Institut für Informatik
der Technischen Universität München
Lehrstuhl Univ.-Prof. Dr. B. Radig

Objektorientierte Daten- und Zeitmodelle für die Echtzeit-Bildfolgenauswertung

Michael Klupsch

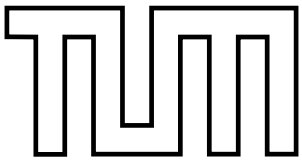
Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

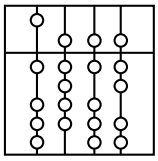
Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Achim Schweikard
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Bernd Radig
2.	Univ.-Prof. Dr. Manfred Paul, emeritiert

Die Dissertation wurde am 24.05.2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 14.12.2000 angenommen.





Fakultät für Informatik
der Technischen Universität München
Lehrstuhl Univ.-Prof. Dr. B. Radig

Objektorientierte Daten- und Zeitmodelle für die Echtzeit-Bildfolgenauswertung

Dissertation

Michael Klupsch

Kurzfassung

In der vorliegenden Arbeit werden neuartige Konzepte für die Modellierung und Repräsentation von *Bild-* und anderen *Sensordatenfolgen* sowie von Funktionen, die diese Datenfolgen verarbeiten, vorgestellt und mit *objektorientierten Methoden* realisiert. Es werden Mechanismen entwickelt, mit denen sich Funktionen und Datenfolgen zu Sensordatenmodulen zusammenfassen lassen, die flexibel ansteuerbar sind und als *logische Sensoren* transparent in komplexe Echtzeitprogrammsysteme eingebunden werden können. Das Ziel ist die Bereitstellung eines Softwaresystems, das die Entwicklung und Implementierung von effizienten und frei skalierbaren Programmkomponenten und Applikationen für die Echtzeit-Bildfolgenanalyse und verteilte Sensordatenauswertung auf *Standardsystemen* unterstützt.

Eine wesentliche Grundlage dieser Arbeit ist die konsequente explizite Modellierung von *zeitlichen* Zusammenhängen, sowohl bei der sensorbasierten Datenerfassung und der Modellierung des äußeren Prozesses, als auch bei der Beschreibung der Datenverarbeitungsprozesse. Ersteres erlaubt die Einordnung der Daten in das Geschehen der realen Welt und die Modellierung der Szenendynamik, letzteres gestattet die Analyse von Ressourcenverbrauch und Performanz der Datenverarbeitungsprozesse.

Datenfolgen werden als eigenständige *Sequenzobjekte* modelliert, welche die kontinuierlich über die Zeit anfallenden Werte einer bestimmten Größe — Bilder, aber auch beliebige andere Sensordaten oder die daraus abgeleiteten Merkmale — zusammenfassen und gemeinsam mit dem jeweiligen Verarbeitungskontext verwalten. Darüber hinaus modelliert die Sequenzklasse allgemeine, vom konkreten Datentyp unabhängige Eigenschaften und Methoden von Datenfolgen, wie die Dateninitialisierung, den Zugriff auf aktuelle und alte Sequenzwerte, Mechanismen für die Datenaktualisierung und die Verwaltung von temporalen Merkmalen der Daten.

Die Einbindung der aus Entwurfssicht relevanten Funktionalitäten für die kontinuierliche Bereitstellung, Verarbeitung oder Auswertung dynamischer Datenfolgen erfolgt mit Hilfe von *Funktoren*. Diese kapseln konkrete Operatoren oder Sensoren einschließlich deren Parameter. Darüber hinaus modellieren sie allgemeine, verfahrensunabhängige Funktionsmerkmale, wie die Beziehungen zu den Ein- und Ausgangsdaten, Methoden für eine Analyse des Zeitverhaltens und Mechanismen für eine zyklische Operatorsteuerung. Dies erleichtert den Austausch von Verfahren einschließlich ihres gesamten Kontexts.

Die Datenflußbeschreibung eines Sensordatenmoduls kann mit Hilfe von Datensequenzen und Funktoren direkt in ein Softwaresystem übertragen werden, ohne daß der Kontrollfluß vom Entwickler explizit sequentialisiert werden muß. Die Steuerung erfolgt daten- oder anfragegetrieben und kann durch Softwareagenten überwacht werden. Die in dieser Arbeit entwickelten Spezifikationsmechanismen erlauben es, Programmsteuerung und Parallelisierungsgrad sowohl bei der Programmentwicklung als auch im laufenden System flexibel an unterschiedliche Randbedingungen anzupassen.

Die vorgestellten Konzepte wurden prototypisch als C++-Klassenbibliothek umgesetzt. In ihr werden die zentralen Klassen für die Repräsentation von Datensequenzen, Funktoren und Agenten sowie von Zeit zur Verfügung gestellt. Basierend auf diese Bibliothek wurde die Software für ein verteiltes, fußballspielendes Robotersystem — die AGILO-RoboCuppers — entwickelt und implementiert. Es wurde im RoboCup-Projekt getestet und bei verschiedenen internationalen Wettbewerben mit Erfolg eingesetzt.

Danksagung

Mein Dank gebührt zuerst Herrn Prof. Radig, der mir in der Forschungsgruppe Bildverstehen (FG BV) die Arbeit an diesem interessanten und spannenden Thema ermöglichte — für das in mich gesetzte Vertrauen, für die großen Freiräume, die er mir bei der Verwirklichung meiner Ideen im Rahmen des RoboCup-Projektes gewährte, und für die Unterstützung dieses spannenden Forschungsgebietes.

Ich möchte mich auch bei meinen Kollegen am Lehrstuhl und der MVTec Software GmbH bedanken, die stets für ein angenehmes Arbeitsklima sorgten, mich bei dem Projekt unterstützten und denen ich interessante Impulse für meine Arbeit verdanke. Besonders zu erwähnen sind Wolfgang Eckstein und Christoph Zierl — für ihre fachlichen Anregungen und die kritischen Anmerkungen hinsichtlich der Ausarbeitung.

Zu großem Dank bin ich meinen Mitstreitern beim Roboterfußball, den Agilo-RoboCuppers verpflichtet. Maximilian Lückenhaus, Thorsten Bandlow, Thorsten Schmitt und Robert Hanek seien stellvertretend genannt für alle Kollegen und Studenten, ohne deren Engagement und nächtliche Programmierarbeit — vor allem in Paris und Stockholm — ein erfolgreiches Auftreten unserer Roboter unmöglich gewesen wäre.

Besonderer Dank gilt meinen Freunden und Familienangehörigen, die mich trotz chronischen Zeitmangels noch nicht ganz abgeschrieben und mir immer wieder viel Mut gemacht haben. Meinen größte Dank aber schulde ich Petra, die mir viel Arbeit abgenommen und mich trotz eigener unaufschiebbarer Abgabetermine immer wieder liebevoll mit Cappuccino und anderen warmen Mahlzeiten versorgt hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

1.1 Motivation und Kontext

Die automatische Auswertung von Kamerabildern ist seit vielen Jahren Gegenstand der Forschung. Zahlreiche wichtige Ergebnisse resultieren aus diesen Aktivitäten — sowohl bei der Entwicklung allgemeiner mathematischer Modelle und leistungsfähiger Bildverarbeitungsoperatoren, als auch bei industriereifen Anwendungen, beispielsweise in der Luftbildauswertung, der Medizin oder der industriellen Bildverarbeitung. Ausdruck dessen sind eine Vielzahl kommerzieller Lösungen und ein wachsender Markt für die genannten Bereiche.

Dabei lassen sich Bildverarbeitungsaufgaben drei grundlegenden Anwendungsklassen zuordnen: der Einzelbildanalyse, der Auswertung zuvor aufgezeichneter Bildfolgen (offline-Analyse) und der (online) Analyse laufender Bildfolgen. Etablierte Produkte findet man vor allem in den ersten beiden Bereichen. In diesen Klassen wird der Softwaremarkt zunehmend durch umfangreiche Bildverarbeitungsbibliotheken geprägt, die zum einen Datenstrukturen für die verschiedenen ikonischen Datentypen als auch zahlreiche Bildverarbeitungsverfahren in Form von Operatoren bereitstellen und es dem Anwendungsentwickler so erlauben, auf Standardhardware mit Standardwerkzeugen und -programmiersprachen relativ schnell für neue Anwendungsgebiete Lösungen zu entwickeln. Zu diesen Bibliotheken und Softwaresystemen gehören z.B. HALCON (eh. HORUS [ES96]), Vista [PL94] und Khoros [KR94]. Rapid-Prototyping-Werkzeuge können den Entwickler beim Softwareentwurf unterstützen [ES97, RK94].

Im Gegensatz dazu sind trotz zahlreicher Forschungsergebnisse Produkte im Bereich der Online-Bildfolgenauswertung noch vergleichsweise selten. Dieser Bereich wird aufgrund des hohen Ressourcenbedarfs der Anwendungen (Rechenleistung, Speicherkapazität und Datendurchsatz) und des Fehlens von Standardbibliotheken immer noch von teuren Speziallösungen dominiert — etwa bei der videobasierten Steuerung von Kraftfahrzeugen und Robotern (z.B. [MD97, DM99, GB98]), bei Servicerobotern und mobilen Manipulatoren (z.B. [Bis97]) oder im Bereich von Kommunikationssystemen, die die Bewegungen, Gestik oder Mimik von Menschen auswerten (z.B. [HVD$^+$99, KHS99]). Dies betrifft zum einen den Hardwareaufbau (z.B. Echtzeit-Bildauswertungsgeräte [Ste87, Kon89, Mat93, Mel95]), vor allem jedoch die eingesetzte Software.

Besonders auf dem Hardware-Markt können seit wenigen Jahren jedoch entscheidende Veränderungen beobachtet werden. Computer werden seit ihrer Vermarktung kontinuierlich leistungsfähiger und preiswerter. Neu ist allerdings, daß aufgrund der rasanten Entwicklung und

zunehmenden Verbreitung von Multimediaanwendungen, die ähnliche Anforderungen an die Systemressourcen stellen wie die Bildfolgenanalyse, sowohl Highend-Computer als auch die notwendige Peripherie — insbesondere Kameras und Framegrabber- bzw. Videocapture-Karten — im preisgünstigen Consumer-Bereich angeboten werden.

Während jedoch die Hardware für die Bildfolgenanalyse immer billiger und leistungsfähiger wird, fehlt es an Softwaresystemen, die eine effiziente Entwicklung komplexerer dynamischer Anwendungen direkt unterstützen. Die Einschränkungen der bestehenden, i.d.R. für die Auswertung von Einzelbildern konzipierten Bibliotheken, ergeben sich vor allem aus dem Fehlen von Modellen, die die Dynamik der Szene und des Datenverarbeitungsprozesses adäquat darstellen können, was auch Auswirkungen auf die Programmsteuerung in derartigen System hat. So fehlen Mechanismen für die einfache und effiziente Einbindung von parallelen, nicht synchronisierten Datenverarbeitungsströmen und für eine flexible Vergabe der Ressource Rechenzeit an konkurrierende Teilaufgaben.

Motivation für diese Arbeit war die Erkenntnis, daß für die stärkere Verbreitung von Bildfolgenapplikationen Softwaresysteme notwendig sind, die die bestehenden Einschränkungen überwinden, indem sie u.a. elementare Datentypen, Methoden und Entwurfsmuster bereitstellen, mit denen die Dynamik von Bildfolgenprogrammen geeignet repräsentiert werden kann. Derartige Systeme sollten dabei nicht auf Bildfolgen beschränkt bleiben, sondern beliebige Sensordatenfolgen und die daraus extrahierten Merkmale darstellen können. Als erste Aufgabe ist dafür eine konsequente Modellierung der relevanten, die Dynamik des Prozesses beschreibenden Größen erforderlich: Zeit, Datenfolgen, Operatoren und Steuerungskomponenten.

Des weiteren erfordern die inhärente Parallelität und Komplexität der Anwendungen Konzepte und Mechanismen für die Programmsteuerung, die es u.a. erlauben, Bearbeitungszyklen flexibel zu skalieren und leicht an unterschiedliche Anforderungen anzupassen.

Die folgenden, für die Bildfolgenanalyse ebenfalls sehr wichtigen und interessanten Aspekte sollen hier dagegen *nicht* untersucht werden, da dies den Rahmen dieser Arbeit sprengen würde: So werden keine neuen Operatoren für die Extraktion von dynamischen Merkmalen aus Bildfolgen entwickelt. Und es werden auch keine neuen objektorientierten Datenstrukturen für die Repräsentation von aus Bildfolgen extrahierten ikonischen oder symbolischen Merkmalen — z.B. Korrespondenzen aufeinanderfolgender 2D-Merkmale — bereitgestellt.

Die vorliegende Arbeit ist wie folgt aufgebaut: Das 1. Kapitel steckt den Rahmen ab und untersucht die relevanten Eigenschaften des Anwendungsgebiets „Bildfolgenauswertung“. Darüber hinaus werden verwandte Arbeiten untersucht und aus den Systemanforderungen und dem Stand der Technik die Ziele, die mit dieser Arbeit verfolgt werden, konkretisiert.

In Kapitel 2 werden die verschiedenen Facetten der *RoboCup-Initiative* vorgestellt. Der „Roboterfußball“ steht dabei stellvertretend für die mit den untersuchten Konzepten anvisierten Applikationen: komplexe, dynamische Multisensor-Multiroboter-Multiagenten-Systeme. Das RoboCup-Projekt bildete den Rahmen für diese Arbeit und stellt das wichtigste Testszenario für die Validierung der Anwendbarkeit der vorgestellten Konzepte dar.

Aufgabe des 3. Kapitels ist es, ein konsistentes System zur quantitativen Modellierung und Darstellung von *Zeitausdrücken* — in Form von absoluten und relativen Zeitpunkten und Intervallen bereitzustellen.

In Kapitel 4 erfolgt die Modellierung der Basiselemente, mit deren Hilfe Programmkomponenten für die kontinuierliche Sensordatenauswertung — unabhängig von einer konkreten Anwendung — repräsentiert werden können. *Sequenzen* verwalten die zeitlich aufeinanderfol-

genden (Sensor-) Daten, und *Funktoren* übernehmen die Einbindung von Operatoren, die diese Datenfolgen bereitstellen, ineinander überführen oder auswerten. In diesem Rahmen wird auch der Begriff des *Sensors* näher untersucht. *Agenten* schließlich steuern und kontrollieren die zyklische Sensordatenverarbeitung.

Gegenstand des 5. Kapitels ist die Untersuchung der bei der Verarbeitung von Datenfolgen relevanten Zeitaspekte. Darauf aufbauend werden für die Basiselemente *Sequenz* und *Funktor* charakteristische Zeitattribute sowie das Zeitverhalten von Sensoren modelliert.

In Kapitel 6 werden grundlegende Aspekte der funktionalen Beschreibung der Sensordatenverarbeitungsmodule untersucht, wobei jedes Modul wiederum als ein logischer Sensor verstanden werden kann. Besondere Aufmerksamkeit wird dem Übergang von der auf Datensequenzen und Funktoren basierenden funktionalen Beschreibung der logischen Sensoren in eine konkrete Programmsteuerung gewidmet. Um den Kontrollfluß bei Bedarf an verschiedene Randbedingungen anpassen zu können, ohne daß der Entwickler die funktionale Beschreibungsebene verlassen muß, werden zusätzliche, über den Datenfluß hinausgehende Beschreibungsmittel eingeführt.

Kapitel 7 hat die objektorientierte System-Modellierung zum Inhalt. Sie baut auf den Konzepten und Modellen der Kapitel 4 – 6 auf und bildet die Grundlage für deren Realisierung in einem Softwaresystem. Dieses Kapitel bewegt sich in der konzeptionellen bzw. strukturellen Entwurfsebene, auf die Implementierung in C++ wird nur am Rande eingegangen.

In Kapitel 8 werden schließlich die Ergebnisse der Arbeit zusammengefaßt. Dabei wird auf die Umsetzung der Konzepte und deren Einsatz im RoboCup eingegangen. Darüber hinaus werden Ansatzpunkte für weiterführende Forschungen aufgezeigt.

