Objekt, wird hier ebenfalls die maximale Laufzeit als ECET verwendet.

Die Kommunikationsphase:

Die Kommunikationsphase ist in der Klasse `LMSCommunication` implementiert, die von der Klasse `FFMCommunication` abgeleitet ist. Hier werden die lokalen Daten periodisch im MP `mainpart()` versandt. Da die Daten sich bereits in einer flachen Datenstruktur befinden, können sie unverändert aus dem Buffer des Datenobjekts verschickt werden. Der Empfang wird in der überladenen Methode `requestData(int dln)` der `FFMUnit` Klasse ausgeführt und die Daten werden in den Buffer des entsprechenden Datenobjekts kopiert. Für die ECET wird hier eine Zeit verwendet, die groß genug ist, so dass

die lokalen Daten versandt werden und alle Daten der anderen Systeme empfangen werden können, bevor die Deadline erreicht wird und die Fusionsphase beginnt.

Die Fusionsstufen:

Wie auch bei den Filterstufen wurden die Fusionsstufen in je einzelnen Klassen implementiert. Die Punktfusion, Kantenfusion und Elementfusion sind in den Klassen `LMSPointFusion`, `LMSContourFusion` und `LMSElementFusion` umgesetzt. Auf die Implementierung der Objektfusion wurde verzichtet, da sie sich in früheren Messungen als wenig effizient erwies. Alle Klassen sind direkt von der Klasse `FFMFusion` abgeleitet. Tabelle 2 listet die möglichen Verarbeitungsketten bei unterschiedlich gewählten Fusionsstufen auf.

Phasen	Punktfusion	Konturfusion	Elementfusion
1.	`LMSInput`	`LMSInput`	`LMSInput`
2.	`LMSCommunication`	`LMSContourFilter`	`LMSContourFilter`
3.	`LMSPointFusion`	`LMSCommunication`	`LMSObjectFilter`
4.	`LMSContourFilter`	`LMSContourFusion`	`LMSElementFilter`
5.	`LMSObjectFilter`	`LMSObjectFilter`	`LMSCommunication`
6.	`LMSElementFilter`	`LMSElementFilter`	`LMSElementFusion`
7.	`LMSElementFusion`	`LMSElementFusion`	`LMSOutput`
8.	`LMSOutput`	`LMSOutput`	

Tabelle 2: Verarbeitungsphasen bei unterschiedlichen Fusionsstufen

In der Klasse `LMSElementFusion` wurde die Elementfusion neu umgesetzt, wie es im fünften Kapitel beschrieben wurde. Unabhängig von der Wahl der Fusionsstufe ist die Elementfusion immer teil der Verarbeitungskette, denn auch wenn die Daten bereits in einer früheren Fusionsphase fusioniert wurden und daher nur von einem Datenobjekt stammen, so findet dennoch eine Fusion mit den Daten der Vergangenheit statt. In der Elementfusion werden außerdem zusätzliche Informationen, und zwar die Geschwindigkeit der Elemente, extrahiert, womit sie auch die Aufgabe eines Filters erfüllt. Sie überführt damit das Datenformat „Elemente“ in das Format „dynamische Elemente“. Die Positionsverfolgung der Elemente mittels Kalman-Filtering Verfahrens wurde in der Elementfusion umgesetzt. Für die Positionsverfolgung muss bekannt sein, wie viele Elemente sich in der zu observierenden Umgebung befinden. Diese Information (Anzahl der Bälle `ballcnt` und Roboter `robotcnt`) wird im Constructor der Klasse übergeben. Dazu wird die Anzahl der Teilnehmer `dlncnt` der verteilten Verarbeitung über-

geben, wenn die Daten im Elementformat fusioniert werden sollen. Wenn die Daten auf einer niedrigeren Stufe fusioniert werden, dann wird `dlncnt=1` übergeben.

```
LMSElementFusion::LMSElementFusion(int dlncnt,
                                   int ballcnt,
                                   int robotcnt);
```

Das Organizer Objekt:

Das Organizer Objekt, was für die Infrastruktur der Verarbeitungsphasen verantwortlich ist, wurde in der Klasse `LMSOrganizer` implementiert und diese ist von der Klasse `FFMOrganizer` abgeleitet. Im Constructor werden die Phasen und Filterstufen angemeldet, je nachdem, welche Fusionsstufe gewählt wurde. Ihm werden benötigte Parameter übergeben, wie die Anzahl der Teilnehmer `lcnt`, der wievielte Teilnehmer `lsc` das lokale System von `lcnt` Teilnehmern ist, das Fusionslevel `fsl` und die Anzahl der zu observierenden Elemente in der Umgebung. Das Fusionslevel `fsl` wird mit einem Wert von 1 bis 4 angegeben, wobei 1 die Punktfusion ist und 4 die Elementfusion.

```
LMSOrganizer::LMSOrganizer(int lcnt, int lsc, int fsl, int ballcnt,
int robotcnt);
```

Die Kontrollapplikation hat die Möglichkeit, mit dem Organizer Objekt über eine FIFO zu kommunizieren, um die Ausgabe unterschiedlicher Datenformate einzustellen, indem einige der letzten Phasen übersprungen werden. Eine weitere Aufgabe ist die Einstellung der Positionen von den Laserscannern der einzelnen teilnehmenden Systeme. Die Positionen sind für die Fusionsstufen relevant und werden daher an diese vom Organizer Objekt weitergegeben.

```
void LMSOrganizer::SetPosition(int laser, POINT pos, double angle);
```

Das Modul:

Das `LMSOrganzer` Objekt wird im Modul `FilterFusionModule.o` erzeugt. Das Modul wird mit dem `g++` Compiler erstellt und gestartet wird es, indem das Modul in den Kernel dynamisch eingebunden wird und die erforderlichen Parameter übergeben werden. Beispiel:

```
insmod FilerFusionModule.o sourcecount=2 lasersource=1 fusionlevel=2
robotcount=3 ballcnt=1
```

Dieses Modul ist von anderen Modulen abhängig, die aus diesem Grund vorher schon eingebunden sein müssen. Es ist von folgenden Modulen in Tabelle 3 abhängig.

Modul	Funktion
`LaserModule.o`	Modul zur Kommunikation mit den Laserscannern
`rt_com.o`	Verbindung zu den Laserscannern über serielle Schnittstelle
`cfplib.o`	Kommunikation über RGCP
`rt_orinoco.o`	Treiber für die WLAN Karte mit Uhrensynchronisation
`rtl_sched.o`	TAFT-Scheduler mit Erweiterung
`rtl_fifo.o`	FIFOs für die Ausgabe der verarbeiteten Daten

Tabelle 3: Modulabhängigkeiten

In Abbildung 36 ist der Datenaustausch der Module untereinander dargestellt.

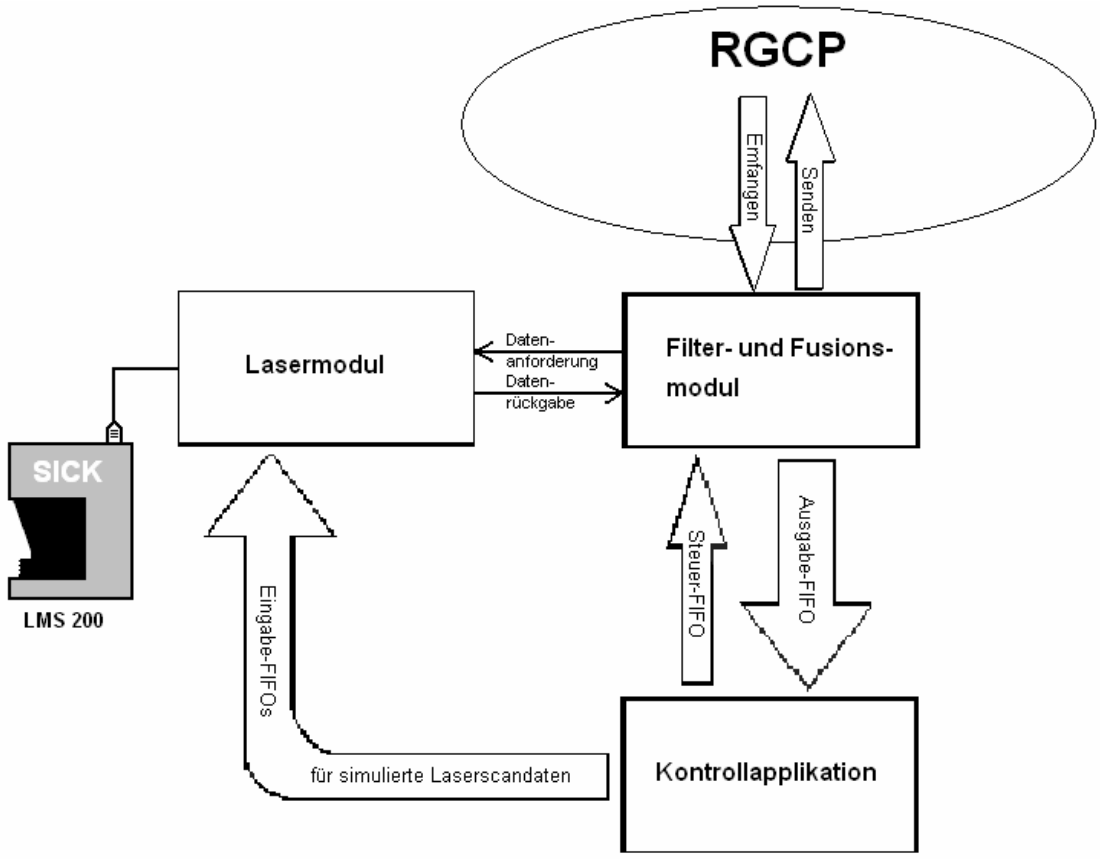

Abbildung 36: Kommunikation zwischen den Modulen

7 Ergebnisse und Messungen

In diesem Kapitel werden nun die Ergebnisse der Implementierung der entwickelten Konzepte mittels Messungen überprüft. Es wird kontrolliert, ob die zeitlichen Garantien, die **Rechtzeitigkeit** und die **Fusion zeitlich kohärenter Sensordaten**, eingehalten werden.

Um die Lösungen testen zu können, wird eine konkrete Anwendung einer verteilten Sensorfusion benötigt, welche die implementierten Lösungen über die FFM-Klassen einbindet. Als Anwendungsbeispiel dient daher die Neuumsetzung der mehrstufigen, verteilten Sensorfusion von Laserscandaten, wie sie im Kapitel 6 vorgestellt wurde. An ihr wurden die folgenden Messungen vorgenommen:

1. **Messungen zur Rechtzeitigkeit und Koordinierung:** Darin wird überprüft, ob die Aktivitäten der Phasen innerhalb ihrer abgesteckten Zeitfenster erfolgt und dass sie rechtzeitig vor ihren Deadlines beendet werden, entweder regulär durch den Abschluss ihrer Berechnungen oder durch den Abbruch, seitens des TAFT-Schedulers und der vollständigen Ausführung des EPs. Es wurden hier stets die Zeiten der globalen, synchronisierten Uhr auf allen beteiligten Stationen aufgezeichnet, um die Wirkung der Phasenkoordinierung überprüfen zu können.

2. **Messungen zur zeitlichen Kohärenz der Daten durch Koordinierung:** In dieser Messung wurden die Zeitunterschiede der Zeitstempel zwischen den Daten der Teilnehmer aufgezeichnet, um zu prüfen, ob zeitlich kohärente Daten fusioniert werden. Außerdem werden zu Vergleichszwecken auch die Unterschiede der Zeitstempel bei deaktivierter Koordinierung aufgezeichnet.

3. **Messungen in der Elementfusionsstufe:** Hier soll überprüft werden, ob das Kalman-Filtering Verfahren erfolgreich zeitliche Diskrepanzen der Daten auflösen kann, indem jeweils die gemessenen Positionen eines Elements aller Teilnehmer und der daraus resultierende Pfad des Elements aufgezeichnet wird.

Für die Durchführung der Messungen standen im Labor drei Rechnersysteme zur Verfügung. Die Tabelle 4 listet diese drei Systeme mit den relevanten Merkmalen auf.

Rechnernahme	• euk-lab14	• euk-lab09	• euk-lab04
Hardware	• AMD Athlon 700MHz 128 MB RAM • 2x Serielle Schnittstelle	• AMD Athlon 700MHz 128 MB RAM • 2x Serielle Schnittstelle	• Celeron 450 MHz 128 MB RAM

zusätzl. Hw.	• CardBus bridge PCI/PCMCIA • Orinoco wlan card • SICK LMS 200 Laserscanner an Com 1	• CardBus bridge PCI/PCMCIA • Orinoco wlan card • SICK LMS 200 Laserscanner an Com 2	• CardBus bridge PCI/PCMCIA • Orinoco wlan Karte
Software	• Redhat Linux 7.2 • gcc 2.95 • Kernel 2.4.4-rtl patched • RTLinux 3.1 • TAFT-Scheduler Version 2 mit Erweiterungen • Treiber: rt_orinoco-timesync-0.9. • Treiber: rt_com 1.1 • RGCP 2.6.4. • Sensorfusions-applikation der Laserscanner • Kontrollapplikation	• Redhat Linux 7.2 • gcc 2.95 • Kernel 2.4.4-rtl patched • RTLinux 3.1 • TAFT-Scheduler Version 2 mit Erweiterungen • Treiber: rt_orinoco-timesync-0.9. • Treiber: rt_com 1.1 • RGCP 2.6.4. • Sensorfusions-applikation der Laserscanner • Kontrollapplikation	• Suse Linux 7.3 • gcc 2.95.3 • Kernel 2.4.4-rtl patched • RTLinux 3.1 • Treiber: rt_orinoco-timesync-0.9. • RGCP 2.6.4.
Aufgaben	• 1. Teilnehmer der verteilten Datenverarbeitung • Datenverarbeitung von Laserscanndaten	• 2. Teilnehmer der verteilten Datenverarbeitung • Datenverarbeitung von Laserscanndaten	• AP im RGCP Protokoll • Indikator der Uhrensynchronisation

	• 1. Teilnehmer im RGCP Protokoll • Master der Uhrensynchronisation	• 2. Teilnehmer im RGCP Protokoll • Slave der Uhrensynchronisation	

Tabelle 4: Systemressourcen für die Messungen

Für die Erzeugung von Sensordaten existieren zwei Optionen. Es können vorab gespeicherte Daten von der Kontrollapplikation über eine FIFO an das Datenverarbeitungsmodul gesendet werden. Die zweite Möglichkeit ist die Verwendung der realen Laserscanner. Dafür wurden Szenarios aus Kisten, Brettern usw. aufgebaut, welche Situationen des Robocup nachahmen oder andere Situationen, denen die Sensoren möglicherweise ausgesetzt sein könnten. Abbildung 37 zeigt ein Beispiel eines aufgebauten Szenarios.