1.2 Systemanforderungen durch den äußeren Prozeß

Die Anforderungen, die an ein Softwaresystem hinsichtlich seines Zeitverhalten, einer möglichen oder notwendigen Parallelisierung und der sich daraus ergebenen Systemkomplexität gestellt werden, hängen stark vom Grad der Kopplung des Systems an den äußeren Prozeß ab, sowie von den Eigenschaften, die diesen Prozeß kennzeichnen. Unter dem Begriff „*äußerer Prozeß*“ werden all die Vorgänge zusammengefaßt, die die Anwendungsumgebung kennzeichnen, und mit denen die Applikation interagiert. Die verschiedenen Zustände oder Zustandsänderungen des äußeren Prozesses wirken sich i.d.R. maßgeblich auf das Verhalten des Softwaresystems aus, dessen Aufgabe es wiederum ist, den Prozeß entsprechend der gestellten Aufgabe zu beeinflussen.

Die wichtigste Eigenschaft des äußeren Prozesses in diesem Zusammenhang stellt dessen Zeitverhalten dar, das sich folgendermaßen kategorisieren läßt:

Offline: Es existieren keine zeitlichen Restriktionen, weder hinsichtlich des Zeitpunkts noch der Dauer der Datenverarbeitungprozesse. Derartige Anwendungen sind dadurch gekennzeichnet, daß Aufnahme bzw. Messung des Prozeßzustands und dessen Auswertung zeitlich getrennt erfolgen und keine direkte Rückkopplung von der Anwendung auf den äußeren Prozeß existiert; äußerer Prozeß und Auswertungsprozeß laufen also nacheinander ab. Aus dem Bereich der Bildverarbeitung fällt i.d.R. die Einzelbildauswertung und die Auswertung aufgezeichneter Bildfolgen in diese Kategorie.

Online, wartend: Zeitliche Restriktionen bestehen nur hinsichtlich des Zeitpunkts der Datenverarbeitung — der äußere und der Softwareprozeß laufen gleichzeitig ab — nicht jedoch bezüglich der Dauer der Berechnungen. Nach einer Zustandsänderung des äußeren Prozesses wartet dieser solange, bis das Softwaresystem die Auswertung der Daten abgeschlossen und eine entsprechende Antwort generiert hat. Die Antwortzeit ist dabei nicht kritisch, da in der Zwischenzeit keine relevanten externen Zustandsänderungen auftreten, und die Wartezeit (in einem bestimmten Rahmen) beliebig groß sein kann. Interaktive Systeme oder — aus dem Bereich der Bildverarbeitung — bildbasierte Identifikationssysteme lassen sich dieser Kategorie zuordnen.

Online, weiche Zeitrestriktionen: Der äußere Prozeß läuft parallel zum Softwaresystem und kann dabei jederzeit und unabhängig von der Software seinen Zustand ändern. Diese Änderungen sollten dem Softwaresystem nicht verborgen bleiben, erfordern also eine gewisse Kontinuität bei der Beobachtung des äußeren Prozesses. Darüber hinaus sollte auf äußere Zustandsänderungen schnell genug reagiert werden, was die zu erreichende Reaktionszeit definiert. Allerdings ist es bei Prozessen dieser Kategorie nicht kritisch oder gefährlich, wenn die gewünschte Antwortzeit einmal nicht eingehalten werden kann, solange in der überwiegenden Zeit die geforderte Funktionalität gewährleistet bleibt.

Online, harte Zeitrestriktionen: Prozesse dieser Kategorie besitzen ein ähnliches Zeitverhalten wie die zuvor beschriebenen Prozesse mit weichen Zeitrestriktionen. Hier ist die Einhaltung bestimmter Zeitschranken allerdings zwingend vorgeschrieben, da schon kurzzeitige Ausfälle die Funktionalität des Gesamtsystems gefährden und kritische Zustände hervorrufen können. Die Software muß daher in der Lage sein, die geforderten Antwortzeiten zu garantieren, was i.d.R. ein Echtzeitbetriebssystem voraussetzt. Harte Zeitrestriktionen treten vor allem im Zusammenhang mit mechanisch bewegten Teilen auf, für die Steuerbefehle in bestimmten Zeitintervallen abgegeben werden müssen, um kritische Situationen, in denen die Systeme selbst oder Personen in deren Umgebung gefährdet werden könnten, zu vermeiden oder um das System am Leben zu erhalten. Eine andere Ursache für harte Zeitanforderungen kann in der Systemdynamik selbst liegen, beispielsweise wenn in Bildfolgen die Korrespondenzsuche nur dann eindeutig ist, falls die Zeiträume zwischen den Bildaufnahmen klein genug sind, und *eine* Fehlkorrespondenz bereits ausreicht, um das Gesamtergebnis nachhaltig zu verfälschen.

Systeme der beiden zuerst genannten Kategorien (Offline-Systeme und wartende Online-Systeme) sollen in dieser Arbeit keine Rolle spielen, da die bestehenden Softwaresysteme für deren Realisierung i.d.R. ausreichen.

Bei Online-Systemen mit Zeitrestriktionen stellt sich die Situation grundsätzlich anders dar. Durch die Forderung, *gleichzeitig* verschiedene zeitliche Vorgaben erfüllen und unterschiedliche Aufgaben bearbeiten zu müssen, sind derartige Programme im allgemeinen deutlich komplexer. Bei diesen Systemen bestimmt im wesentlichen der in der realen Zeit ablaufende äußere Prozeß das Zeitverhalten des Gesamtsystems. Aus diesem Grund werden diese Systeme auch als Echtzeitsysteme bezeichnet. In dieser Arbeit sollen vorrangig solche Echtzeitsysteme betrachtet werden, die mit Hilfe einer Kamera kontinuierlich Szenendaten aufnehmen, um diese auszuwerten. Bei ihnen handelt es sich um „Programmsysteme für die Echtzeit-Bildfolgenauswertung“ oder kurz *„Echtzeit-Bildfolgenprogramme“*. Die vorgestellten Konzepte sollen dabei

jedoch keineswegs auf Bildfolgen beschränkt bleiben, sondern auf beliebige Sensordatenfolgen anwendbar sein. Die interessierenden Applikationen sollen als *„dynamische Sensordatenprogramme“* bezeichnet werden. Der Begriff der „Echtzeitfähigkeit“ definiert sich in diesem Zusammenhang nicht über die Fähigkeit, Bilddaten in Videorate verarbeiten zu können, sondern vielmehr über die geforderten Systemeigenschaften „*kontinuierlich*“, „*schritthaltend*“ und „*rechtzeitig*“. Diese Eigenschaften stehen dabei in einem engen Zusammenhang mit der Prozeß- oder Systemdynamik, d.h. sie werden nicht als absolute Größen definiert, sondern hängen stark von dem Tempo ab, mit dem sich Veränderungen in der Szene vollziehen. Definieren lassen sich diese Eigenschaften wie folgt:

Kontinuität: Die hier untersuchten Prozesse und Systeme sind dadurch gekennzeichnet, daß sie kontinuierlichen Veränderungen unterworfen sind. Daraus folgt zum einen, daß — entsprechend der Systemdynamik — die Zeiträume zwischen zwei Beobachtungen der Szene so klein sein müssen, daß wesentliche Änderungen in der Umgebung nicht unentdeckt bleiben. Zum anderen sollte das System in der Lage sein, trotz seiner diskreten Arbeitsweise die kontinuierlichen Veränderungen der beobachteten Größen darstellen und ggf. fehlende Werte interpolieren zu können. Diese Eigenschaft ist insbesondere notwendig, um die diskreten Daten unterschiedlicher, nicht synchronisierter Sensoren abzugleichen und so eine gemeinsame Zeitbasis für sie bereitzustellen.

Schritthaltende Datenverarbeitung – Wiederholungszeit: Im Gegensatz zur Offline-Analyse steht nicht beliebig viel Rechenzeit zur Verfügung. Dies bedeutet, daß die verwendeten Algorithmen so schnell sein müssen, daß sie mit der erforderlichen Verarbeitungs- oder Aufnahmerate Schritt halten können. Wie hoch diese Rate ist, hängt in erster Linie von der Systemdynamik und den verwendeten Algorithmen ab. Sollen beispielsweise bestimmte Bildmerkmale in einer Bildfolge verfolgt werden, ist es für die Korrespondenzsuche wichtig, daß sich die Merkmale von Bild zu Bild nicht zu stark bewegen oder aufgrund einer verschobenen Perspektive verändern. Die Zeit, die maximal zwischen zwei Aufnahmen toleriert werden kann, hängt dabei stark von den Geschwindigkeiten ab, mit denen sich die entsprechenden Szenenobjekte oder die Kamera in der Szene bewegen.

Rechtzeitigkeit – Reaktionszeit: Ein weiteres Merkmal bezüglich der Rechenzeit der Algorithmen ergibt sich aus der Forderung an das System, auf Zustandsänderungen der Umgebung schnell genug, d.h. „rechtzeitig“ — wiederum entsprechend der Systemdynamik — mit einer geeigneten Aktion reagieren zu können. Im Gegensatz zur Wiederholungszeit ist dies nicht unbedingt die Zykluszeit, also die Zeit zwischen zwei Aufnahmen bzw. Messungen, sondern die Zeitspanne zwischen einer Aufnahme und der auf ihr basierenden Aktion. Insbesondere in Mehrprozessormaschinen kann die Reaktionszeit deutlich über der Wiederholungszeit liegen.

Für die Echtzeitfähigkeit einer Anwendung sind alle drei Punkte gleichermaßen von Bedeutung. Mit welcher Sicherheit die jeweiligen Reaktions- oder Wiederholzeiten eingehalten werden müssen, hängt von der Anwendung ab. Falls auch nur eine dieser Zeiten in einem Programmzweig unbedingt eingehalten werden muß, handelt es sich um ein System mit harten Zeitrestriktionen, was ein Echtzeitbetriebssystem erfordert. Für viele Anwendungen reicht es jedoch aus, die Zeitforderungen mit einer gewissen Wahrscheinlichkeit einzuhalten. Ebenfalls möglich ist, daß für einzelne Teilaufgaben (z.B. die Roboteransteuerung) harte Zeitrestriktionen bestehen und durch ein Subsystem garantiert werden, während für andere Programmzweige

(etwa die Bildverarbeitung) weiche Zeitforderungen ausreichen. Ist die Bildverarbeitung dann beispielsweise einmal nicht in der Lage, ihre Daten schnell genug zu liefern, kann das von der Robotersteuerung erkannt und entsprechend berücksichtigt werden.

Während in den bisherigen Überlegungen immer davon ausgegangen wurde, daß der äußere Prozeß fest vorgegeben ist und bestimmte Anforderungen an das Zeitverhalten des Programmsystems und die eingesetzte Hardware stellt, kann das Problem auch von der anderen Seite her betrachtet werden: Mit den zur Verfügung stehenden Mitteln ist ein Programmsystem in der Lage, bestimmte Zeiten mit einer gewissen Sicherheit einzuhalten. Geeignete Mechanismen für eine Analyse des Zeitverhaltens vorausgesetzt lassen sich daraus dann Aussagen über die zulässigen Randbedingungen für den Einsatz des Systems (z.B. maximale Geschwindigkeit oder maximale Anzahl der Objekte in der Szene) oder über die Zuverlässigkeit der Ergebnisse in bestimmten Situationen ableiten.

1.3 Merkmale von Echtzeit-Bildfolgenprogrammen

Nach den im vorhergehenden Abschnitt formulierten elementaren Eigenschaften von Echtzeit-Bildfolgenprogrammen, sollen hier nun weitere charakteristische Merkmale untersucht werden. Diese Merkmale können direkt aus den Anforderungen der untersuchten Anwendungsklassen abgeleitet werden. Dabei muß jedoch nicht jedes Bildfolgen- oder Sensordatenprogramm alle diese Eigenschaften aufweisen. Die ersten beiden betreffen das Zeitverhalten einzelner Programmzweige:

Zyklen unterschiedlicher Länge: Unterschiedliche Teilaufgaben können mit unterschiedlichen Zykluszeiten verbunden sein. Kurze Zyklen bei der Sensordatenaufbereitung stehen mittleren oder langen Zykluszeiten in der Planungsebene oder im Bereich der Modellgenerierung bzw. -adaption gegenüber. Darüber hinaus besitzen viele Sensoren charakteristische Datenraten, die von Sensor zu Sensor sehr unterschiedlich ausfallen können und i.d.R. die Zykluszeiten der anschließenden Datenauswertung mitbestimmen. Als Beispiele hierfür seien die unterschiedlichen Meßraten von Video-, Odometrie- und Ultraschalldaten genannt.

Variable Zykluszeiten: Bei zyklischen Prozessen kann im allgemeinen weder von einer festen noch von einer genau bekannten Zykluszeit ausgegangen werden. Das gilt oft selbst dann, wenn Daten von Sensoren ausgewertet werden, die an und für sich zyklisch arbeiten, wie das bei Videobilddaten der Fall ist. Mitverantwortlich für die Dauer eines Zyklus, in dem zum Beispiel die kontinuierliche Bildauswertung erfolgt, ist die Rechenzeit der Algorithmen für die Sensordatenanalyse, die wiederum stark vom Bildinhalt abhängen kann, ebenso wie vom Ressourcenbedarf anderer, parallel laufender Programmzweige. Dieser kann aber seinerseits wiederum sowohl von den Sensordaten als auch von der aktuell auszuführenden Aufgabe abhängen, sich also im laufenden Programm ändern.

Weiterhin ergeben sich bestimmte Systemmerkmale aus der den Anwendungsgebieten innewohnenden Komplexität, d.h. dem Umfang der Datenströme, Querbeziehungen zwischen ihnen, der Anzahl der zur Lösung eingesetzten Teilsysteme und der Menge gleichzeitig zu bewältigender Aufgaben:

Datenumfang: Die kontinuierliche Verarbeitung von Bildfolgen setzt einen hohen Datendurchsatz des Systems sowie die Fähigkeit, große Mengen komplexer Datenströme verwalten zu können, voraus. Bereits in der Softwareentwicklungsphase stellt diese Datenfülle ein erhebliches Problem dar. Bei Testläufen und während der Fehlersuche kommen herkömmliche Visualisierungswerkzeuge und Debug-Methoden aufgrund der Nebenläufigkeit mehrerer Programmzweige und der vom System geforderten Kontinuität bei der Programmausführung schnell an ihre Grenzen. Aus diesem Grund werden Verfahren benötigt, die einerseits Datenströme für deren Visualisierung optimal aufbereiten können, andererseits aber das laufende Programm dabei so wenig wie möglich beeinflussen.

Parallelität: Bildfolgenanalyse und Sensordatenaufbereitung stellen i.d.R. nur eine Teilaufgabe des Gesamtsystems dar. Parallel dazu erfordern die Anwendungen zumeist die Bewältigung weiterer Aufgaben — in Robotersystemen z.B. die Bewegungsplanung, Robotersteuerung und die Generierung bzw. Adaption von Modell- oder Umgebungswissen. Neben der Aufgabenparallelität findet man Parallelität oder Nebenläufigkeit auch in verschiedenen anderen Programmebenen. Aufgrund der hohen Rechenzeitanforderungen spielen Multiprozessorsysteme in der Bildfolgenanalyse eine besondere Rolle. Die erhöhte Leistungsfähigkeit sollte nach Möglichkeit ohne besonderen Mehraufwand des Anwendungsentwicklers vom System ausgenutzt werden können. Aufgrund ihrer Bedeutung werden die verschiedenen Aspekte der Parallelität in Abschnitt 1.4.3 gesondert untersucht.

In komplexen Robotik- und Multisensoranwendungen spielt schließlich auch eine flexibel konfigurierbare Programmsteuerung eine wichtige Rolle:

Asynchrone Programmzweige: Neben zyklischen Vorgängen können in Echtzeit-Applikationen auch asynchrone, ereignisgesteuerte Programmzweige auftreten. Signale, die vom äußeren Prozeß bzw. von speziellen, den Prozeß überwachenden Sensoren generiert werden, haben hier die Aufgabe, bestimmte Aktionen zu triggern oder wartende Programmzweige zu reaktivieren.

Dynamische Konfiguration: Eine immer wichtiger werdende Forderung in komplexen Anwendungen ist, daß das System auf veränderte Randbedingungen sinnvoll reagieren und dynamisch (re-) konfigurierbar sein soll. Für diese Aufgabe können beispielsweise Agenten eingesetzt werden — Softwareobjekte, die selbständig in der Lage sind zu entscheiden, ob, und wenn ja, welche Methoden aufzurufen sind, um Programmteile im Sinne der aktuellen Aufgabe zu manipulieren. Der Agentenbegriff, wie er in dieser Arbeit verwendet wird, orientiert sich im wesentlichen an den in [Lüc00] genannten Kriterien Autonomie, Aktivität, Rolle, Agentenzyklus, explizite Wissensrepräsentation und Fähigkeit zur komplexen Kommunikation.