Abbildung 37: Beispiel: Setup eines Szenarios im Labor

Die Daten der Laserscanner werden durch mehrere Phasen verarbeitet und durchlaufen dabei mehrere Abstraktionsstufen. Die Abstraktionsstufen sind die Punkte, Konturen, Objekte, Elemente und die dynamischen Elemente. In Abbildung 38 wurde jede Abstraktionsstufe einmal von der Kontrollapplikation ausgegeben und hier abgebildet. Es wird die Ausgabe bei der Punktfusion als gewählte Fusionsstufe gezeigt.

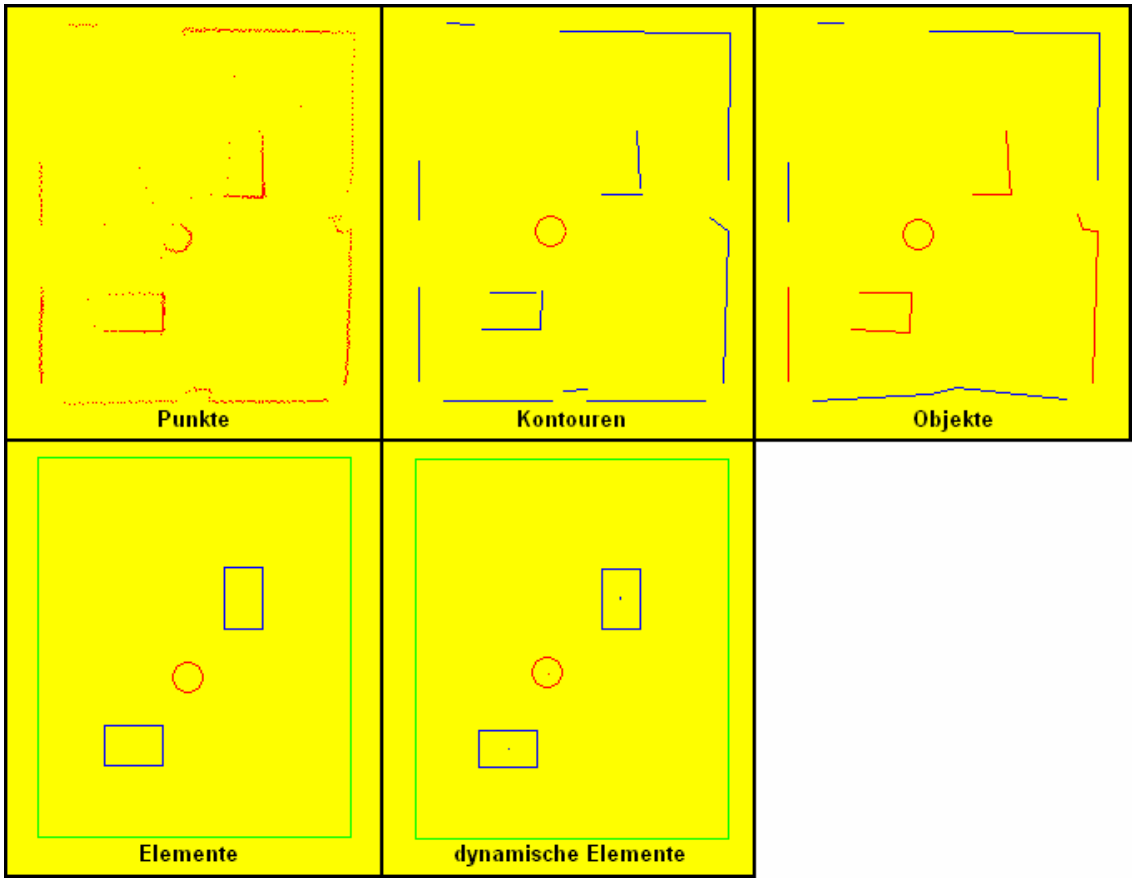

Abbildung 38: Abstraktionsstufen der Sensordaten

Für die erste Messung wurden gespeicherte Daten verwendet, da die Verarbeitung mit einer Periodendauer von 100 ms durchgeführt wurde, die Verwendung der realen Laserscanner jedoch nur ca. 3 Scanndaten pro Sekunde liefern kann, da auf den verwendeten Systemen nur eine maximale Baudrate von 38400 zur Kommunikation über die serielle Schnittstelle mit den Laserscannern eingestellt werden konnte. Es wurden die gleichen Daten verwendet, die auch in den früheren Zeitmessungen am Bogenfilter [18] genutzt worden waren. Es handelt sich dabei um ein Szenario (Abbildung 39), bei dem der Laserscanner durch eine Umgebung bewegt wird, die aus einer Begrenzung, drei rechteckigen Objekten und einem kreisförmigen Objekt besteht. Durch die Bewegung wird erreicht, dass die Wahrnehmung von unterschiedlichen Perspektiven erfolgt, womit die resultierenden Berechnungen der Phasen variable Laufzeiten aufweisen.

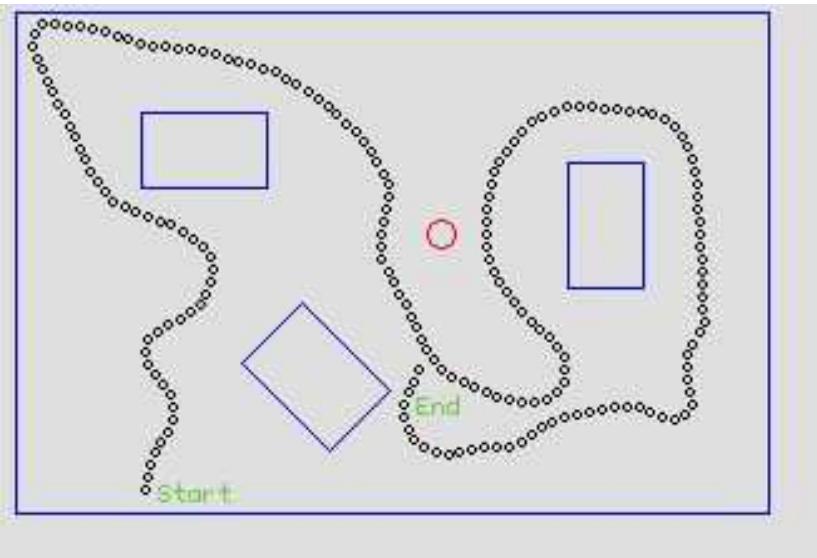

Abbildung 39: Szenario: bewegter Laserscanner

Diese Daten wurden in je 200 ms Takten an beide Stationen gesendet, welche sie mit einer Periodendauer von 100 ms verarbeiteten und auf der Abstraktionsebene der Konturen die Daten übertrugen und fusionierten. In Tabelle 5 sind die Phasen in der Reihenfolge aufgelistet, in der sie pro Periode ausgeführt werden. Dazu sind je die ECETs der MPs und die WCETs der EPs aufgeführt. Die Deadlines wurden mit Hilfe der ECETs und WCETs vom Organizer Objekt bestimmt, der damit die Periodenzeit von 100 ms unter den Phasen aufteilt.

Phasen	**ECET des MP**	**WCET des EP**	**Deadline**	**Deadline rel. zur Periodenstartzeit**
LMSInput	4 ms	0,1 ms	4.269.925 ns	4.269.925 ns
LMSContourFilter	30 ms + 4 ms	2**x** 0,2 ms	35.825.717 ns	40.095.642 ns
LMSCommunication	35 ms	0,1 ms	36.554.792 ns	76.650.434 ns
LMSContourFusion	4 ms	0,1 ms	4.269.926 ns	80.920.360 ns
LMSObjectFilter	4 ms	0,1 ms	4.269.926 ns	85.190.286 ns
LMSElementFilter	4 ms	0,1 ms	4.269.925 ns	89.460.211 ns
LMSElementFusion	4 ms	0,1 ms	4.269.926 ns	93.730.137 ns
LMSOutput	4 ms	0,1 ms	4.269.926 ns	98 ms
FFMSchedSync	nicht in den Messungen enthalten		immer 2 ms unabhängig von den Parametern	100 ms

Tabelle 5: ECET,WCET und resultierende Deadlines

Während der Datenverarbeitung wurden alle signifikanten Ereignisse zeitlich auf beiden Stationen festgehalten. Dies wurde über 400 Perioden lang getan. Die signifikanten Ereignisse waren der Beginn der Berechnung einer Phase und der Abschluss durch die Ausführung der Funktion `lastAction()`. Diese Funktion kann dabei im Kontext des MPs ausgeführt werden, oder auch im Kontext des EPs, wenn der Thread der Phase vom TAFT-Scheduler abgebrochen werden musste, um ein überschreiten der Deadline zu verhindern. Bei der Kommunikationsphase wurde das etwas anders gehandhabt. Hier wurde als Ende der Phasenaktivität der Zeitpunkt angesehen, ab dem die Daten der aktuellen Periode beider Teilnehmer lokal verfügbar waren. Also die Zeit, an der das letzte erwartete Datenpaket empfangen wurde. Die Diagramme in Abbildung 40 und Abbildung 41 stellen die Ergebnisse der Messungen von beiden Stationen dar. Sie zeigen die zeitliche Abfolge der Aktivitäten jeder Periode. Um diese zeitliche Abfolge einer bestimmten Periode abzulesen, wird beim Wert 0 der y-Achse auf der entsprechenden Stelle der x-Achse begonnen. Von dort werden die Aktivitäten der Phasen durch eine Bewegung, parallel zu y-Achse, nach oben durchlaufen. Die einzelnen Phasenaktivitäten sind in unterschiedlichen Farben dargestellt. Die weißen Bereiche stellen die Zeiten dar, in denen keine Phasenaktivitäten stattfanden. Alle Zeiten wurden mit der globalen, synchronisierten Uhr gemessen, womit sie zwischen beiden Stationen verglichen werden können.