Systeme, deren Ziel es ist, allgemeine Verfahren für die Echtzeit-Bildfolgenauswertung bereitzustellen, müssen sich diesen Anforderungen stellen und Konzepte für deren Umsetzung anbieten.

1.4 Ausgangssituation und verwandte Arbeiten

Bei der Umsetzung dynamischer Echtzeitsysteme wird man mit zwei gegensätzlichen, wenn nicht gar sich widersprechenden Forderungen konfrontiert: Die Notwendigkeit einer optimalen Ausnutzung der Systemressourcen steht dem Wunsch nach einer effizienten Programmentwicklung mit standardisierten Werkzeugen und Komponenten entgegen. Aufgrund der hohen Anforderungen an die Hardware, die bisher immer an ihren Leistungsgrenzen betrieben wurde, hat die Laufzeiteffizienz i.d.R. den Vorrang vor allgemeinen, wiederverwendbaren Datenstrukturen erhalten. So basieren die in der Vergangenheit entwickelten Lösungen meist auf teurer Spezialhardware und sind mit Software ausgestattet, die speziell auf eine konkrete Anwendung zugeschnitten und für diese optimiert wurde.

Diese enge Ausrichtung von Entwicklungen auf eine ganz bestimmte Anwendung und einen stark spezialisierten Hardwareaufbau spiegelt sich auch in den zugrundeliegenden Modellen wider. Diese sind i.d.R. stark vereinfacht und beschränken sich auf einige wenige Aspekte der Anwendungsdomäne. Viele Annahmen über das System wandern dabei als implizites Wissen in die Implementierung. Das macht eine Wiederverwendung der entwickelten Komponenten nahezu unmöglich und selbst kleine Veränderungen des Einsatzgebietes oder der Austausch von Hardware erfordern die Neuentwicklung großer Teile des Systems nahezu von Grund auf. Diese Vorgehensweise ist höchst unwirtschaftlich und stellt ein Haupthindernis für eine schnelle und effiziente Programmentwicklung dar.

Ansätze für ein allgemeines applikationsunabhängiges Bildfolgensystem müssen sich daher zuerst mit der Modellierung der zentralen Größen und Komponenten von Bildfolgenprogrammen auseinandersetzen. Zu diesen Größen zählen vor allem die Zeit und zeitliche Ausdrücke, weiterhin die dynamischen, über die Zeit zu beobachtenden Daten, Möglichkeiten für die Einbindung der Operatoren, die diese Daten bereitstellen oder verarbeiten und schließlich aktive Komponenten, die für die Steuerung des Programmablaufs verantwortlich sind.

In den folgenden Abschnitten wird untersucht, auf welche Vorarbeiten hinsichtlich der Modellierung dieser Größen zurückgegriffen und aufgebaut werden kann. Begonnen wird im Abschnitt 1.4.1 mit einer Übersicht über verschiedene Arbeiten zur rechnerinternen Repräsentation zeitlicher Ausdrücke. Im darauf folgenden Abschnitt werden bestehende Ansätze zur Repräsentation von Bildfolgen untersucht. Abschnitt 1.4.3 beleuchtet die Aspekte Parallelität und Nebenläufigkeit in Bildfolgensystemen. Hier werden vor allem die inhärente Parallelität sowie Möglichkeiten, diese darzustellen und auf parallele Hardware zu übertragen, untersucht. Die objektorientierte Programmierung stellt ein mächtiges Werkzeug für die Repräsentation komplexer Zusammenhänge auf hohem Abstraktionsniveau dar. Auf deren Stellung und Einsatzmöglichkeiten in der Bildverarbeitung wird in Abschnitt 1.4.4 eingegangen.

1.4.1 Repräsentation von Zeit

Ansätze für eine explizite Repräsentation von Zeit und zeitlichen Ausdrücken in Computerprogrammen findet man vor allem im Bereich temporaler Logiken und wissensbasierter Systeme. Erstere stellen Operatoren bereit, die z.B. für die Spezifikation und Verifikation nebenläufiger Programme eingesetzt werden [Hai82, Lam83, Krö87]. In wissensbasierten Systemen werden zeitliche Aussagen insbesondere im Bereich der temporalen Planung verwendet, wozu auch das temporale Schließen und die Terminplanung gehören. Unter temporaler Planung versteht man

dabei den Vorgang, aus einer Menge möglicher Aktionen eine zeitliche Abfolge von Aktionen so festzulegen, daß ausgehend von einer gegebenen Anfangssituation ein gewünschter Endzustand erreicht werden kann [AK83, BD89]. Wird die Aktionsplanung parallel zum laufenden Prozeß durchgeführt, wird das auch als Echtzeitplanung bezeichnet [Dor89].

Bei der temporalen Planung sind zeitliche Abhängigkeiten zu berücksichtigen, deren Konsistenz mit Hilfe des temporalen Schließens überprüft werden kann. Darüber hinaus kann das temporale Schließen dazu verwendet werden, neue temporale Abhängigkeiten aus den bestehenden Restriktionen herzuleiten [McD82, Gam96]. Ziel der Terminplanung ist es, unter Berücksichtigung bestehender Abhängigkeiten die Reihenfolge ausgewählter Aktionen und deren Ressourcenbedarf festzulegen [Win94].

Die rechnerinterne Repräsentation zeitlichen Wissens spielt auch in sprachverstehenden Dialogsystemen eine Rolle. Derartige Systeme — z.B. Fahrplanauskunftssysteme der Deutschen Bahn [HUKW86, AO94, OA94] — setzen Rechenanlagen ein, um sprachliche Zeitangaben vom Menschen zu verstehen und korrekt zu interpretieren. Untersuchungen zu linguistischen Aspekten von Zeitangaben und -repräsentationsformen sowie zu der Bedeutung temporaler Konstituenten mit zahlreichen weiterführenden Literaturangaben findet man u.a. in [Hil95].

Im Zusammenhang mit Bildverarbeitungssystemen gibt es bisher nur sehr wenige Arbeiten zu einer expliziten Darstellung von Zeit. Für die automatische Planung und Erzeugung von optimalen Programmen für Echtzeit-Bildauswertungsgeräte modelliert Melchert die Reaktionszeit und die Wiederholzeit von Bildauswerteprozessen sowie die Zeitanforderungen externer Prozesse [Mel95]. Ausreichend gut ist laut Melchert das generierte Programm dann, wenn es die vorgegebenen Mindestanforderungen an die Verarbeitungsgeschwindigkeit einhält und dabei nur soviel der zur Verfügung stehenden Ressourcen verwendet, wie unbedingt erforderlich sind. Zeitangaben werden dabei nur für die Planung der Maschinenprogramme verwendet, in der Bildverarbeitung selbst stehen sie jedoch nicht zur Verfügung. Zugriffe in die Vergangenheit werden mit Hilfe einer Verzögerungs- oder Vorgänger-Funktion realisiert, wofür ein synchrones Datenflußmodell mit konstanter Datenrate vorausgesetzt wird. Zeitliche Beziehungen asynchroner Prozesse und variable Zykluszeiten lassen sich dabei nicht modellieren.

Der Versuch eines Brückenschlags zwischen einem Logikprogrammiersystems für die unscharfe temporallogische Programmierung [Brz94] über Wissensrepräsentationsformalismen bis hin zur Bildfolgenauswertung und Robotik wird in [Sch96] beschrieben. Diese Arbeit stellt eine auf einer metrischen Zeitlogik basierende Sprache als Wissensrepräsentation für Situationsbeschreibungen vor. Durch die Behandlung unscharfer temporaler Operatoren geht sie über den Sprachumfang rein unscharfer und rein temporallogischer Sprachen hinaus. Für die problemangepaßte Beschreibung von Situationsabfolgen und deren assoziierten Handlungen und Handlungserwartungen werden Situationsgraphenbäume verwendet, die in zeitlogische Programme übersetzt werden können.

Die meisten Arbeiten, die sich mit Algorithmen für die Bildfolgenauswertung befassen, vereinfachen die Modellierung der Zeit dahingehend, daß sie von einer konstanten Zeitdifferenz zwischen zwei aufeinanderfolgen Bildaufnahmen ausgehen ($\Delta t = const$, konstante Framerate) [Ill95, Jäh97]. Andere gehen noch einen Schritt weiter und vernachlässigen die reale Zeit ganz: $\Delta t = 1$ [IW94].

Diese Herangehensweise spiegelt sich in den meisten der in der Praxis eingesetzten Bildverarbeitungssysteme wider. In diesen wird weder die Beobachtungszeit noch das Zeitverhalten der verwendeten Soft- und Hardwarekomponenten wie Sensoren oder Bildverarbeitungsope-

ratoren explizit modelliert. Für die Bildaufnahme wird angenommen, daß sie zu diskreten, äquidistanten Zeitpunkten erfolgt, die auf die Menge der natürlichen Zahlen als Bildindizes abgebildet werden. Dabei wird ein synchrones Datenflußmodell vorausgesetzt und der aktuelle Bearbeitungszeitpunkt oder -zeitraum meist mit der Aufnahmezeit gleichgesetzt. Daten von unterschiedlichen Sensoren werden damit als *gleichzeitig* gemessen angenommen. Die Dauer der Messung (Belichtungs- oder Shutterzeit) wird i.d.R. genauso ignoriert, wie die Tatsache, daß sich vollaufgelöste PAL- oder NTSC-Bilder aus zwei zu unterschiedlichen Zeitpunkten aufgenommenen Halbbildern zusammensetzen.

Aus den zuvor genannten Vereinfachungen ergeben sich zum einen Ungenauigkeiten bei der Bestimmung der Systemdynamik sowie der Verzicht auf Flexibilität, Skalierbarkeit und dadurch auf die Wiederverwendbarkeit der Entwicklungen. Besitzt die Applikation eine Komponente für die Echtzeitplanung von Aktionen, ist eine explizite Zeitmodellierung für die adäquate Repräsentation von Beobachtungszeiten sowie dem Zeitverhalten von Sensoren oder Aktionen jedoch unverzichtbar. Gleiches gilt bei verteilten Systemen, in denen die Daten mehrerer Rechner ausgetauscht oder fusioniert werden sollen. Hier sind nicht nur die genauen Aufnahmezeiten der Sensordaten von Bedeutung, sondern auch die zeitliche Synchronisation der Rechner.

Aus den genannten Gründen soll in dem hier vorgestellten System die Zeit explizit modelliert werden. In Kapitel 3 werden dafür die Grundlagen gelegt. Absolute Zeitpunkte dienen der zeitlichen Zuordnung von Daten und Aktionen und der zeitlichen Synchronisation von Daten unterschiedlicher Quellen. Mit Hilfe von Intervallen lassen sich Zeiträume, in denen Daten gültig sind oder über denen Daten angesammelt wurden, Zeitanforderungen an Operatoren und Aktionen sowie mittlere Rechenzeiten beschreiben. Relative Zeitpunkte schließlich erlauben es, das Alter von Daten und den zeitlichen Versatz zwischen verschiedenen Daten auszudrücken.

1.4.2 Modellierung von Bildsignalen als dynamische Datenfolgen

Eine Umgebung oder Szene läßt sich im allgemeinen durch eine Menge von Vektorfunktionen, die den Verlauf verschiedener charakteristischer Größen über die Zeit wiedergeben, beschreiben. Damit reale Objekte oder Systeme — ganz gleich ob künstliche oder natürliche — in ihrer Umgebung agieren können, müssen einige Größen oder Szenenmerkmale mit Hilfe von Sensoren meßtechnisch erfaßt werden, was sie zu kontinuierlichen Eingangssignalen der entsprechenden Sensoren macht [PH95]:

$$U = f^r(\vec{x}) |\ r = 1, 2, \ldots$$

Während manche spezialisierte Sensoren — wie z.B. Temperatursensoren — die interessierenden Größen direkt als Eingangssignal verwenden und in eine für die Weiterverarbeitung geeignete Form umwandeln können, geht die kamerabasierte Informationsverarbeitung einen Umweg. Das kontinuierliche Eingangssignal eines CCD-Sensors ist die — ggf. spektralabhängige — Helligkeitsverteilung der durch ein optisches System in die Bildebene projizierten Raumpunkte. Die eigentlich interessanten Informationen, beispielsweise die Position und Orientierung einzelner Objekte der Szene, sind — wenn überhaupt — nur implizit im Bildsignal enthalten. Ein zentrales Ziel der Bildverarbeitung ist es nun, diese impliziten Informationen zu extrahieren, um sie in eine explizite und damit für die Weiterverarbeitung geeignete Darstellungsform zu überführen. Die verschiedenen Bearbeitungsschritte auf dem Weg dahin lassen

sich als eine Abfolge von Signaltransformationen auffassen. Das Ergebnis ist, ausgehend von dieser Betrachtungsweise, wiederum ein Signal. Komponenten, die diese Signale liefern, können als logische Sensoren bezeichnet werden.

Der erste Schritt auf dem Weg zur computergestützten Bildsignalverarbeitung ist die Digitalisierung des analogen Bildsignals. Im Rechner steht danach ein digitales Bildsignal zur Verfügung, das durch drei Arten von Diskretisierung gekennzeichnet ist:

1. Zeitliche Auflösung: In erster Linie durch die Aufnahmerate wird festgelegt, wie groß Veränderungen in der Szene sein dürfen, die noch verlustfrei aus den digitalen Signalen rekonstruiert werden können. Für die mathematische Beschreibung der Zusammenhänge spielt das Abtasttheorem eine besondere Rolle.
2. Räumliche Auflösung: Diese legt die Größe der kleinsten, gerade noch erkennbaren Strukturen fest und stellt einen Zusammenhang her zwischen der Größe von Objekten, deren Entfernung und der erlaubten Bewegung.
3. Quantisierung der Signalwerte: Hierdurch wird festgelegt, welche konkreten Werte (aus einer endlichen Menge, z.B. $0, 1, \ldots, 255$) ein Signal annehmen kann, was wesentlich die Güte des abgetasteten Signals bestimmt.

Unter einer Bildfolge versteht man nun ein derart diskretisiertes und im Rechner bereitgestelltes Bildsignal. Eine allgemeine, informelle Definition für Bildfolgen findet sich bei Haralick und Shapiro:

> *A time-varying image, multitemporal image, dynamic imagery, or image time sequence is a multi-image set in which each successive image is taken of the same scene at a successive time. Between successive snapshots, the objects in the scene may move or change and the sensor may move.* [HS93]

Entsprechend dieser Definition sind die wichtigsten Eigenschaften einer Bildfolge: Es handelt sich um eine *Multimenge* von Bildern, die in *einer* (*dynamischen*) Szene von *einem* Sensor aufgenommen wurden und *zeitlich geordnet* sind.

Diese Definition läßt sich leicht von Bilder auf beliebige Datentypen erweitern. Dabei kann der Begriff des Sensors weitergefaßt werden und — als logischer Sensor — zur Bezeichnung beliebiger Datenquellen dienen. Eine dynamische Datenfolge ist dann die Menge all der Daten, die von einem logischen oder physikalischen Sensor in einer Szene über die Zeit geliefert werden, um eine charakteristische Größe der Szene (zu diskreten Zeitpunkten) zu beschreiben. Das aber ist nichts anderes als ein digitales Signal, das wiederum von anderen Komponenten abgegriffen, modifiziert und ausgewertet werden kann. Die einzelnen Signalwerte stehen zueinander in einem engen semantischen Zusammenhang: Sie stammen vom gleichen Sensor und beschreiben gemeinsam den Verlauf *einer* bestimmten Szenengröße.

Diese Betrachtungsweise motiviert die explizite Modellierung von Datenfolgen bzw. digitalen Signalen im Rechner als eigenständige Instanzen, für die verschiedene Attribute und Operationen definiert werden. Insbesondere lassen sich so eine Reihe allgemeiner, vom konkreten Datentyp unabhängiger Operationen umsetzen, etwa der Zugriff auf vergangene Werte, die Dateninitialisierung, die Freigabe alter, nicht mehr benötigter Werte, die Analyse von Rechenzeit und Ressourcenbedarf oder der transparente Zugriff auf dynamische Daten in einem verteilten System. Der Zugriff ist dabei für alle dynamischen Daten einheitlich, da sie immer als ein

am Ausgang eines logischen Sensors anliegendes Signal betrachtet werden, ganz gleich, ob es sich um Bildfolgen oder um Sequenzen von aus Bildern extrahierten Merkmalen handelt, und unabhängig davon, ob diese Sequenzen von einer Kamera oder einer Dateiliste, durch einen digitalen Signalprozessor oder einen in Software realisierten Filter, durch lokale Operatoren oder auf einem beliebigen anderen Rechner im Netz geliefert wurden.