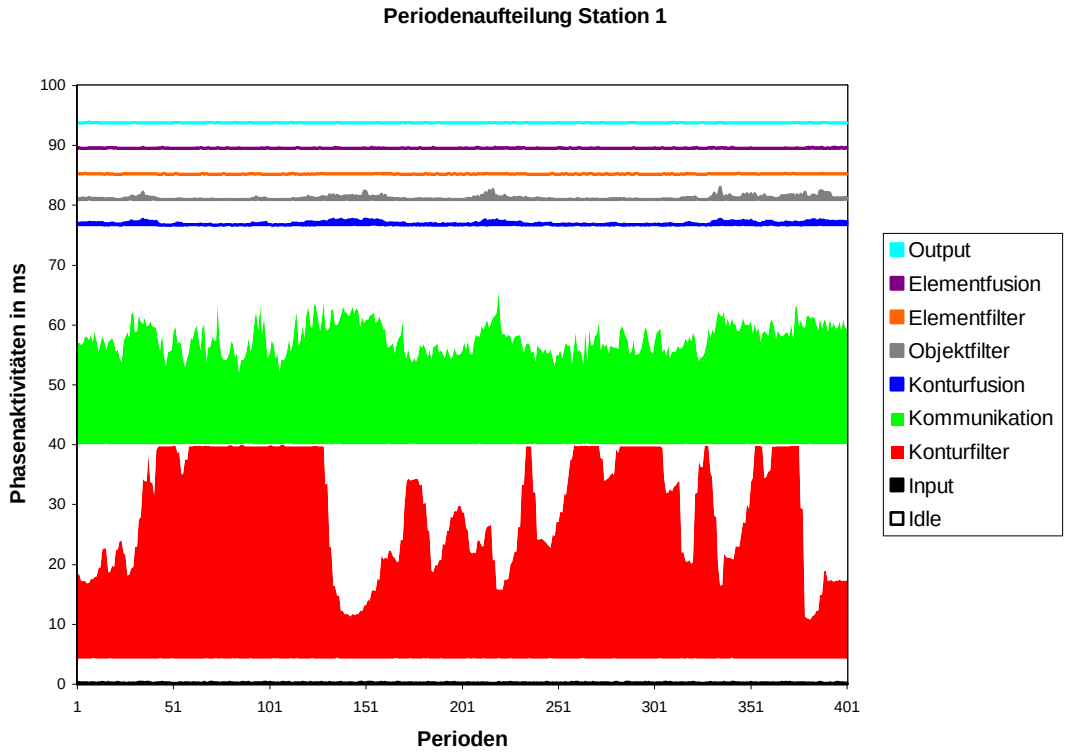

Abbildung 40: Phasenaktivitäten der 1. Station

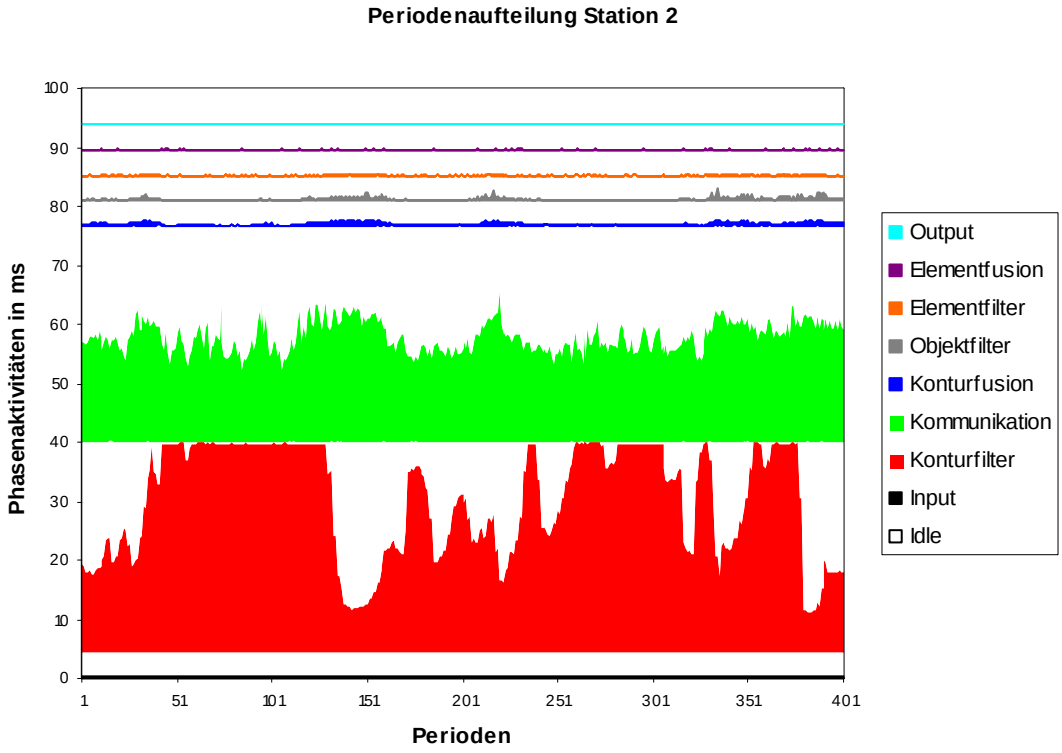

Abbildung 41: Phasenaktivitäten der 2. Station

Die Messwerte ähneln sich auf den beiden Stationen, was daran liegt, dass auf beiden die gleichen vorab gespeicherten Laserscanndaten verarbeitet wurden. Ansonsten würden sich die Zeiten der Phasen, die die lokalen Daten verarbeiten, in diesem Fall der Konturfilter, in der Regel unterscheiden. Beim Überprüfen der Messwerte mit Hilfe der Deadlines aus Tabelle 5 ist zu erkennen, dass alle Phasenaktivitäten in ihren vorbestimmten Zeitfenstern ablaufen und ihre Deadlines einhalten. Diese werden ebenso eingehalten, wenn die Berechnungen der Phasen nicht regulär beendet wurden, sondern durch Ausführung des EPs. Während der Messungen wurden auf der ersten Station 121 Exceptions im Konturfilter ausgelöst und 123-mal im Konturfilter der zweiten Station. In den Diagrammen sind sie dort zuerkennen, wo die Datenreihe des Konturfilters knapp unter der Datenreihe der Kommunikationsphase scheinbar abgeschnitten wurden. Da auf beiden Stationen die globale, synchronisierte Uhr zur Messung verwendet wurde und diese auch die Grundlage für die Koordinierung der Periodenstartzeiten ist, kann die Aussage gegeben werden, dass sich die Zeitfenster gleicher Phasen auf den beiden Stationen auch zeitlich überdecken, womit auch die Koordinierung korrekt arbeitet.

Ziel der Koordinierung der Phasen ist es, dass zum Beginn der Fusionsphase stets zeitlich kohärente Daten zur Verarbeitung bereitstehen, da die Fusion von zeitlich inkohärenten Daten zu ungenauen oder fehlerhaften Resultaten führen kann. Abbildung 42 zeigt am Beispiel der mehrstufigen, verteilten Sensorfusion von Laserscandaten die Folgen einer Fusion zeitlich inkohärenter Daten von zwei Stationen bei der Punktfusion und der Konturfusion. Es wurde in allen Fällen genau ein Objekt beobachtet, dass sich relativ schnell in der Umgebung bewegt. Auf der linken Seite sind die Resultate der

unkoordinierten Verarbeitung und Fusion zu sehen. Scheinbar existieren je zwei Objekte in der Umgebung, was aber nicht der Realität entspricht. Rechts sind zum Vergleich die Resultate der koordinierten Verarbeitung und Fusion abgebildet.

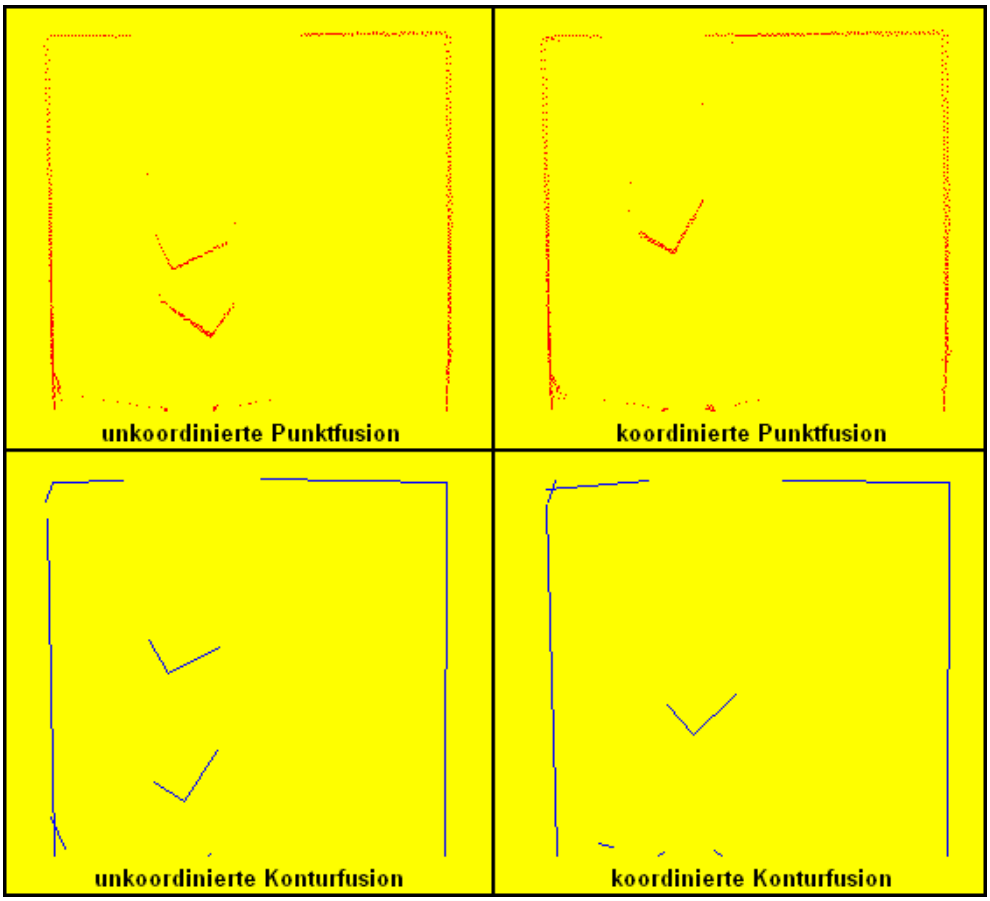

Abbildung 42: Beispiel: Fusion inkohärenter und kohärenter Sensordaten

Die rechtzeitige Bereitstellung zeitlich kohärenter Sensordaten wird erreicht, durch die gleichzeitige Aktivierung der Phase, die für die Erzeugung der Daten mittels der Sensoren verantwortlich ist und der gleichzeitigen Aktivierung der Kommunikationsphase, deren Deadline dem Beginn der Fusionsphase entspricht. Mit der zweiten Messung wurde überprüft, ob die Koordinierung ihr Ziel erreicht. Zu diesem Zweck wurden während der Datenverarbeitung vor der Fusionsphase die Differenzen der Zeitstempel der Sensordaten (Abbildung 43) von beiden Stationen über mehrere Perioden aufgezeichnet. Zu Vergleichszwecken wurde das einmal mit koordinierten Phasen und einmale mit deaktivierter Koordinierung getan. Bei der unkoordinierten Datenverarbeitung wurde lediglich einmalig eine gleiche Startzeit für die erste Periode vereinbart. Anschließend lief die Verarbeitung auf beiden Stationen voneinander unabhängig ab. Es wurde also keine periodische Korrektur der Periodenstartzeiten durchgeführt. Für beide Messungen wurde eine Periodendauer von 500 ms gewählt und es wurden jeweils die Differenzen von ca. 2200 Perioden aufgezeichnet.

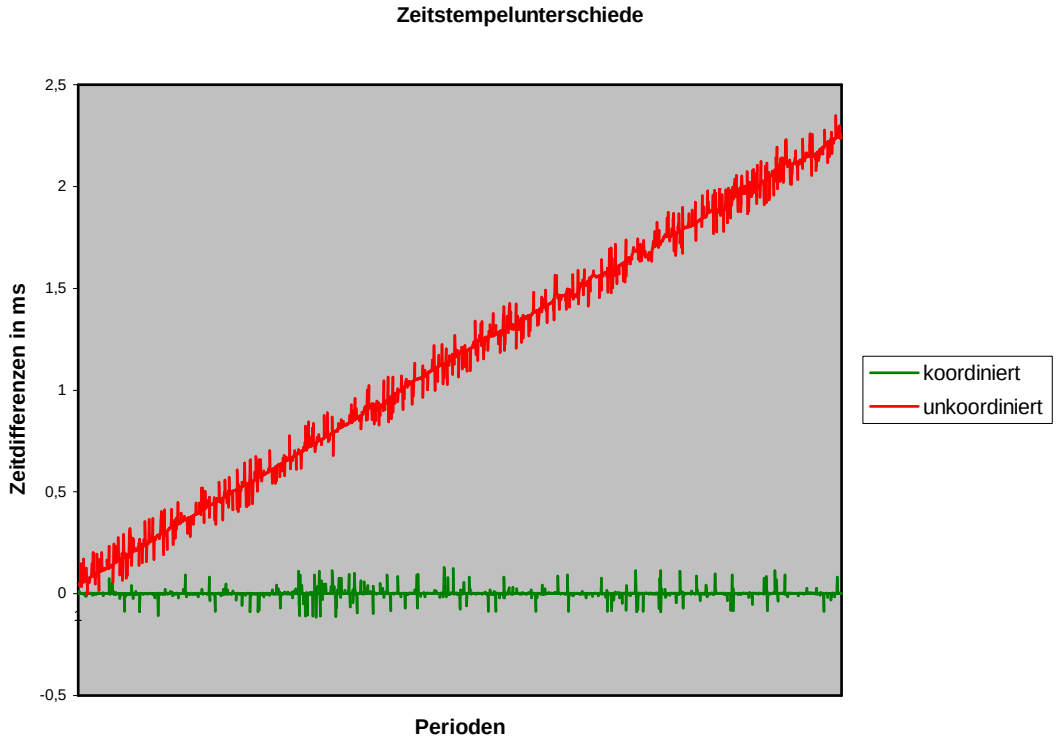

Abbildung 43: Zeitstempelunterschiede der Sensordaten

Bei der Überprüfung der Messwerte wurde im koordinierten Fall ein maximaler Betrag der Abweichung von 128170 ns festgestellt. Im unkoordinierten Fall dagegen war der maximale Betrag der Abweichung 2,35 ms. Im Diagramm der Abbildung 43 ist zu sehen, dass die Differenzen der koordinierten Datenverarbeitung nahe um die x-Achse, also den Wert 0 ms auf der y-Achse, schwanken. Das zeigt, dass die periodische Korrektur der Periodenzeiten erfolgreich das auseinanderdriften der Zeitstempel verhindern kann. Bei der unkoordinierten Datenverarbeitung schwanken die Werte um eine gedachte Linie, die sich ständig von der x-Achse entfernt. Würde die Messung über einen großen Zeitraum fortgesetzt, so würden auch größere Abweichungen erreicht werden, als hier in dieser Messung. Der Verlauf der Abweichung der unkoordinierten Sensordatenverarbeitung würde dem in Kapitel 5 auf Abbildung 21 ähneln und der maximale Betrag der Zeitstempeldifferenz läge im Bereich von der halben bis zur ganzen Periodendauer, also hier von 250 ms bis 500 ms. Während der koordinierten Datenverarbeitung wird somit eine weit geringere maximale Abweichung erreicht.

In Kapitel 5 wurde die Möglichkeit vorgestellt, auch ohne Koordinierung die Daten zu fusionieren, indem zeitliche Inkohärenzen der Daten durch das Kalman-Filtering Verfahren aufgelöst werden. Das Kalman-Filtering Verfahren wurde in der Elementfusionsstufe der Neuumsetzung der mehrstufigen, verteilten Sensorfusion von Laserscandaten aus Kapitel 6 für die Positionsverfolgung der Elemente implementiert. Die dritte Messung dient dazu, die korrekte Funktionsweise des Kalman-Filtering Verfahrens bei der Positionsverfolgung der Elemente zu überprüfen und sicherzustellen, dass zeitliche Diskrepanzen in den Daten erfolgreich aufgelöst werden. Dazu wurde ein Szenario im

Labor (Abbildung 44) nachgestellt, in dem zwei Laserscanner die Umgebung beobachten, worin sich ein Element annähernd geradlinig bewegt. Die Sensordaten wurden mit einer Periodendauer von 330 ms auf beiden Systemen verarbeitet. Die zeitliche Koordinierung der Verarbeitung war dabei deaktiviert. Außerdem wurde die periodische Datenverarbeitung auf den beiden Systemen zueinander zeitlich um 160 ms versetzt, um gezielt zeitliche Diskrepanzen in den Sensordaten zu erzeugen. Während der Verarbeitung wurden in der Elementfusionsstufe die ermittelten Positionen des Elements durch die Elementfilter jeweils beider Stationen aufgezeichnet. Zudem wurden die resultierenden Positionen, die mittels Kalman-Filter ermittelt wurden, festgehalten. Die gemessenen Positionen und der resultierende Pfad des Elements sind in Abbildung 45 grafisch dargestellt.