Die in der Praxis eingesetzten Bildverarbeitungssysteme besitzen kein derartiges Signal- bzw. Sequenzmodell. Bildfolgen werden im Computer i.d.R. lediglich in Form von Bildmatrizen repräsentiert, die in regelmäßigen Abständen überschrieben werden. Um Aussagen über Veränderungen im Bild treffen zu können, wird eine zweite Matrix als Zwischenspeicher für das vorhergehende Bild zu Hilfe genommen [Jäh97]. Zugriffe weiter in die Vergangenheit können mit diesem Ansatz nur sehr umständlich realisiert werden, außerdem geht dabei die Signalsemantik, d.h. der über alle Daten einer Folge gemeinsame Kontext, verloren.

In anderen Ansätzen wird die Zeitachse als dritte räumliche Dimension interpretiert, d.h. eine Folge zweidimensionaler Orts-Bilder wird zu einem dreidimensionalen Orts/Zeit-Bild. Auf dieses werden dann z.B. 3D-Bildverarbeitungsmethoden angewendet [Jäh97]. Diese Darstellungsform ist jedoch nicht für die kontinuierliche Bildauswertung geeignet, d.h. deren Einsatz ist auf den Bereich der Offline-Bildanalyse beschränkt.

Wie die Untersuchungen gezeigt haben, existieren bisher keine Modelle für Bild- und vergleichbare Datenfolgen, die deren Signalcharakter adäquat darstellen können. Diese Lücke soll in dieser Arbeit durch das im Abschnitt 4.2 vorgestellte Sequenzmodell geschlossen werden.[1]

1.4.3 Parallele Aspekte und funktionalsprachliche Ansätze in der Bildverarbeitung

Im Bereich der Bildverarbeitung wird die Analyse von Parallelität und Nebenläufigkeit durch verschiedene Fragestellungen und Probleme motiviert. Zum einen ist die Bildverarbeitung ein Gebiet, das sich aufgrund seiner inhärenten Parallelität sehr gut zur Parallelisierung eignet, zum anderen ist der Einsatz von parallelen Rechnerarchitekturen oft der einzige Weg, die erforderliche Rechenleistung für die Lösung umfangreicher und komplexer Aufgaben zu bekommen. Trotzdem klafft immer noch eine große Lücke zwischen den Möglichkeiten paralleler Architekturen und deren Ausnutzung durch Bildanalyseapplikationen [BBG94, Lüc00]. Versuche, diese Lücke zu schließen, müssen sich mit den folgenden Fragestellungen auseinandersetzen:

- Welche Parallel-Hardware wird eingesetzt? — Spezialkarten, Standard-Multiprozessor-PCs oder Workstations, massivparallele Rechner (MIMD oder SIMD), ...
- Wie werden parallele Programmaspekte modelliert? — explizit, implizit, durch imperative, funktionale oder problemspezifische Sprachen, graphisch, ...
- In welchen Ebenen erfolgt die Parallelisierung? — Daten, Operatoren, Aufgaben, Pipelining, ...
- Welche Instanz ist für die Parallelisierung zuständig? — Anwendungsentwickler, Parallelisierungssystem, Software-Agenten, Betriebssystem, ...

[1] Im folgenden sollen hier im Zusammenhang mit Sensordaten ausschließlich die Begriffe *Folge* und *Sequenz* verwendet werden, da der Begriff *Signal* in den meisten Computersystemen und Programmiersprachen bereits fest belegt ist und zu Mißverständnissen führen könnte.

Speziallösungen vs. Standardsysteme: In der Vergangenheit waren Bildverarbeitungsaufgaben aufgrund ihrer hohen Anforderungen an die Rechenleistung, insbesondere im Bereich der Echtzeit-Bildfolgenanalyse nur mit massiv parallelen Hardwarekomponenten zu bewältigen, und auch heute noch sind spezielle Bildauswertungsgeräte im Einsatz, die zum Teil komplette Bildmatrizen parallel bearbeiten oder aber mehrere Prozessoren für die gleichzeitige Ausführung mehrerer Operatoren besitzen [Mel95]. Der Nachteil dieser Geräte ist, daß ihre Programmierung kompliziert ist, da sie spezielle Verfahren und Algorithmen für die Parallelisierung der Bildverarbeitungsprogramme sowie für die Umsetzung dieser Programme auf die jeweilige Hardware erfordern. Eine Lösung ist i.d.R. auf eine Hardwarekonfiguration beschränkt, was die Wiederverwendung der Software nahezu unmöglich macht. Durch die steigende Verfügbarkeit von leistungsfähigen Standard-PCs und Workstations mit zwei, vier oder mehr Prozessoren (Multiprozessormaschinen) rückt allerdings in jüngster Zeit der Aspekt der Parallelisierung verstärkt auch in den Bereich der Anwendungen, die für Standardsysteme entwickelt werden. Damit wird zunehmend die Frage interessant, wie mit Standardverfahren die Parallelität derartiger Systeme optimal ausgenutzt werden kann.

Darstellung von Nebenläufigkeiten und inhärenter Parallelität: Bildverarbeitungsprogramme besitzen ein hohes Maß an inhärenter Parallelität. Diese ergibt sich einerseits aus den verwendeten Datenstrukturen, bei denen sehr viele Datenelemente (z.B. Pixel oder Bildbereiche) parallel, d.h. quasi gleichzeitig und auf genau die gleiche Art und Weise behandelt werden müssen. Andererseits besteht ein Bildverarbeitungsprogramm i.d.R. aus zahlreichen Operatoren und Teilaufgaben, die z.T. nebenläufig abgearbeitet werden können. Hier interessiert vor allem die Frage einer adäquaten, hardwareunabhängigen Darstellung möglicher Nebenläufigkeiten. Sequentielle Programmiersprachen sind für diese Aufgabe an sich relativ ungeeignet, da sie eine feste Reihenfolge für die Abarbeitung der Operatoren und Befehle vorgeben. Besser geeignet sind dagegen funktionale Sprachen oder graphische Methoden wie Datenflußgraphen oder Petrinetze, die die Reihenfolge für die Berechnung der Ausdrücke nicht festlegen, die Parallelität folglich implizit enthalten. Mit Hilfe deklarativer und objektorientierter Programmkonstrukte lassen sich auch in Sprachen wie C++ funktionalsprachliche Ausdrücke nachbilden und so nebenläufige Programmaspekte darstellen.

Parallelisierungsebenen: Als letzter wichtiger Aspekt soll hier untersucht werden, in welchen Ebenen eine Parallelisierung erfolgen kann. Eine erste Möglichkeit ergibt sich direkt aus der *Datenparallelität* der Bildmatrizen. Eine Bildmatrix läßt sich in einzelne Segmente — bis hin zu Pixelgröße — aufteilen, denen jeweils ein (ggf. virtueller) Prozessor zugewiesen wird. Bei diesem Ansatz führen alle Prozessoren parallel die gleiche Aufgabe auf unterschiedlichen Ausschnitten des gleichen Bildes aus. Realisieren läßt sich dieser Ansatz beispielsweise mit Hilfe einer eigenen parallelen BV-Sprache für die Beschreibung von Matrixoperationen [BFRR95] oder mit Hilfe automatischer Parallelisierer. Diese haben allerdings den gravierenden Nachteil, daß sie i.d.R. auf eine bestimmte Hardware oder einzelne Bildverarbeitungsprobleme eingeschränkt sind [FU94, NJ95], die Anpassung an neue Hardware also dementsprechend teuer wird. Ein anderer, allgemeinerer Ansatz wird in [Lüc00] verfolgt, bei dem das zu bearbeitende Bild in Abhängigkeit von der Anzahl der zur Verfügung stehenden Prozessoren beim Operatoraufruf agentenbasiert in optimale Teilbereiche aufgeteilt und bearbeitet wird. Aus Sicht des Anwendungsprogrammierers hat dies den Vorteil, daß diese Funktionalität vollständig durch die Bildverarbeitungsbibliothek gekapselt wird.

Eine weitere Parallelisierungsebene ergibt sich aus der möglichen Nebenläufigkeit verschiedener, voneinander weitestgehend unabhängiger Teilaufgaben. Eine solche *Aufgabenparallelität* kann auf Operatorebene und auf Programmebene auftreten. In [Lüc00] werden auch hierfür Ansätze für die automatische, agentenbasierte Parallelisierung untersucht.

Als dritte Möglichkeit für eine Parallelisierung kann das *Pipelining*-Konzept genutzt werden, das insbesondere für die Bildfolgenverarbeitung auf Spezialhardware eingesetzt wird. Jeder Prozessor ist dabei für eine bestimmte Teilaufgabe zuständig, wobei die verschiedenen Prozessoren auf unterschiedlichen, nacheinander aufgenommenen Bildern arbeiten. Konzepte für eine automatische Programmierung von Bildauswertungsgeräte, die nach dem Pipelining-Prinzip arbeiten, werden in [Mel95] beschrieben.

Selbstverständlich kann ein System auch gleichzeitig in mehreren dieser Ebenen parallelisiert werden. Für den Anwendungsentwickler sind vor allem die Ansätze interessant, die sich auf Standardhardware effizient umsetzen lassen, problemlos in der verwendeten Programmiersprache ausgedrückt werden können und die keinen oder nur wenig zusätzlichen Programmieraufwand erfordern. Das trifft insbesondere für die auf Operatorebene durchgeführte Daten- und Aufgabenparallelisierung zu, wenn sie durch die verwendeten Bildverarbeitungsoperatoren gekapselt wird.

Des weiteren läßt sich Aufgabenparallelität auch auf Programmebene nutzen, vorausgesetzt das Betriebssystem stellt entsprechende Mechanismen und Hilfsmittel bereit, und die einzelnen Aufgaben können als unabhängige Programmzweige implementiert werden. Die erste Forderung wird von allen modernen Betriebssystemen durch *Prozesse*, *Threads* und *Semaphore* unterstützt. Das Betriebssystem übernimmt dabei die Aufgabe, die einzelnen Zweige eines Programms auf die vorhandenen Prozessoren aufzuteilen. Ein großer Vorteil dieser Form der Parallelisierung ist, daß die geforderte Hardwareunabhängigkeit prinzipiell erhalten bleibt, da die Programmaufteilung unabhängig von den tatsächlich zur Verfügung stehenden Prozessoren ist. Stehen allerdings nur sehr wenige Prozessoren zur Verfügung führt eine zu starke Unterteilung in konkurrierende Programmzweige aufgrund des Overheads beim Prozeßwechsel zu Performanzverlusten.

Die zweite grundlegende Voraussetzung für die Aufgabenparallelität ist die Aufteilung der Programme in entsprechend viele Bearbeitungszweige. Dies stellt im allgemeinen keine triviale Aufgabe dar und bedeutet einen zusätzlichen Aufwand, etwa für die Absicherung kritischer Bereiche oder das Verhindern von Verklemmungen. Darüber hinaus ist die Verwendung rein sequentieller, imperativer Programmiertechniken nur bedingt für die Implementierung nebenläufiger Teilaufgaben geeignet. Neben dem hohen Programmieraufwand bringt eine explizite Spezifikation den Nachteil mit sich, daß sie aufgrund des Overheads nur für eine bestimmte Hardwarekonfiguration optimal ausgelegt werden kann. Ändert sich diese, ist i.d.R. eine erneute Spezifikation notwendig.

Funktionale Sprachen und Datenflußgraphen, sind aufgrund ihrer parallelen Struktur prinzipiell besser geeignet für die Darstellung paralleler Kontrollflüsse. Im Zusammenhang mit Bildverarbeitungssystemen wurden sie bisher jedoch in erster Linie als Grundlage für graphische Programmierwerkzeuge [PL94, KR94, MG95] und zur graphischen Darstellung der Beziehungen zwischen Daten und Operatoren beim Programmentwurf [PH95, Mel95] untersucht. Ist die Anzahl der Operatoren nicht zu groß, bieten derartige Systeme einen guten Überblick über die Funktionen und bestehenden Abhängigkeiten eines Bildverarbeitungsmoduls. Graphische

Programmierwerkzeuge erlauben es, einen Entwurf direkt in ein entsprechendes Programm umzusetzen. Für große Systeme mit unzähligen Operatoren und Parametern wird die graphische Programmierung jedoch von vielen erfahrenen Entwicklern als unübersichtlich und ineffizient abgelehnt und ein textbasierter, sequentieller Entwurf vorgezogen. Damit ein graphischer Systementwurf handhabbar bleibt, sollte daher die Granularität der Darstellung sich nicht zuerst an den zur Verfügung stehenden Operatoren orientieren, sondern von einigen wenigen, aus Sicht der Anwendung signifikanten Daten und Operationen ausgehen. Die Entwicklung dieser Operatoren kann dann nach Belieben textuell oder graphisch erfolgen. Eine äquivalente Darstellungsform zu den Datenflußgraphen findet man in den funktionalen Sprachen, in denen die Beziehungen zwischen Daten und Funktionen mit Hilfe von Gleichungssystemen ausgedrückt werden.

Ein weiteres Ziel dieser Arbeit ist es daher, für relativ abstrakte Teilaufgaben dem Programmentwickler die explizite Spezifikation möglicher Nebenläufigkeiten weitestgehend abzunehmen. Dies soll auf der Grundlage von funktionalsprachlichen Ausdrücken bzw. von Datenflußgraphen erfolgen, die durch die im Rahmen dieser Arbeit entwickelten objektorientierten Datenstrukturen (vgl. Kapitel 7) direkt in die Applikationen eingebunden werden können. Für die Optimierung der nebenläufigen Programmausführung sollen in einer eigenen Beschreibungsebene Mechanismen bereitgestellt werden, mit denen der Parallelisierungsgrad gezielt — auch im laufenden System — beeinflußt werden kann.

1.4.4 Objektorientierte Ansätze in der Bildverarbeitung

Objektorientierte Methoden und Klassenbibliotheken stellen ein wichtiges Mittel für die Bereitstellung von wiederverwendbaren Standardkomponenten dar. Durch Polymorphie, Vererbung und das Kapseln komplexer Verhaltensweisen und Datenstrukturen hinter definierten Schnittstellen erlauben sie es, von Implementierungsdetails zu abstrahieren, was zu übersichtlicheren und leichter handhabbaren Softwareentwürfen führt und somit die Programmentwicklung beschleunigt.

Daten werden in diesem Konzept nicht nur als eine Ansammlung von Werten repräsentiert, sondern als *Objekte* mit fest zugesicherten Eigenschaften, Attributen und speziell auf diesen Datentyp zugeschnittenen Methoden. Repräsentiert werden Objekte als Instanzen einer bestimmten Klasse, welche die allgemeinen Eigenschaften einer Gruppe gleichartiger Objekte beschreibt. Objektklassen und -instanzen können über unterschiedliche Beziehungen und Hierarchien miteinander verbunden sein: *Generalisierung* und *Spezialisierung* stellen die grundlegenden Konzepte beim Aufbau von Klassenhierarchien dar. Allgemeine Objekteigenschaften werden in Basisklassen modelliert. Von diesen werden konkretere Klassen abgeleitet, die die allgemeinen Eigenschaften erben und darüber hinaus verschiedene Spezialisierungen enthalten. Objekte einer abgeleiteten Klasse sind damit auch immer Objekte der Vaterklasse (*is-a*-Beziehung). *Aggregation* beschreibt die Teil-Ganzes-Beziehung zwischen Objekten: Objekte können Bestandteil eines anderen Objekts sein, werden also mit diesem zusammen erzeugt und auch zerstört (*part-of*-Beziehung). Im Gegensatz dazu bezeichnet *Assoziation* eine andere Objektbeziehung, bei der Objekte einander kennen und verwenden oder Nachrichten schicken, die Objekte jedoch unabhängig voneinander existieren.