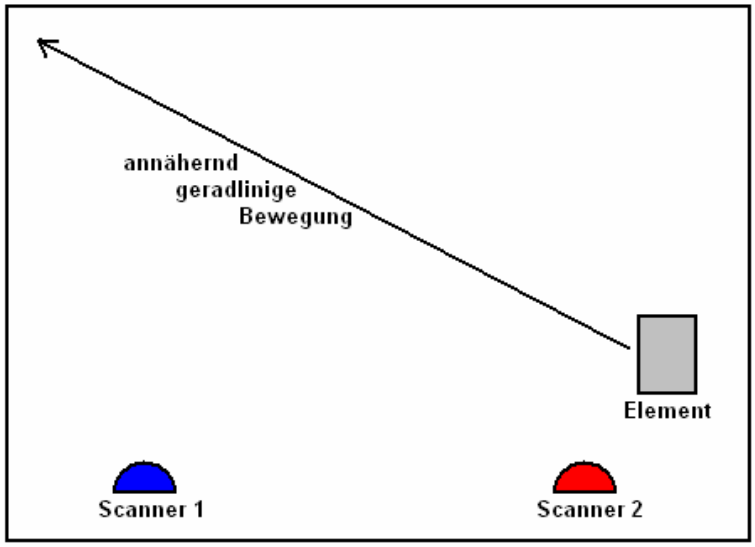

Abbildung 44: Szenario der 3. Messung

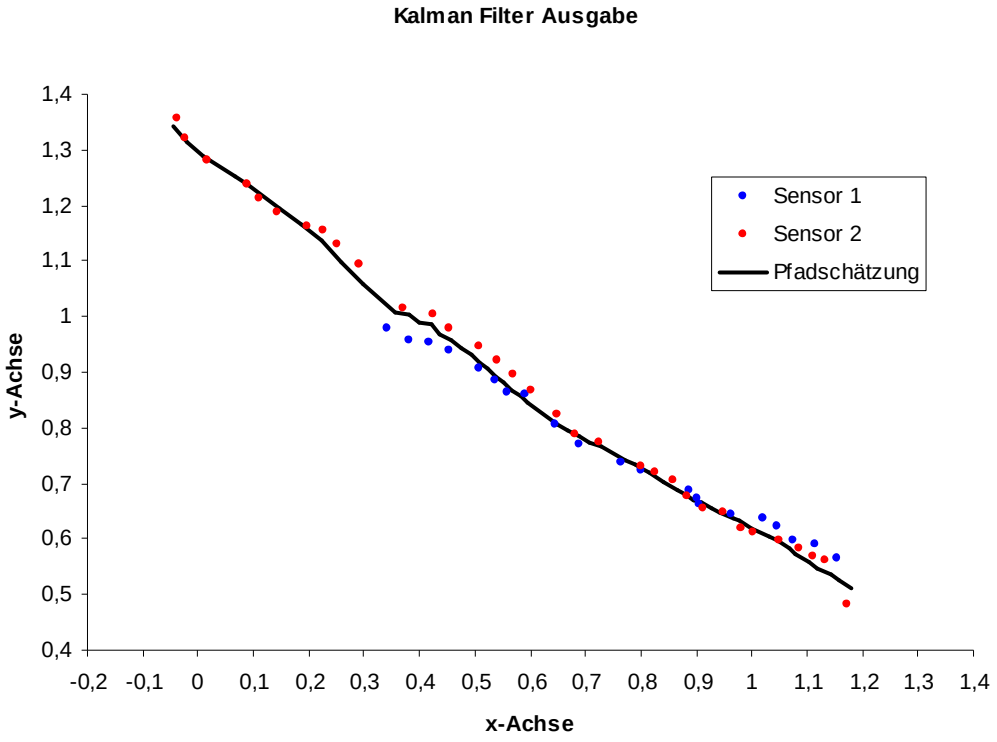

Abbildung 45: Messung der Positionsverfolgung

Zusätzlich zu den Positionen, wurden auch die Zeitstempel der Daten aufgezeichnet, die jedes Mal zu Beginn der Fusionsphasen bereitlagen. Die Unterschiede zwischen den Zeitstempeln beider Datenobjekte lagen stets im Bereich von ca. 160 bis 170 ms, was bedeutet, dass zwischen den Daten starke zeitliche Diskrepanzen herrschten, die schließlich von der Elementfusion aufgelöst werden mussten. Ob und wie gut das gelungen ist, ist in Abbildung 45 zusehen. Die Punkte stellen die Positionen dar, die aus den Wahrnehmungen der Sensoren gewonnen wurden. Die Farbe der Punkte gibt darüber Auskunft, von welchem Teilnehmer die Positionen jeweils erkannt wurden. Der Pfad, der von der Elementfusionsstufe verfolgt wurde, ist als schwarze Linie dargestellt. Es ist zu sehen, dass die ungefähre geradlinige Bewegung des Elements gut nachvollzogen werden konnte, trotz der zeitlichen Diskrepanzen der Daten, was bedeutet, dass diese durch den Kalman-Filter erfolgreich aufgelöst werden konnten. Stellenweise ist zu beobachten, wie abwechselnd die Positionen mal vom ersten und mal vom zweiten Teilnehmer stammen. Durch Wahrnehmungs- und Positionsfehler und durch die unterschiedlichen Perspektiven der Sensoren weichen die extrahierten Positionen von den wahren Positionen meist geringfügig ab. Würde, rein diesen Positionen folgend, der Pfad nachvollzogen, so wäre das Resultat eine wilde Zickzacklinie, was kaum der Realität entspräche. Hier zeigt sich jedoch, dass die Eigenschaft des Kalman-Filters, zufällige Fehler in den Messwerten herauszufiltern, hier dennoch zu einem Resultat einer geraderen Linie als Pfad des Elements führt, was näher an der Wirklichkeit liegt.

Die Messungen konnten die korrekte Funktionsweise der entwickelten Lösungen und deren Implementierung bestätigen. Sie zeigen, dass die Auflagen der Rechtzeitigkeit

der Datenverarbeitung eingehalten wurden. Auch die beiden Möglichkeiten, zeitliche Kohärenz der Sensordaten für die Fusion zu gewährleisten, konnten erfolgreich überprüft werden. Die in dieser Arbeit entwickelten Lösungen erfüllen damit die gestellten Aufgaben aus Kapitel 1.

8 Zusammenfassung und Ausblick

In der vorliegenden Diplomarbeit wurden Lösungen vorgestellt, die sich den Problemen der mehrstufigen, verteilten Sensordatenverarbeitung unter Echtzeitbedingungen widmen. Spezieller, die Datenverarbeitung unter Verwendung des TAFT-Schedulers, welcher zu der Klasse der Task-Pair-Scheduling Verfahren gehört. Die Probleme sind zum ersten, die harten Zeitbeschränkungen der mehrstufigen Verarbeitung und zum zweiten, die zeitlichen Inkohärenzen der Sensordaten bei der verteilten Verarbeitung und der Sensordatenfusion, die durch die Wahrnehmung der verteilten Sensoren in einer dynamischen Umwelt hervorgerufen werden.

Die Verwendung des TAFT-Schedulers als Teil der Ausführungsumgebung des Systems erlaubt es, die Vollführung der Berechnungen auf der Basis von erwarteten Ausführungszeiten, statt der maximal möglichen Ausführungszeiten, zu planen. Das führt zu einer besseren CPU-Auslastung und zu einer höheren Verarbeitungsfrequenz der Sensordaten, wenn die Berechnungen zeitlich stark variieren. Das führt jedoch zu neuen Problemen, wenn in der mehrstufigen Verarbeitung mehrere oder alle Stufen je selbst variable Laufzeiten besitzen, da lange Berechnungszeiten der Verarbeitungsstufen zu dem Resultat führen können, dass beim Erreichen der Deadline die Daten nicht alle Stufen durchlaufen haben und daher nicht das gewünschte Ausgabeformat besitzen. Das Konzept, was in dieser Arbeit entwickelt wurde, löst diese Probleme durch eine Aufteilung jeder Verarbeitungsperiode in Phasen, wobei jeder Phase mindestens ein Thread zugeordnet wird. Jede Phase entspricht je einer Verarbeitungsstufe und die Ausführung ist in jeder Periode je auf ein festes Zeitfenster beschränkt. Somit werden die Rechtzeitigkeit der einzelnen Phasenaktivitäten und die Rechtzeitigkeit der gesamten Verarbeitung garantiert. Dabei wird außerdem abgesichert, dass die Sensordaten von allen Phasen bis zur gesetzten Deadline verarbeitet werden, und dies in der richtigen Reihenfolge. Jeder Phase wird zudem auch je ein Minimum an Ausführungszeit garantiert.