Für eine effiziente objektorientierte Problemanalyse, Modellierung und Implementierung wurden standardisierte Entwurfsmethoden [Boo94, RBP+91, Sch94], Modellierungssprachen

wie die *Unified Modelling Language* UML [FS97, BJR98] und Werkzeuge [Coa97] entwickelt. Verschiedene komponentenbasierte Ansätze definieren Schnittstellen zwischen weitestgehend unabhängigen Modulen oder Komponenten. Entwurfsmuster (*Design Pattern*) [Pre94, GHJV95, Coa97] beschreiben prototypische Lösungen für immer wiederkehrende Problemstellungen und Aufgaben in objektorientierten Programmen. Als Anwendungsrahmen (*Frameworks*) [JBR99] werden dem Entwickler schließlich Klassen für bestimmte, sich wiederholende Standardaufgaben in Softwareapplikationen zur Verfügung gestellt, die die Architektur der Anwendung oder Teile davon vorgeben.

Auch im Bereich der Bildverarbeitung existieren bereits eine Reihe objektorientierter Ansätze und Systeme, die sich jedoch z.T. erheblich hinsichtlich Umfang und Abstraktionsgrad unterscheiden. In den meisten Fällen werden Klassenbibliotheken für die Repräsentation von ikonischen Daten und den daraus abgeleiteten Bild- und Objektmerkmalen bereitgestellt. Diese sind aufgrund der weiten Verbreitung und der guten Laufzeiteigenschaften i.d.R. in C++ implementiert, für eine wissensbasierte Auswertung oder die Einbindung in bestehende Systeme existieren aber auch Schnittstellen zu anderen objektorientierten Programmiersprachen wie z.B. Smalltalk [KE96].

Unterschiede zwischen den bestehenden Systemen können z.B. bei der Modellierung von Bildern und der mit ihnen assoziierten Semantik beobachtet werden. Einige Systeme betrachten Bilder lediglich als eine spezielle Form von Matrix, leiten folglich die Klasse *Bild* von der Klasse *Matrix* ab [ZB96]. Damit stehen durch Vererbung zwar alle auf Matrizen definierten Operationen auch für Bildmatrizen direkt zur Verfügung, der eigentlichen Semantik von Bildern wird so jedoch nicht entsprochen. Schon relativ einfache Erweiterungen wie mehrere Kanäle pro Bild oder zusätzliche Bildeigenschaften, wie Definitionsbereiche oder die Verknüpfungen mit einem Aufnahmekontext lassen sich mit dieser Modellvorstellung nur auf sehr unnatürliche Weise darstellen.

Die meisten objektorientierten Bildverarbeitungssysteme stellen die Beziehung zwischen Bild und Matrix nicht über Generalisierung, sondern mit Hilfe von Aggregation [Pau92, PH95] oder Assoziation (HALCON/C++) [ES97, Mun93, MKB$^+$95] dar. In ihnen wird zwischen dem Bild und der Matrix unterschieden. Bilder *besitzen* hier neben den auf ihnen definierten Operatoren und verschiedenen Attributen — wie Format, Pixeltyp oder Definitionsbereich — auch eine Matrix, die den Inhalt des Bildes beschreiben. In HALCON ist jedes Bild mit einer oder mehreren Bildmatrizen assoziiert. Eine Matrix ist kein direkter Bestandteil der Bildobjekte und kann somit auch von mehreren Bilder gemeinsam verwendet werden. Der Vorteil dieser Vorgehensweise liegt u.a. im geringeren Speicherplatzbedarf und in der hohen Effizienz von Operatoren, die neue Bildobjekte erzeugen, ohne die Bildmatrix zu verändern (z.B. Zugriff auf einzelne Kanäle, Modifikation des Definitionsbereichs, etc.). Der funktionale Charakter der HALCON-Operatoren — die Operatoren haben keine Seiteneffekte, d.h. die Eingangsdaten bleiben i.d.R. unverändert — garantiert dabei, daß einzelne Matrizen nicht durch andere Operationen unbeabsichtigt modifiziert werden.

Weitere Unterschiede zwischen den verschiedenen Systemen bestehen hinsichtlich Art und Umfang der bereitgestellten Operatoren, hinsichtlich der Modellierung und der Menge zusätzlicher Datentypen, beispielsweise für die Repräsentation von Bildbereichen oder Regionen, Linien und anderen geometrischen Objekten, bis hin zu Nachbarschaften oder Filtermasken, und schließlich hinsichtlich der Unterstützung von 3D-Bildverarbeitung und räumlichen Objektmodellen.

Gemeinsam ist all diesen Systemen, daß sie die Objektorientierung in erster Linie als ein Hilfsmittel für die Kapselung und Abstraktion der von den Operatoren verwendeten Datenstrukturen sowie für die gemeinsame Nutzung von Programmcode verstehen. Man kann diese Vorgehensweise als „*Objektorientierung von unten*“ bezeichnen, in der die Datenabstraktion von der Programmierebene ausgeht. Ausdruck dessen ist, daß die rechnerinterne Repräsentation eines Bildes (als Matrix oder Array, mit konkretem Format und Pixeltyp) stets gleichgesetzt wird mit dem digitalen Bildsignal der Kamera und der optischen Projektion einer Szene in die Bildebene (Abbild, Ansicht).

Eine weitere Einschränkung bestehender Systeme bedeutet das Fehlen von Ausdrucksmitteln, die es erlauben, die gemeinsame Ausgangsbasis der verschiedenen, aus ein und demselben Szenenabbild extrahierten Informationen darzustellen und sie so miteinander in Beziehung zu setzen. Und schließlich gibt es in keinem der untersuchten Systeme Datenstrukturen, mit denen sich Folgen von zeitversetzt aufgenommenen Bildern gemeinsam, als ein Objekt mit einer bestimmten Semantik darstellen und mit ihrem Verarbeitungskontext assoziieren lassen.

Wesentliche der zuvor genannten Einschränkungen bestehender Systeme sollen mit dieser Arbeit überwunden werden. Dafür sind Klassen bereitzustellen, die die verschiedenen Sensordaten und Merkmale eines Verarbeitungszyklus über die Datenmeßzeit mit der Szene und zueinander in Beziehung setzen. Darüber hinaus sollen Folgen von Daten, die über die Zeit anfallen und den gleichen Verarbeitungskontext besitzen, durch jeweils ein Objekt im Programm repräsentiert werden.

1.5 Ziele

In diesem Abschnitt sollen die Ziele der Arbeit zusammengefaßt und konkretisiert werden. Die vielleicht wichtigste Aufgabe, die bei der Konzipierung von großen und komplexen Systemen — zu denen Multisensor-Multiroboterprogramme zweifellos zählen — zu beachten ist, stellt die Unterteilung des Systems in kleinere Module oder Komponenten und die damit verbundene Abstraktion bedeutender Details dar. Die Eigenschaften des Gesamtsystems sollten sich dabei weitestgehend aus den Spezifikationen der Module ableiten lassen, ohne das dafür zusätzliche Informationen über deren innere Struktur notwendig wären [Ost95]. Eine Hauptintention des hier vorgestellten Systems ist es daher, wesentliche Aspekte dynamischer Sensordatenprogramme in separaten, relativ unabhängigen Teilsystemen zu repräsentieren, damit das Gesamtsystem entflechtet und leichter handhabbar wird. Zu den Programmaspekten, denen sich diese Arbeit widmet, gehören die Verwaltung dynamischer Datenfolgen, die ggf. parallele Steuerung zyklischer Datenverarbeitungszweige und die flexible Einbindung und Parametrisierung von Verfahren für die Bereitstellung, Verarbeitung und Auswertung von Sensordaten. Dies soll im Verlauf dieses Abschnitts näher spezifiziert werden.

Zeit und temporale Ausdrücke: Eine elementare Größe des Entwurfs stellt die **Zeit** dar. Ihre explizite Repräsentation ermöglicht es, temporale Restriktionen, die Dynamik der Szene sowie das Zeitverhalten der beteiligten Daten, Sensoren, Objekte und Operatoren auszudrücken. Das Bestimmen der *Gleichzeitigkeit* oder des zeitlichen Versatzes von Daten unterscheidlicher Sensoren und von unterschiedlichen Rechnern — z.B. im Zusammenhang mit Interpolations-

oder Datenfusionsverfahren — ist elementar für viele dynamische Prozesse. So erlauben Zeitwerte die Synchronisation verschiedener parallel ablaufender Programmzweige und der darin verwendeten Daten. Unbedingt erforderlich ist die Datensynchronisation in Verbindung mit bewegten Kameras, wo der korrekte Zusammenhang zwischen Bildaufnahme und Kameraposition elementar für die Lokalisation von Objekten der Szene ist. Sollen in einem verteilten Multirobotersystem die Sensordaten der verschiedenen Roboter ausgetauscht oder gemeinsam genutzt werden, stellt die zeitliche Synchronisation der einzelnen Roboterrechner eine Grundvoraussetzung für die Datenfusion dar. Durch eine Analyse des Zeitverhaltens von Operatoren oder Programmkomponenten wird schließlich die Grundlage gelegt für eine flexible und lastabhängige (Re-) Konfiguration des Systems.

Die verschiedenen angesprochenen temporalen Ausdrucksformen verdeutlichen, daß in der Bildfolgenverarbeitung sowohl qualitative als auch quantitative Zeitausdrücke erforderlich sind. In Kapitel 3 wird eine Notation für quantitative Zeitausdrücke eingeführt. Diese erlaubt zu einem gewissen Grad auch die Ableitung qualitativer Aussagen. Wird jedoch die aussagenlogische Interpretation komplexerer temporaler Ausdrücke gefordert, sollte ein entsprechendes temporallogisches System, wie z.B. die Intervallogik nach [Win94] eingebunden werden.

Bild- und Datenfolgen: Die bei der Bildfolgenanalyse anfallenden Daten sind i.d.R. äußerst umfangreich, weisen implizite Parallelitäten auf und sind komplex miteinander verknüpft. Dadurch reicht eine bloße Erweiterung statischer Datenstrukturen, beispielsweise auf Felder von Einzelbildern für eine adäquate Repräsentation nicht aus. Eine Bildfolge unterscheidet sich von einer Menge von Einzelbildern durch zusätzliche Eigenschaften, die in einer eigenen Objektklassen modelliert werden sollen.

Wichtige Eigenschaften, die alle Bilder einer Bildfolge teilen, sind die zeitliche Einordnung der Bilder in eine Liste, das Wissen um deren Aktualisierungszeitpunkte sowie der — über einen gewissen Zeitraum — für alle Bilder gleiche Kontext, d.h. alle Bilddaten einer Folge werden durch ein und dieselbe Quelle (Sensor oder Methode) bereitgestellt und durch die gleichen Operationen weiterverarbeitet. Das gilt nicht nur für *Bild*-Folgen sondern auch für alle anderen Daten, die zyklisch aktualisiert werden. Mit dem Wissen um den gemeinsamen Kontext lassen sich Mechanismen für die Aktualisierung der Datenfolgen, das Verwalten alter Werte, die transparente Bereitstellung von Elementen einer Sequenz auf anderen Rechnern und die Interpolation fehlender Werte als allgemeine Merkmale *einer* Sequenzklasse modellieren, was natürlicher ist, als diese Eigenschaften für jedes Element und jeden Elementtyp neu zu definieren. Ein allgemeines Datenfolgenmodell stellt damit eine wesentliche Säule des hier vorgestellten Konzepts dar.

Objektorientierte Datenmodelle und Realisierung: Aufgrund der in Abschnitt 1.4.4 genannten Stärken der objektorientierten Programmierung werden die verschiedenen Konzepte für die Repräsentation dynamischer Datenfolgen mit Hilfe objektorientierter Mittel umgesetzt. Für eine effiziente Implementierung wird auf C++ zurückgegriffen, da dies, insbesondere im Bereich der Bildverarbeitung, die am weitesten verbreitete objektorientierte Programmiersprache ist. Die eigentlichen Bildverarbeitungsoperatoren und Objekte zur Repräsentation ikonischer Daten werden durch das Bildverarbeitungssystem HALCON [ES97] bereitgestellt, das in das vorgestellte System eingebettet ist.

Für die Repräsentation der allgemeinen Eigenschaften und Grundverhaltensweisen dynamischer Datenfolgen wird das **Sequenz**-Modell eingeführt. Ein Sequenzobjekt verwaltet eine

Zeitfolge beliebiger Daten, die als **Sequenzwerte** gekapselt sind. Zusammen mit einem Objektmodell für die Repräsentation von Aktionen bzw. Operationen — hier als **Funktoren** bezeichnet, bilden sie das Grundgerüst, mit dem sich komplexe dynamische Zusammenhänge und parallele Programmstrukturen auf natürliche Weise in einer objektorientierten Programmumgebung darstellen lassen.

Ausgangspunkt der Implementierung ist die Beschreibung des Datenflusses zwischen Datensequenzen und Funktoren. Diese kann textuell — in Form eines funktionalen Gleichungssystems — oder graphisch — als Datenflußdiagram — vorliegen. Dabei wird von einer Beschreibung auf einem relativ hohem Abstraktionsniveau ausgegangen, d.h. die im Datenflußdiagramm modellierten Daten und Operatoren gehen von den aus Sicht der Anwendungsdomäne zentralen Daten aus und nicht von der durch das Bildverarbeitungssystem vorgegebenen Granularität. Damit wird eine Brücke geschlagen zwischen rein textuellen, auf imperativen Sprachen basierenden Programmiersystemen und den meist graphischen Systemen, die auf einer funktionalen Beschreibung des Programms beruhen. Die oft als Nachteil empfundene Umständlichkeit der graphischen Programmierung komplexer Programmsequenzen und die Unnatürlichkeit der Darstellung von Programmsteuerkonstrukten wie Schleifen oder Verzweigungen lassen sich durch das hier vorgestellte Konzept vermeiden, da diese Programmteile in die Funktoren hinein verlegt werden, und deren Funktionalität auf herkömmliche Weise, d.h. sequentiell programmiert wird.

Programmsteuerung: Für die Steuerung der Programmkomponenten kommen verschiedene Methoden in Frage. Funktoren können durch neue Eingangsdaten oder aber durch Anfragen bezüglich der Ausgangsdaten angesteuert werden. Beide Methoden werden durch Sequenzen und Funktoren sowie durch spezielle Relationen, die das Zusammenspiel dieser Objekte beschreiben, unterstützt. Mögliche Konflikte und Inkonsistenzen, die durch einen asynchronen Zugriff auf dynamische Daten entstehen können, müssen dabei durch das System erkannt bzw. verhindert werden; Mehrfachaufrufe von Funktoren können durch eine Indizierung der Zugriffe mit der Zeit als Index vermieden werden.

Durch **Agenten** wird ein flexibler Mechanismus für die Programmsteuerung bereitgestellt. So übernehmen sie im vorgestellten System die Rolle einer aktiven Instanz bzw. eines Iteratorobjekts, das kontinuierlich dafür sorgt, daß bestimmte Daten aktualisiert und Funktionen ausgeführt werden. Um zusätzliche Fähigkeiten erweitert, können Agenten weitere Aufgabe übernehmen, z.B. die Überwachung von Systemressourcen und in Abhängigkeit davon die Rekonfiguration des Datenflußgraphen durch den Austausch bestimmter Verfahren oder durch veränderte Verknüpfungen zwischen den Datensequenzen und Funktoren. Für diese Erweiterungen werden durch das Sequenz-Funktor-Modell die Grundlagen gelegt, sie selbst sollen jedoch nicht Gegenstand dieser Arbeit sein.

Die zentrale Aufgabe des Sequenz-Funktor-Agenten-Konzepts ist somit eine möglichst direkte Umsetzung der in der Entwurfsphase aufgestellten Daten- und Steuerflußgraphen in Softwareinstanzen. Der Entwickler wird bei der Implementierung durch applikationsunabhängige Klassen für die Daten- und Operatorverwaltung (u.a. *Sequenz*, *Sequenzwert*, *Funktor* und *Agent*) entlastet, da diese ihm sich ständig wiederholende Standardaufgaben, wie Dateninitialisierung, Datenaustausch oder die Verwaltung temporaler Informationen — etwa für die Bestimmung des Alters, die Neuberechnung oder die Interpolation von Daten — abnehmen. Definierte Schnittstellen für die Einbindung dieser Klassen in die Applikation ermöglichen eine Aufteilung des

Programms, und zwar sowohl bei der Programmentwicklung als auch bei seiner Ausführung in einem verteilten System. Das Zusammenspiel der einzelnen Komponenten kann auf einfache Weise textuell oder aber mit Hilfe graphischer Werkzeuge beschrieben werden. Weiterhin ermöglicht der Entwurf die Kapselung unterschiedlicher Programmaspekte, wie Datenmanagement, Datenverteilung im Netz, nebenläufige Programmsteuerung, Initialisierung, Konfiguration, Bildverarbeitung, Planung und weitere anwendungsspezifische Aufgaben in separate, weitestgehend unabhägige Module, was die Gesamtkomplexität reduziert. Für die Entwicklung einzelner Bildverarbeitungskomponenten muß man lediglich die Schnittstellen zu den relevanten Daten kennen. Nicht zu interessieren brauchen dagegen Fragen wie: „Wie werden die berechneten Ergebnisse auf andere Rechner übertragen?“, „Mit welchen Methoden und wie oft erfolgt die Bereitstellung der Eingangsdaten einer Komponente?“ oder „In welcher Reihenfolge müssen die einzelnen Komponenten aufgerufen werden?“.

In Systemen, die in einer realen Umgebung agieren, spielen Sensoren für die Datenaufnahme eine wichtige Rolle. Das hier vorgestellte Konzept erlaubt es, Sensoren systemkonform ohne zusätzlichen Aufwand darzustellen. Sie werden ebenfalls als Funktoren modelliert, wodurch sie sich nahtlos in das System einfügen. Da der Zugriff auf sie identisch ist zu dem auf reguläre Funktoren, lassen sich physikalische Sensoren einfach durch simulierte oder logische Sensoren ersetzen. Darüber hinaus bietet dieser Ansatz die Möglichkeit, beliebige Datensequenzen als Sensordaten — und demzufolge die sie bereitstellenden Funktoren oder Datenverarbeitungsmodule als logische Sensoren — anzusehen.

> **Das Ziel dieser Forschungsarbeit war die Entwicklung und Implementierung eines Softwaresystems, mit dem auf Basis adäquater objektorientierter Datenmodelle für Zeit und zeitliche Ausdrücke, für Sensoren und Operatoren sowie für dynamische Daten und aktive Programmkomponenten die effiziente Entwicklung flexibler, robuster und paralleler Programme für die Echtzeit-Bildfolgenauswertung wesentlich verbessert werden kann.**
>
> **Dieses Ziel wurde durch die Realisierung einer C++-Klassenbibliothek, deren zentrale Objekte das Grundgerüst für die zu entwickelnden Anwendungen bilden, umgesetzt. Entwurfsmuster unterstützen den Entwickler bei der direkten Übertragung der funktionalen Problembeschreibungen in komplexe Bildfolgenprogramme und bilden die Grundlage für eine Automatisierung dieser Aufgabe.**

Neu an diesem Ansatz ist, daß mit wenigen, einfach zu verwendenden und effizient implementierten Klassen ein mächtiges Gerüst für die Entwicklung von Bildfolgenprogrammen auf hohem Abstraktionsniveau bereitgestellt wird. Diese Klassen realisieren allgemeine, sich häufig wiederholende Standardaufgaben im Zusammenhang mit dynamischen Sensordaten. Dazu gehören beispielsweise deren Initialisierung und Aktualisierung, die Verwaltung der Historie, Interpolationsmechanismen sowie die Visualisierung, Protokollierung und Verteilung der Daten im Netz. Dadurch entlastet dieser Ansatz den Entwickler und läßt ihm Zeit, sich mit den eigentlichen Problemen der jeweiligen Anwendungsdomäne zu beschäftigen. Er ermöglicht durch ein einheitliches, typunabhägiges Sensor- und Datenfolgenmodell die klare Trennung verschiedener Programmebenen, wie der Sensordatenbereitstellung, die Datenauswertung und die Programmsteuerung. Durch die explizite Modellierung von Zeit und dem Zeitverhalten aller beteiligten Objekte wird eine korrekte zeitliche Einordnung dynamischer Daten und deren

Zuordnung zueinander, selbst wenn sie von unterschiedlichen Sensoren stammen, ermöglicht. Darüber hinaus wird so die Grundlage geschaffen für eine dynamische Performanzanalyse und Rekonfiguration der Anwendung, die z.B. mit Hilfe von Agenten entsprechend vorgegebener temporaler Restriktionen erfolgen kann.

1.6 Randbedingungen und Testumgebung

Um die Praxistauglichkeit der Konzepte unter Beweis zu stellen, wurden die wesentlichen Komponenten in Form einer C++-Klassenbibliothek implementiert und in einer exemplarischen Anwendung eingesetzt. Als Beispielumgebung wurde das RoboCup-Szenario ausgewählt, bei dem mehrere autonome Roboter eine Mannschaft bilden und gegen ein ähnlich aufgebautes gegnerisches Team Fußball spielen [KAK+97a, KAK+97b].

Dieses Szenario wird in Kapitel 2 detaillierter vorgestellt. Es ist durch zahlreiche, nebenläufig zu bearbeitende Aufgaben, durch die Kombination mehrere Sensoren und die Kooperation mehrerer Roboter und Softwareagenten gekennzeichnet. Eine der zentralen Aufgaben ist die Sensordaten- und Bildfolgenanalyse und die Interpretation der extrahierten Daten.

Die in den Abschnitten 1.2 und 1.3 aufgeführten Merkmale von Echtzeit-Bildfolgenprogrammen gelten uneingeschränkt auch im RoboCup-Szenario. Dabei soll hier auf harte Zeitrestriktionen verzichtet werden, da dies die Unterstützung durch ein Echtzeitbetriebssystem erfordern würde. Ein Onlinesystem mit weichen Zeitrestriktionen kann als ausreichend angesehen werden, da beim RoboCup keine gefährlichen Zustände auftreten und eine hohe Wahrscheinlichkeit für das Einhalten der gewünschten Wiederhol- und Reaktionszeiten als Zusicherung genügt. Die Wohldefiniertheit der Umgebung (beispielsweise wird das Spielfeld von einer überwiegend weißen Wand begrenzt, und die Farben aller relevanten Objekte sind vorgeschrieben und bekannt) stellt eine gewisse Vereinfachung für die Entwicklung der Bildanalysemethoden dar. Auf den Systemaufbau und die Systemkomplexität hat dies jedoch kaum Einfluß.

Mit Hilfe des hier vorgestellten Konzepts und der in diesem Rahmen entwickelten Klassenbibliothek gelang es in relativ kurzer Zeit, aufbauend auf einem Pool von Pioneer-1-Robotern [Act99] ein Multiagentensystem zu entwickeln, das in verschiedenen Wettbewerben (RoboCup-98 [KLZ+99], Vision-98, RoboCup-99 [BHKS99, BKHS99] und Vision-99) seine Funktionsfähigkeit unter Beweis stellen konnte. Bei einer im Vergleich zu anderen Teams relativ hohen Bildauflösung von 384×172 Pixel wird eine für eine reine Softwarelösung beachtliche Bildrate von 10 – 15 Farbbildern pro Sekunde (inklusive der Selbstlokalisation) erreicht, was die Echtzeittauglichkeit und den geringen Overhead des vorgestellten Systems demonstriert. Dieser liegt selbst auf den verwendeten älteren Pentium-Pro-Systemen weit unter den für die Bildverarbeitung benötigten Rechenzeiten und kann daher vernachlässigt werden.

Kapitel 2

Anwendungsszenario Roboterfußball: Der RoboCup

Da der RoboCup für die in dieser Arbeit vorgestellten Konzepte das wichtigste Testszenario darstellt, soll in diesem Kapitel ein relativ detaillierter Überblick über die noch junge *Robot World Cup Initiative* (kurz *RoboCup*) gegeben werden. Ziel dieser Initiative ist es, durch die Vorgabe eines Standardproblems die Forschungen im Bereich der Künstlichen Intelligenz und intelligenter mobiler Roboter zu fördern. Der Roboterfußball steht dabei stellvertretend für eine noch relativ neue Klasse von Applikationen: Multisensor-, Multiroboter- und Multiagenten-Systeme, die in einer komplexen, dynamischen und z.T. destruktiven Umgebung gemeinsam eine anspruchsvolle Aufgabe zu erfüllen haben.

Dieser Überblick stellt die Grundideen, Forschungsschwerpunkte und wichtigsten Regeln des RoboCups vor. Dabei sollen die für diese Arbeit relevanten Aspekte hervorgehoben werden, ohne sich auf diese zu beschränken.

2.1 Einführung

Unter dem Namen *Robot World Cup Initiative* wurde 1993 in Japan eine Initiative ins Leben gerufen, die zum Ziel hat, Forschungen im Bereich der Künstlichen Intelligenz und intelligenter mobiler Roboter durch die Vorgabe eines Standardproblems zu fördern. Als Standardproblem wurde der *Roboterfußball* gewählt, da Fußball zum einen als Herausforderung für die KI und Robotik unmittelbar zuvor von einigen Forschungsgruppen ins Gespräch gebracht worden war (z.B. [Mac93, Sah93]), zum anderen verlangt der Roboterfußball die Integration zahlreicher sehr unterschiedlicher Verfahren und die Kombination verschiedenster Fähigkeiten, wodurch sich dieses Szenario sehr nah an den komplexen Anforderungen der Realität bewegt, ohne sich jedoch auf eine spezielle Anwendung zu beschränken. Ein weiterer Grund für Fußball als Standardszenario ist sicherlich die Popularität dieser Sportart und die dadurch erhoffte Außenwirkung, die der Forschung das Interesse einer breiten Öffentlichkeit und potentieller Sponsoren näher bringen soll. Die Initiative fand ein derart breites Echo, daß 1997 in Nagoya/Japan die erste Weltmeisterschaft abgehalten wurde, und sich der RoboCup inzwischen als ein anerkanntes Standardproblem etablieren konnte.

Neben dem Beherrschen der verschiedenen Einzeltechniken, wie z.B. Bildfolgenverarbeitung, Sensorik, Datenfusion, Echtzeitschließen, Aktionsplanung, Strategieauswahl, Robotik, Multiagenten-Kooperation und Maschinelles Lernen stellt der RoboCup aufgrund der Dynamik, Komplexität, Verteiltheit und Parallelität spezielle Anforderungen an die Systemarchitektur. Die Klasse der mittelgroßen, weitestgehend autonom arbeitenden Roboter in der *Middle-Size League* stellt z.Zt. die höchsten Anforderungen an den Systementwurf und die dem System zugrundeliegenden Softwarekonzepte. Aus diesem Grund eignet sich diese Klasse besonders als Testszenario für die in dieser Arbeit entwickelten Konzepte.

2.2 RoboCup als Standardproblem für KI und Robotik

Die RoboCup-Initiatoren sehen im Roboterfußball ein neues Standardproblem für die Künstliche Intelligenz und Robotik. Insbesondere in der KI stellen Standardprobleme eine treibende Kraft für die Forschung dar, da sie systematische Forschungen und den Vergleich unterschiedlicher Ansätze besser unterstützen, als z.B. kommerzielle Anwendungen. Computerschach stellte das bisher wohl bekannteste Standardproblem der KI dar. Ihm sind eine Reihe neuer und effizienter Suchverfahren zu verdanken. Eine häufig vorgetragene Kritik, Standardprobleme seien zu abstrakt und ignorierten wesentliche Schwierigkeiten der Realität, wird durch die Übertragbarkeit der entwickelten Algorithmen auf reale Anwendungen entschärft; trotzdem bleibt die Forderung, daß neue Forschungsaufgaben sich verstärkt an der Realität ausrichten müssen.

	Computerschach	**RoboCup**
Umgebung	statisch	dynamisch
Zustandswechsel	abwechselnd	kontinuierlich in Echtzeit
Information	vollständig und sicher	unvollständig und unsicher
Sensordaten	symbolisch	nicht symbolisch
Steuerung	zentral	dezentral

Tabelle 2.1: Vergleich von Computerschach und RoboCup [KAK+97b]

Aus diesem Grund ist der RoboCup sehr gut als neues Standardproblem geeignet: Er orientiert sich stark an der Komplexität der Realität, auch wenn (vorläufig noch) gewisse Einschränkungen in Kauf genommen werden. Wesentliche Unterschiede zum Computerschach werden in Tabelle 2.1 aufgelistet. Die Problemgröße ist so gewählt, daß sie auf der einen Seite mit vertretbaren Kosten und Aufwand bearbeitet werden kann, auf der anderen Seite aber skalierbar ist, so daß auch für die kommenden Jahre noch genügend Forschungspotential offen bleibt. Ausdruck dessen sind die verschiedenen Ziele des RoboCups: Letztes und oberstes Ziel ist, mit einem Roboterteam nach den bestehenden FIFA-Regeln[1] gegen den amtierenden Weltmeister zu gewinnen, was ausgehend vom heutigen Stand der Technik praktisch unmöglich erscheint und sicherlich auf lange Zeit noch eine große Herausforderung darstellen wird. Der RoboCup, wie er sich heute präsentiert, mit verschiedenen Roboterklassen und reinen Softwareprogrammen, definiert dagegen eine Landmarke auf dem Weg dahin.

[1] Die FIFA (**F**édération **I**nternationale de **F**ootball **A**ssociation) ist der internationale Fußballverband.

Dabei soll der RoboCup nicht Selbstzweck sein und auch nicht zu einem reinen Medienereignis mit einem gewissen Unterhaltungswert werden. Primäres Ziel ist vielmehr die Entwicklung neuer Theorien, Algorithmen und Systeme in den verschiedenen Forschungsbereichen sowie deren Integration zu komplexen Multiagentensystemen. Mögliche Anwendungsgebiete finden sich z.B. in der Raumfahrt, wie etwa in der *Deep Space Mission* — einem NASA-Projekt zur Erkundung der äußeren Bereiche des Weltraums [PBC+97] — oder bei der Erkundung der Mars-Oberfläche durch eine Gruppe von Robotern [KANM98].

Noch offensichtlicher wird die wirtschaftliche und gesellschaftliche Bedeutung mobiler Roboter bei der Suche und Rettung von Menschen in und nach Katastrophen, wie z.B. Erdbeben. Roboter-Teams könnten Trümmer systematisch nach Opfern durchsuchen, wobei sie mit ähnlichen Problemen wie im RoboCup konfrontiert werden: Mehrere autonome Agenten haben eine gemeinsame Aufgabe zu erfüllen, wofür sie sich koordinieren müssen. Die lokal verfügbaren Informationen sind unvollständig und ungenau, sie können zum Teil widersprüchlich oder sogar falsch sein und müssen trotzdem zu einem möglichst konsistenten Umgebungsbild zusammen geführt werden. Auf erwartete wie auf unerwartete Ereignisse muß angemessen reagiert werden. Einzelne Roboter können ausfallen, was im laufenden Einsatz u.U. eine Neuplanung, Strategieänderungen und die dynamische Neuverteilung von Teilaufgaben erforderlich macht. Darüber hinaus müssen so elementare Aufgaben wie die Selbst- und die Objektlokalisation, koordinierte Bewegungen und die genaue zeitliche Abstimmung von Aktionen beherrscht werden. Weiterer Forschungsbedarf ergibt sich bei der Entwicklung von universellen und problemangepaßten Hardwarekomponenten, wie Sensoren und Aktoren.

2.3 Aktivitäten, Klassen und Regeln im RoboCup

2.3.1 Die RoboCup-Klassen

Die Forschungsaktivitäten und Wettbewerbe im RoboCup erfolgen in verschiedenen Klassen bzw. Ligen. Die Simulatorliga stellt eine reine Softwareklasse dar. Klassen mit realen Robotern gibt es gegenwärtig drei: kleine Roboter auf Rädern (*Small-Size League*), mittlere, ebenfalls mit Räder ausgestattete Roboter (*Middle-Size League*) und kleine, vierbeinige Roboter (*Legged-Robot League*). Die angegebenen Regeln spiegeln den Stand von 1999 wieder. Sie werden ständig weiterentwickelt, wodurch auch auf neue Forschungsergebnisse reagiert werden soll.

Simulatorliga: Die Simulatorliga wird in Form einer Client-Server-Anwendung betrieben. Jede Mannschaft besteht aus elf unabhängigen Programmen, den Softwareagenten. Diese stellen die Clients dar und kommunizieren über definierte Kanäle mit dem Fußball-Server (*Soccer Server*). Von dem erhalten sie auf Anfrage die Relativpositionen anderer Roboter, des Balls und von Landmarken auf dem Spielfeld, insofern sie sichtbar sind. Dabei werden die Daten vom Server mit einem entfernungsabhängigen, zufälligen Fehler versehen. In einem bestimmten Zeitraum kann nur eine begrenze Datenmenge abgefragt werden. Der Server arbeitet unabhängig von den Spielerprogrammen, d.h. nach einer Anfrage an den Server bewegen sich Ball und Spieler weiter und die Daten veralten.