Für das Problem der zeitlichen Inkohärenzen der Sensordaten wurden zwei Lösungsmöglichkeiten betrachtet. Die erste Möglichkeit ist die Auflösung zeitlicher Diskrepanzen in den Daten durch geeignete Verfahren in den Fusionsstufen. Als Unterstützung dienen dazu globalgültige Zeitstempel der Daten, die von einer synchronisierten Uhr stammen. Als Beispiel wurde das Kalman-Filtering Verfahren in der Elementfusionsstufe der mehrstufigen, verteilten Sensorfusion von Laserscandaten umgesetzt. Die zweite Möglichkeit ist die Koordinierung der entscheidenden Phasenaktivitäten auf allen beteiligten Stationen, wie die Erzeugung der Sensordaten durch die Wahrnehmung der Sensoren und die Kommunikation, so dass zu Beginn der Fusionsphasen stets die zeitlich kohärente Daten auf allen beteiligten Stationen bereitliegen. Erreicht wird diese Koordinierung durch eine Zeitgesteuerte Strategie, bei der die Perioden- und Phasenstartzeiten zu festen, einheitlichen Zeiten einer globalen, synchronisierten Uhr festgelegt werden. Diese Startzeiten werden auf die lokale Systemuhr des Schedulers abge-

bildet und ihm als Parameter übergeben. Da die lokalen Uhren der einzelnen Systeme technisch bedingt stetig auseinanderdriften, ist eine regelmäßige Korrektur dieser Abbildung notwendig. Das wird durch eine periodische Berechnung eines Korrekturwertes getan, um den anschließend sämtliche Aktivierungszeiten aller Threads im Scheduler verschoben werden. In Zukunft währe es denkbar, eine Implementierung des TAFT-Schedulers umzusetzen, die direkt, auf der globalen, synchronisierten Uhr basierend, arbeitet. Damit könnte auf die periodische Korrektur der Startzeiten verzichtet werden. Solch eine Version eines Schedulers währe außerdem auch allgemein für die Entwicklung von Anwendungen, bei denen zeitliche Koordinierung eine Rolle spielt, auf verteilten Echtzeitsystemen nützlich.

Die in dieser Arbeit entwickelten Konzepte wurden in Form von wieder verwendbaren C++ Klassen umgesetzt. Sie bilden ein Framework für die Implementierung von Anwendungen der mehrstufigen, verteilten Sensordatenverarbeitung mit Hilfe der bereitgestellten Lösungen, indem sie mittels Vererbung eingebunden werden. Im Rahmen dieser Diplomarbeit wurde die mehrstufige, verteilte Sensorfusion von Laserscandaten als Anwendungsbeispiel mit Hilfe dieser Klassen neu umgesetzt. An ihr wurden die entwickelten Lösungen getestet und durch Messungen konnten deren korrekte Funktionsweise nachgewiesen werden. Somit können sie zukünftig als Grundlage für die Implementierung weiterer Anwendungen der mehrstufigen, verteilten Sensordatenverarbeitung dienen, für mobile Roboter oder andere Echtzeitsysteme.

9 Literaturverzeichnis

[01] L. B. Becker, M. Gergeleit: *Execution Environment for Dynamically Scheduling Real-Time Tasks*, 22nd IEEE Real-Time Systems Symposium (RTSS 2001), WIP-Session, London, 2001

[02] M. Gergeleit, E. Nett: *Scheduling Transient Overload with the TAFT Scheduler* , GI / ITG Fachgruppe Betriebssysteme - Herbsttreffen 2002, 7. Nov. 2002, Berlin

[03] Streich, H.: TaskPair-Scheduling: An Approach for Dynamic Real-Time Systems, Int. Journal of Mini & Microcomputers, Vol.17, No.2, 1995

[04] Nett, E. Schemmer, S., *Reliable real-time communication in cooperative mobile applications*, Computers, IEEE Transactions on, 2003

[05] S. Schemmer: Diplomarbeit: *Zuverlässige Echtzeit-Gruppenkommunikation auf einem lokalen Funknetz*, Rheinische Friedrich-Wilhelms-Universität Bonn, 2000

[06] Edgar Nett, Michael Mock, Martin Gergeleit: *Das drathlose Ethernet*, Addison-Wesley, 2001

[07] S. R. Trikaliotis, *"Uhrensynchronisation in einem lokalen Funknetzwerk"*, diploma thesis, University of Bonn, February 2000

[08] M. Mock, R. Frings, E. Nett, S. Trikaliotis: *Continuous Clock Synchronization in Wireless Real-Time Applications*, SRDS 2000

[09] M. Mock, R. Frings, E. Nett, S. Trikaliotis: *Clock Synchronization for Wireless Local Area Networks*, Euromicro-RTS 2000

[10] Wilfried Elmenreich, *„Sensor Fusion in Time-Triggered Systems"*, Dissertation, Technische Universitaet Wien, 2002

[11] Hermann Kopetz, *"Event-Triggered versus Time-Triggered Real-Time Systems"*, Research Report, Technische Universität Wien, 1991

[12] Traian Pop: *Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems*, Linköping University, thesis 2003

[13] R. Bade: Diplomarbeit: *Modifikation einer geeigneten Stereobildverarbeitungsmethode für die Anwendung in einer Echtzeitumgebung*, Otto-von-Guericke-Universität Magdeburg, 2003

[14] T. Rudolph: Diplomarbeit*: Kamerabasierte, echtzeitfähige Objekterkennung in der Teamrobotik*, Otto-von-Guericke-Universität Magdeburg, 2002

[15] Michael Feist: Diplomarbeit: *Mehrstufige, verteilte Sensorfusion von Laserscandaten*, Otto-von-Guericke-Universität Magdeburg, 2002

[16] Nett, E. and S. Schemmer: *Realizing Virtual Sensors by Distributed Multi-Level Sensor Fusion,* In Proc. 2003 International Workshop on Multi-Robot Systems, Washington, D.C, Kluwever Academic Publishers, 2003

[17] M. Gergeleit, L. Buss Becker, E. Nett: *Robust Scheduling in Team-Robotics*, (WPDRTS 2003), in conjunction with IPDPS 2003, April 22-24, 2003, Nizza, Frankreich

[18] Schemmer, S. and E. Nett: *Achieving Reliable and Timely Task Execution in Mobile Embedded Applications* (WORDS 2003F), Capri Island, Italy, 2003

[19] R. Kalman. *"A new approach to linear filtering and prediction problems"*, Transaction of the ASME--Journal of Basic Engineering, 1960

[20] G. Welch, G. Bishop: *An Introduction to the Kalman Filter,* University of North Carolina at Chapel Hill, 1995

[21] T. Schmitt, R. Hanek, M. Beetz, S. Buck, and B. Radig. *„Watch their moves: applying probabilistic multiple object tracking to autonomous robot soccer"* IEEE Trans. on Robotics and Automation, 18(10):--, 2002

[22] T. Schmitt, R. Hanek, M. Beetz, S. Buck, *„Cooperative probabilistic state estimation for vision-based autonomous mobile robots"*, Eighteenth national conference on Artificial intelligence, 2002

[23] J. Neira and J. Tardos, *"Data association in stochastic mapping using the joint compatibility test"* IEEE Transactions on Robotics and Automation, 2001

[24] SICK Lasermesssysteme LMS 200 / LMS 211 / LMS 220 / LMS 221 / LMS 291, Technische Beschreibung

[25] SICK LMS / LMI 400 Definition der Telegramme zwischen Benutzerschnittstelle und LMS- oder LMI 400-System über RS 422/RS 232, Version 4

[26] M. Barabanov, V. Yodaiken,: Real Time Linux, (white papers), http://www.fsmlabs.com/developers/white_papers/, Juni 2003

[27] E. Nett: *Vorlesungsscript – Echtzeitsysteme*, IVS, Otto-von-Guericke-Universität Magdeburg, 2000

[28] S. Zilberstein, *„Using Anytime Algorithms in Intelligent Systems"*, University of Massachusetts, 1996

[29] M. Gergeleit: *Vorlesungsscript – Kooperative Robotersysteme: Kommunikationsmodelle für kooperative Roboter*, IVS, Otto-von-Guericke-Universität Magdeburg, 2002

Selbstständigkeitserklärung

Ich versichere hiermit, die vorliegende Arbeit selbst verfasst, Zitate gekennzeichnet und keine anderen, als die offen gelegten Quellen und Hilfsmittel benutzt zu haben.

Frederic Rossau