Jeder Client-Prozeß darf nur einen Spieler steuern, und die Spieler können nicht direkt miteinander kommunizieren, sondern müssen die vom Server bereitgestellten Möglichkeiten ver-

wenden. Die Forschungsschwerpunkte der Simulatorklasse liegen vor allem in den Bereichen Aktionsplanung, Strategieakquisition, Agentenkooperation, Verhaltensmodellierung und Echtzeitanimation.

Small-Size League — F180: Die kleinen Roboter haben eine Grundfläche von ca. 180 cm^2, was einem Durchmesser von etwa 15 cm entspricht. Das Spielfeld hat die Größe einer Tischtennisplatte (ca. 152 cm $\times$ 274 cm), der Ball die eines Golfballs. Über dem Spielfeld ist eine Kamera montiert, die die Steuerprogramme der Roboter mit den notwendigen Bilddaten versorgt. Definierte Farben für das Spielfeld und den Ball sowie farbige Marker auf den Robotern helfen den Programmen bei der Bestimmung der Positionen der verschiedenen Objekte auf dem Spielfeld.

Alle Roboter einer Manschaft werden i.d.R. von einem Programm auf einem externen Rechner gesteuert, nur die reinen Fahrbefehle werden über eine Funkverbindung zum Roboter übertragen. Für die Forschung ist hier neben der Planung von Spielzügen vor allem das Beherrschen der Dynamik der Aktionen (software- und regelungstechnisch) interessant, da die Roboter relativ schnell sind.

Middle-Size League — F2000: Da die Konzepte und Verfahren, die in dieser Arbeit vorgestellt werden, auf einem System der *Middle-Size League* umgesetzt und getestet wurden und sich verschiedene Beispiele auf dieses Szenario beziehen, soll dieser Klasse etwas mehr Raum bei der Beschreibung eingeräumt werden. Abgesehen von der *Legged-Robot League,* die sich noch in einem sehr frühen Teststadium befindet, stellt diese Klasse derzeit die mit den komplexesten Systemanforderungen dar. Im Gegensatz zur Simulatorliga müssen hier reale Roboter in einer realen Welt beherrscht werden, und im Vergleich zur *Small-Size League* sind die einzelnen Roboter wirklich autonom und nur auf ihre lokalen Sensoren angewiesen.

Objekt	**Farbe**
Ball	Rot
Tore	Blau bzw. gelb
Spielfeld	Grün
Linien	Weiß
Bande	Weiß mit schwarzer Schrift
Roboter	Schwarz mit Markern in Zyan bzw. Magenta

Tabelle 2.2: Farben in der *Middle-Size League*

Die Roboter der *Middle-Size League* haben eine Grundfläche von 2000 cm^2, was einem Durchmesser von etwa 50 cm entspricht. Die gesamte Sensordatenverarbeitung, Planung und Robotersteuerung läuft lokal auf den Roboterrechnern ab. Als Sensoren kommen vor allem Kameras, Laserrangefinder sowie taktile und Ultraschallsensoren zum Einsatz. Um die Bildverarbeitung etwas einfacher zu gestalten, haben alle relevanten Objekte definierte Farben (vgl. Tabelle 2.2). Im Gegensatz zur *Small-Size League* sind globale Kameras nicht zugelassen, d.h. den Robotern steht a-priori nur die durch die lokalen Sensoren gelieferte, relativ eingeschränkte Sicht auf die Szene zur Verfügung. Allerdings können diese Informationen ausgetauscht und zu einer „globalen“ Weltsicht fusioniert werden. In Anbetracht der Unsicherheiten bei der Objekterkennung, insbesondere aber bei der Bestimmung der eigenen Position, ist dies jedoch

keineswegs trivial und stellt eine der großen Herausforderungen im Bereich der verteilten Sensordatenverarbeitung dar.

Bei einem Spiel stehen sich maximal 4 Roboter je Team gegenüber. Das Ziel ist — wie im richtigen Fußball — den 20 cm großen Ball (ein regulärer Jugendfußball) in das gegnerische Tor zu spielen. Das Spiel dauert 2×10 Minuten, das Spielfeld ist $4{,}5\,\mathrm{m} \times 9\,\mathrm{m}$ groß, von einer 50 cm hohen Bande umgeben und die Tore sind 2 m breit. Viel diskutiert ist der Einsatz und die Gestaltung der Bande als Spielfeldbegrenzung. Nachdem sie beim RoboCup-99 zum ersten Mal mit schwarzen Logos versehen war, sind für die nähere Zukunft farbige Logos und später ihr vollständiges Entfernen im Gespräch, was deutlich höhere Anforderungen an die Bildverarbeitung und die Selbstlokalisationsverfahren stellen wird.

Durch Linien am Boden wird die Spielfeldmitte und ein $1\,\mathrm{m} \times 3\,\mathrm{m}$ großer Strafraum markiert. In diesem dürfen sich die Roboter — vom Torwart abgesehen — nur kurze Zeit aufhalten, eine Regel, deren Einhaltung zumindest eine vage Bestimmung der eigenen Position auf dem Spielfeld erfordert.

Eine weitere wichtige Regel verbietet das Attackieren anderer Roboter. Dies erfordert eine funktionierende Hindernisvermeidung, was das Erkennen anderer Robotern sowie deren Positionsbestimmung voraussetzt. Die Bewegungen der Roboter, einschließlich des beobachtenden, verschärfen das Problem noch. Je schneller die Roboter sich bewegen, umso wichtiger ist es zum einen, für die absolute Lokalisation der Objekte die eigene Position während der Bildaufnahme genau zu kennen, was einen Angleich der unterschiedlichen Zeitbasen von Odometrie- und Kameradaten erfordert, und zum anderen die Bewegungen anderer Roboter zu prädizieren.

Diese Beispiele zeigen bereits die Notwendigkeit einer genauen Modellierung zeitlicher Zusammenhänge. Aus den Anforderungen, die ein verteiltes dynamisches Robotersystems an den Systementwurf stellt, ergeben sich weitere Anknüpfungspunkte zu der hier vorgestellten Arbeit.

Legged-Robot League: Die *Legged-Robot League* ist die jüngste RoboCup-Klasse. Während des RoboCup-98 in Paris sind die vierbeinigen Roboter (mit Kopf und Schwanz) von Sony in einer Ausstellungsrunde erstmals gezeigt worden. Beim RoboCup-99 wurde bereits ein Wettbewerb mit acht Mannschaften gespielt.

Die hohe Anzahl an Freiheitsgraden eröffnet viele neue Möglichkeiten für die Bewegung der Roboter und die Manipulation des Balles. Allerdings steigt die Komplexität auch deutlich an, und die Steuerung einfacher Bewegungen erfordert einen wesentlich höheren Aufwand. Dies erklärt auch, warum zum gegenwärtigen Zeitpunkt die Roboter noch in erster Linie mit sich selbst beschäftigt sind und elementare Bewegungsabläufe wie Aufstehen, Laufen und Drehen einüben.

Für die Zukunft ist schließlich auch eine Liga für zweibeinige, menschenähnliche Roboter geplant. Dabei soll zwischen autonomen und ferngesteuerten Robotern unterschieden werden. Ein erstes Spiel in dieser Klasse wurde von der Firma Honda, die einen zweibeinigen humanoiden Roboter bereits vorstellen konnte, für den RoboCup-2002 angekündigt.

2.3.2 RoboCup-Aktivitäten

Die RoboCup-Initiative ist als ein umfangreiches Programm angelegt, zu der neben dem eigentlichen Roboter-Fußball verschiedene weitere Aktivitäten gehören:

Wettbewerbe: Am bekanntesten sind die RoboCup-Wettbewerbe, insbesondere die Weltmeisterschaften: 1997 in Nagoya, 1998 in Paris, 1999 in Stockholm, 2000 in Melbourne. In Turnierform wird in verschiedenen Ligen der Gewinner ermittelt. Etabliert haben sich bisher eine Simulatorliga und verschiedene Ligen mit realen Robotern unterschiedlicher technischer Spezifikationen (vgl. Abschnitt 2.3.1). Ein großer Vorteil von Wettbewerben ist, daß durch sie der Bau von *zuverlässigen* Systemen gefördert wird, und ein Erfolg bei ihnen die Kombination von elementaren Fähigkeiten aus allen Forschungsbereichen voraussetzt.

Die Wettbewerbspraxis bringt allerdings auch einige Nachteile mit sich. In den Medien wird i.d.R. stark auf die Spielergebnisse und das Ziel „Turniersieg“ fokussiert, was den Forschungsleistungen der beteiligten Institute oft nicht gerecht wird. Spiel- und Wettbewerbsgewinn erlauben zwar Rückschlüsse auf die Leistungsfähigkeit eines Gesamtsystems, nicht jedoch den Vergleich einzelner Algorithmen, Lösungsansätze oder Teilsysteme und erst recht nicht der Qualität der Forschungsaktivitäten der beteiligten Institute. Darüber hinaus führt der Wunsch, in den Turnieren möglichst gut abzuschneiden, dazu, daß ein recht hoher Aufwand auch in wissenschaftlich weniger interessante Ad-hoc-Lösungen oder Randbereiche gesteckt wird. Schließlich stellen Wettbewerbe für viele Forschungseinrichtungen eine erhebliche finanzielle (und organisatorische) Belastung dar.

Konferenzen: Die RoboCup-Wettbewerbe sind i.d.R. an wissenschaftliche Konferenzen gekoppelt. Diese werden meist in Form spezieller Workshops im Rahmen einer größeren Konferenz (IJCAI, ICMAS, IROS, PRICAI) durchgeführt. Sie stellen das wichtigste Forum für den Austausch der Forschungsergebnisse dar.

RoboCup ‘Challenges’: Mit abgeschlossenen und überschaubaren Teilaufgaben sollen vor allem zwei Dinge erreicht werden: Erstens werden kurzfristige Forschungsziele vorgegeben und zweitens Ergebnisse vergleichbar gemacht, die im Wettbewerb nur schwer zu messen wären. Aufgaben an reale Roboter beinhalten das Führen, Annehmen und Passen des Balls. Auch der Strafstoß, also das Spiel ein Angreifer gegen den Torwart, stellt eine solche technische Aufgabe dar.

Für die Simulator-Klasse werden ähnliche Aufgaben vorgegeben. Diese adressieren insbesondere die Lernfähigkeit, die Kooperation im Team und die Modellierung von Verhaltensweisen des Gegners [K+97]. Ein weiterer Aufgabenkomplex hat die Generierung natürlichsprachlicher Ausgaben zum Ziel, beispielsweise für automatische Spielkommentare in der Simulatorklasse.

Bildung: Im Rahmen des RoboCup ist ein spezielles Bildungsprogramm integriert. Zum einen umfaßt es projektorientierte Kurse für Schüler und Studenten. Zum anderen sollen unter dem Stichwort *RoboCup Jr.* neueste Erkenntnisse aus den Bereichen KI und Robotik einem breiten Publikum bekannt gemacht werden. Insbesondere Kindern soll unter Zuhilfenahme preiswerterer Technik und einfacherer Aufgaben spielerisch der Zugang zu Robotern, zu deren Bau und Programmierung sowie zur KI ermöglicht werden.

Infrastruktur: Die Bestrebungen nach einer einheitlichen und standardisierten Infrastruktur betrafen bisher in erster Linie die Simulatorliga, da dort die Programme der beteiligten Teams auf Serverprogramme für die Simulation und Visualisierung des Spiels zugreifen. Es gibt aber auch Überlegungen, Standards für die Roboterhardware festzulegen bzw. eine einheitliche Hardwareplattform bereitzustellen. Dafür werden besonders zwei Gründe angeführt:

Zum einen ist der Aufbau von verläßlichen Robotern und deren Low-level-Ansteuerung sehr aufwendig und für die meisten Teams ohne besondere Forschungsrelevanz, die dafür notwendigen Mittel sollten daher verstärkt in die Sensordatenauswertung, High-level-Steuerung und Planung fließen. Zum anderen könnte eine Standardplattform mit Standardkomponenten den Preis für die Anschaffung eines RoboCup-Teams deutlich verringern, was vor allem dem Bildungsprogramm zugute käme.

Nebendomänen: Das wichtigste Ziel des RoboCups ist, die Forschung in der Robotik, KI und einer Reihe weiterer Gebiete voranzubringen. Daher ist es eine wichtige Aufgabe, neue Verfahren nicht nur im Roboterfußball sondern auch in anderen Szenarien zu testen, zum einen um die verschiedenen Komponenten unter anderen dynamischen Anforderungen oder bei veränderten Randbedingungen zu testen, zum anderen um Merkmale und Anforderungen, die im Roboterfußball keine Rolle spielen, ebenfalls unter realen Bedingungen untersuchen zu können. Solche Neben- oder Sekundärdomänen sind z.B. Such- und Rettungsszenarien, die Erkundung gefährlicher und unzugänglicher Gebiete oder Roboterteams in Büroumgebungen.

2.4 Die wissenschaftlichen Herausforderungen des Roboterfußballs

Für die Entwicklung eines funktionierenden RoboCup-Teams ist es notwendig, verschiedene Forschungsdisziplinen wie Künstliche Intelligenz, Robotik, Sensorik und Multiagentensysteme zusammenzuführen sowie deren Ergebnisse zu kombinieren und in ein komplexes Gesamtsystem zu integrieren. Neben den zahlreichen Aufgaben und wissenschaftlichen Herausforderungen, die jedes dieser Gebiete für sich bereithält, ergeben sich eine Reihe neuer Probleme aus der Kombination unterschiedlicher Verfahren und Disziplinen. Die wichtigsten Herausforderungen und Problemfelder sollen in den folgenden Abschnitten näher beleuchtet werden.

2.4.1 Entwicklung, Bau und Steuerung der Roboterhardware

Beim Bau autonomer Roboter sind noch eine Reihe ungelöster Aufgaben und Probleme zu bewältigen. Das betrifft als erstes die eigentliche Roboterplattform sowie die Entwicklung verschiedener Aktoren für die unterschiedlichen Handlungen, die die Roboter ausführen müssen. Der RoboCup wird gegenwärtig (noch) von Robotern auf Rädern dominiert. Hier haben unter anderem die Geschwindigkeit und die Exaktheit, mit der die Bewegungen des Roboters erfolgen können, bedeutenden Einfluß auf den Erfolg einer Mannschaft. Neue omnidirektionale Antriebe, wie z.B. die Entwicklungen von RMIT[2] [PJK98] oder Uttori-United[3] [YOM+98], können deutlich die Manövrierfähigkeit der Roboter verbessern, da diese es ihnen ermöglichen, direkt und ohne zu drehen oder zu rangieren in jede beliebige Richtung zu fahren. Die Bedeutung von Beinen wird aufgrund deren universelleren Einsetzbarkeit und der Ausrichtung am echten Fußball in der Robotik- und RoboCup-Forschung zunehmen, wie das Interesse an der *Legged-Robot League* beim RoboCup-99 in Stockholm zeigte.

[2] RMIT-Raiders — *Middle-Size*-Team vom Royal Melbourne Institute of Technology, Australien

[3] Gemeinschaftsteam von Riken, Tokyo University und Utsunomiya University, Japan

Neben universell einsetzbaren, hand- oder fußähnlichen Aktoren spielen auch Spezialgeräte für einzelne Tätigkeiten eine wichtige Rolle. Im RoboCup stellt eine Schußvorrichtung („Kicker“) ein solches Gerät dar, in anderen Einsatzgebieten können dies Werkzeuge oder Hebevorrichtungen sein. Wesentliche Herausforderungen ergeben sich aus den Randbedingungen, unter denen die Aktoren eingesetzt werden sollen. Sie können z.B. Größe, Gewicht, Stabilität, Energiebedarf sowie Anschaffungs- bzw. Entwicklungskosten betreffen.

Als weiteres Ziel sind verschiedene Aktoren und Sensoren so miteinander zu kombinieren, daß kompakte Gesamtgeräte mit einem möglichst modularem Aufbau entstehen. Bestimmte Komponenten können mit eigener Intelligenz ausgestattet sein, wodurch sich Aufgaben leichter parallelisieren lassen. Dies erfordert aber auch spezielle Synchronisationsmechanismen. Eine neue Klasse von Aktoren, die durch die Kombination einfacher Aktoren mit taktilen und visuellen Sensoren entstehen, könnte in der Lage sein, „gefühlvoll“ zu agieren, beispielsweise um im RoboCup den Ball zu führen oder zu dribbeln, oder — in anderen Anwendungsszenarien — um zerbrechliche Dinge aufzusammeln.

Ebenfalls noch ungelöst ist die Frage einer dauerhaften, zuverlässigen, leichten und preiswerten Energieversorgung. Diagnosesysteme schließlich können die Spannungsversorgung aber auch den Zustand oder die Funktionsfähigkeit einzelner Komponenten überwachen und diese Informationen der Planung zur Verfügung stellen, beispielsweise um Roboter, die nur noch eingeschränkt handlungsfähig sind, an einer geeigneten Position optimal einzusetzen, oder um bei sinkender Spannung auf energieintensive Aktionen zu verzichten bzw. aktiv einen Batteriewechsel zu veranlassen.

2.4.2 Informationsakquisition mit Hilfe von Sensoren

Eine der notwendigen Basisfähigkeiten mobiler Roboter ist die Aufnahme von Daten aus der Roboterumgebung. Diese Daten stellen die wichtigste Handlungsgrundlage für den Roboter dar. Sie sind Ausdruck des aktuellen Zustands des Szene. Hierbei können verschiedene Sensoren zum Einsatz kommen, wobei Kameras wegen der großen Informationsdichte eine herausragende Rolle spielen. Darüber hinaus werden im RoboCup Ultraschall- und taktile Sensoren sowie verstärkt Laserrangefinder verwendet. Die Auswertung von Videobildfolgen stellt den interessantesten Teilbereich der Sensordatenverarbeitung dar, da Kameras die informationsreichsten und am universellsten einsetzbaren, sicherlich aber auch die am schwierigsten auszuwertenden Sensoren sind.

Die mit Hilfe der Sensordatenverarbeitung aus der Umwelt gewonnenen Informationen können in verschiedenen Formen vorliegen:

geometrisch: eigene Position, Position relevanter Objekte (Ball, Gegner, Mitspieler, Tore), relativ zum Roboter und absolut,

topologisch: z.B. „Ball ist rechts“, „gegnerisches Tor ist links“,

symbolisch: z.B. „Ball und eigenes Tor sind sichtbar“, „Weg ist durch Gegner blockiert“.

Hinsichtlich der Genauigkeit, Sicherheit und Vollständigkeit der gewonnenen Informationen kann es je nach Aufgabengebiet unterschiedliche Anforderungen geben. Gerade für reaktive Verhaltensweisen ist es meist nicht notwendig, die Positionen aller Objekte exakt zu kennen oder eine vollständige 3D-Rekonstruktion der gesamten Szene zu besitzen. Die Planung von

komplexen Spielzügen dagegen kann i.d.R. umso besser erfolgen, je genauer die Informationen sind, die zur Verfügung stehen.

Eine adäquate Aufbereitung aller verfügbaren Informationen unter Berücksichtigung der vorgegebenen zeitlichen Restriktionen stellt sicherlich die größte Herausforderung im Bereich der Sensordatenverarbeitung dar. Die durch die laufenden Aktionen gestellten Anforderungen hinsichtlich Genauigkeit, Sicherheit und Vollständigkeit der Daten spielen dabei ebenso eine Rolle, wie die Fusion der auf den einzelnen Robotern mit ihren verschiedenen Sensoren ermittelten Daten. Im Gegensatz zu vielen bisherigen Szenarien mit einzelnen, langsam bewegten Robotern stellt die hohe Dynamik des RoboCup-Szenarios neue Anforderungen an die Modellierung von zeitlichen Zusammenhängen und der Dynamik der Sensoren, der Szene und des Gesamtsystems. Weitere interessante Forschungsansätze ergeben sich z.B. durch den Einsatz aktiver Sensoren oder von 360°-Kameras.

2.4.3 Lernen von speziellen Verhaltensweisen

Jeder Roboter muß eine Reihe unterschiedlicher Verhaltensweisen beherrschen, um in den verschiedenen Situationen angemessen (re-) agieren zu können. Eine Möglichkeit, diese bereitzustellen, besteht darin, elementare Fußball-Aktionen, wie z.B. Zum-Ball-Bewegen, Dribbeln, Schießen, Abwehren, Passen usw., einzeln auszuprogrammieren. Dabei bewegt man sich in einem Grenzbereich zwischen Bewegungsplanung und Regelungstechnik, beispielsweise wenn ein Roboter den Ball auf einer Trajektorie um Hindernisse herumbewegen soll. Die verschiedenen Regelkreise, mit denen die einzelnen Bewegungen gesteuert werden sollen, erfordern das Einstellen zahlreicher Parameter; darüber hinaus bestimmen weitere Parameter, welche Aktionen auszuwählen sind.

Mit dieser Vorgehensweise sind für überwiegend statische Situationen recht gute Ergebnissen zu erzielen, mit zunehmender Spieldynamik und vernünftigeren Aktionen des Gegners stößt man jedoch sehr schnell an Grenzen. Die Komplexität des Problems wird durch die hohe Anzahl der Systemzustände und damit die Zahl der abzufragenden Parameter so groß, daß es unmöglich ist, alle Situationen, die auftreten können, bei der Programmierung zu berücksichtigen und explizit zu testen. Die Unsicherheit der Daten durch die Sensordatenauswertung vergrößert noch einmal den Zustandsraum, so daß ein manuelles Einstellen der Regel-, Steuer- und Planungsparameter unmöglich ist. Weitere Probleme können an der Schwelle zwischen zwei Aktionen auftreten, da sich die Voraussetzungen für Aktionen oder deren äußere Parameter dynamisch verändern.

Somit stellt der RoboCup auch für Forschungen im Bereich des Maschinellen Lernens ein interessantes Einsatzgebiet dar. Mit Hilfe von Lernverfahren lassen sich möglicherweise die Parameter für die Robotersteuerung automatisch bestimmen. Eine Schlüsselfrage ist dabei, welche Parameter den aktuelle Systemzustand optimal beschreiben. Ebenfalls noch ungelöst ist die Frage, welche Lernverfahren letztendlich geeignet bzw. optimal sind, insbesondere in Anbetracht des Übergangs vom individuellen zum kooperativen Lernen und der zahlreichen unterschiedlichen Aspekte beim Lernen im RoboCup. Eine naive Anwendung aktueller Lerntheorien wird in Zukunft dafür sicherlich nicht ausreichen.

Mögliche Wechselwirkungen zwischen Aktionsauswahl und Sensordatenbestimmung stellen ein weiteres, auch aus Sicht der vorliegenden Arbeit interessantes Forschungsgebiet dar. Nicht alle Aktionen, die ein Roboter ausführen kann, verwenden die gleiche Datenbasis, re-

aktive Verhaltensweisen kommen u.U. mit 2D-Bildmerkmalen aus, erwarten diese jedoch mit einer möglichst hohen Datenrate, während in anderen Situationen relativ exakte Raumdaten — ggf. auch mit einer geringeren Datenrate — benötigt werden. Da die zur Verfügung stehende Hardware noch auf lange Zeit nicht in der Lage sein wird, stets alle Daten, die für irgendeine Aktion nützlich sein könnten, mit beliebiger Genauigkeit und zeitlichen Auflösung bereitzustellen, werden einerseits Mechanismen benötigt, die entscheiden, welche Daten aktuell gebraucht werden, andererseits muß sich die Sensordatenauswertung wechselnden Anforderungen flexibel anpassen können, wofür das vorgestellte System grundlegende Mechanismen bereitstellt.

2.4.4 Echtzeitplanung und Strategieaneignung

Die im vorhergehenden Abschnitt beschriebenen Aktionen werden i.d.R. von einer übergeordneten Planungskomponente auf der Grundlage von Strategievorgaben und Rollenzuteilungen aufgerufen. In diesem Kontext ergeben sich zwei Aufgabenfelder: die Echtzeitplanung und die Strategieakquisition. Zum einen stellt sich die Aufgabe, kontinuierlich und in Echtzeit auf jedem Roboter geeignete Aktionen auszuwählen und aufzurufen. Maßgeblich dafür ist

- der aktuelle Zustand auf dem Spielfeld,
- die Strategie und
- die ausgewählte oder zugewiesene Rolle des jeweiligen Roboters.

Denkbar ist darüber hinaus auch, geplante Aktionen von Mitspielern zu berücksichtigen oder eine zentrale Instanz vorzusehen, die komplette Spielzüge vorgibt. Verschiedene Handlungsoptionen müssen bewertet, Teilziele festgelegt und dann eine Aktion — ggf. entsprechend parametrisiert — ausgewählt werden. Maschinelles Lernen scheint auch in diesem Kontext ein geeignetes, zumindest jedoch ein untersuchenswertes Werkzeug zu sein. Da die Aktionsauswahl meist zeitkritisch ist, sind spezielle Verfahren wie partielle Planung auf ihre Eignung hin zu untersuchen. Die Planung in einer dynamischen Umgebung erfordert ein hohes Maß an Flexibilität, wobei die Schwierigkeit in der Balance zwischen der Stabilität der Handlungsfolgen und der Anpassung an veränderte Bedingungen besteht.

Eine wichtige Grundlage für die Planung ist die Strategie — das Gesamtkonzept, mit dem vorgegangen werden soll, um zu gewinnen. Ziel ist es dabei, das eigene Verhalten an der Spielweise des Gegners auszurichten. Dazu gehört, Verhaltensmuster der Gegner zu beobachten, zu klassifizieren und diese wiederzuerkennen, bekannte eigene Strategien diesen Verhaltensweisen zuzuordnen, deren Wirksamkeit zu bewerten und ggf. neue Strategien zu erlernen. Entscheidungen bezüglich der Strategie können auch von anderen Kriterien abhängig sein, beispielsweise vom Spielstand oder von Schiedsrichterentscheidungen. Trainingsszenarien unterschiedlicher Komplexität und Spielprotokolle können dabei ebenso zum Einsatz kommen wie evolutionäre Verfahren, die durch Mutationen und Auslese neue Verfahren hervorbringen. Ein Lernen sollte dabei auch während der Spiele erfolgen.

2.4.5 Kooperation in einem Multiagentensystem

Kooperative Systeme, in denen mehrere autonome Agenten gemeinsam an bestimmten Vorgängen beteiligt sind, bringen eine neue Qualität bei der Planung mit sich. Dabei macht es keinen

Unterschied, ob man unter Agenten nur die verschiedenen Roboter als Hardwareeinheit versteht oder auch einzelne intelligente Komponenten auf den Robotern. Gemeinsame Aktionen müssen geeignet synchronisiert werden, außerdem werden Mechanismen benötigt, die, wenn mehrere gleichartige Agenten zur Verfügung stehen, für jede Aktion festlegen, welche Agenten an ihr beteiligt sein und welche Rolle sie übernehmen sollen.

Bei der Ausführung gemeinsamer Handlungen können jederzeit Komplikationen auftreten und einzelne Agenten bei ihrer Aufgabe behindern. Das wird durch die Dynamik der Szene und den Gegner, dessen Bestrebungen den eigenen Zielen entgegenlaufen, verstärkt. In solchen Fällen ist zu entscheiden, ob und wie eine Aktion weitergeführt werden kann. Dabei muß insbesondere zwischen häufigem Umplanen und einer gewissen Stabilität bei der Ausführung abgewogen werden, wofür das Erkennen der Ursachen von Komplikationen eine entscheidende Rolle spielt.

So ist es möglich, daß nur eine kleinere Störung oder Behinderung vorliegt (roboterintern oder aufgrund einer veränderten Spielsituation), die durch den Agenten selbst beseitigt werden kann, während die anderen Agenten kurz warten oder an ihren Teilaufgaben weiterarbeiten. Ein Roboter kann aber auch so massiv behindert werden, daß er sein Teilziel nicht in vertretbarer Zeit erreichen kann. In diesem Fall wird es besser sein, wenn ein anderer Roboter einspringt und die Aufgabe übernimmt. Ist dies nicht möglich oder sinnvoll, muß u.U. vollkommen neu geplant werden.

Schließlich können Roboter oder deren Komponenten auch komplett ausfallen, was erkannt und für die weiteren Planungen berücksichtigt werden muß. Ein kompletter Strategiewechsel stellt eine mögliche Option dar, um auf einen solchen Ausfall zu reagieren.

2.4.6 Systementwurf und Projektmanagement

Die für den RoboCup zu entwicklenden Programme stellen aufgrund der Komplexität, Dynamik und Verteiltheit der Anwendung besondere Anforderungen an den Softwareentwurf. Das breite Spektrum der zu beherrschenden Themen wird i.d.R. dazu führen, daß verschiedene Forschungsgruppen fachübergreifend an einem solchen Projekt zusammenarbeiten. Das erfordert eine saubere Definition der Schnittstellen zwischen den verschiedenen Komponenten und eine klare Trennung von Kompetenzbereichen. Es werden adäquate Konzepte für die Modellierung und Realisierung der dem RoboCup innewohnenden Dynamik, Parallelität und Verteiltheit benötigt, eine weitere Herausforderung, der sich diese Arbeit stellt. Verschiedene Programmkomponenten, die durch unterschiedliche Zykluszeiten und eine nebenläufige Ausführung gekennzeichnet sind, konkurrieren um begrenzte Ressourcen, insbesondere um die Rechenzeit, was wiederum eine klare Modellierung von zeitlichen Zusammenhängen erfordert.

Aufgrund der Größe und der Komplexität des Projekts ist ein modularer und frei skalierbarer Systemaufbau essentiell für die Handhabbarkeit und Erweiterbarkeit des Systems. Gleiches gilt für Mechanismen, die eine systematische Fehlersuche ermöglichen. Herkömmliche Debug-Methoden scheitern i.d.R. am hohen Parallelisierungsgrad und der Dynamik der Anwendung. Weitere Problemfelder ergeben sich aus den großen Datenmengen, die kontinuierlich, schritthaltend und rechtzeitig verarbeitet werden müssen, aus der Unsicherheit und der Unvollständigkeit der Eingangsdaten sowie aus der starken Kopplung zwischen den einzelnen Komponenten. So bilden beispielsweise auf der einen Seite die von der Bildverarbeitung gelieferten Daten die Entscheidungsgrundlage für die verschiedenen Planungsebenen und für eine reaktive Roboter-

steuerung. Aufgrund der begrenzten Rechenzeit ist es aber auf der anderen Seite wünschenswert, daß diese Komponenten die Bildverarbeitung so beeinflussen können, daß einzelne von der aktuellen Aktion benötigte Daten bevorzugt geliefert werden.

Ob sich Aktionen ausführen lassen oder ob sie scheitern, zeigt sich oft erst bei dem Versuch der Ausführung. Statusinformationen auf der untersten Ebene der Aktionsausführung stellen in diesen Fällen die Grundlage für die Weiterplanung dar. Dafür sind Kommunikationskanäle zwischen den verschiedenen Modulen notwendig, Softwareagenten können dabei die verschiedenen Komponenten steuern und als „Ansprechpartner“ für andere Module agieren.

2.4.7 Visualisierung

Die Möglichkeit der Visualisierung der verschiedenen Systemzustände sowie von extrahierten Daten und ausgewählten Aktionen stellt eine Grundvoraussetzung für den Entwurf eines RoboCup-Systems, wie auch anderer komplexer Systeme dar. In der Simulatorliga werden durch entsprechende Hilfsmittel die Aktionen der Agenten überhaupt erst sichtbar. Eine realistische Echtzeit-Animation ist dabei gleichermaßen forschungsrelevant wie eine optimale Informationsaufbereitung. Erstere erhöht für die Zuschauer den Reiz simulierter Fußballspiele, letztere kann dem Entwickler bei der Diagnose der eingesetzten Verfahren helfen.

Auch bei realen Robotern sind Visualisierungskomponenten unverzichtbar. Aufgrund der großen Datenmengen und der komplexen Zusammenhänge kann aus den in der Szene zu beobachtenden Roboterbewegungen allein nicht auf deren inneren Zustand und die Planungsvorgänge geschlossen werden. Insbesondere in der Entwurfs- und Testphase sowie bei der Fehlersuche ist es essentiell, verschiedene Möglichkeiten zur Visualisierung der Sensordaten, der aus ihnen extrahierten Informationen, von Planungsabläufen, der Robotersteuerung sowie von Kommunikations- und Synchronisationsvorgängen zu besitzen. Graphischen Systemen kommt dabei eine besondere Bedeutung zu, da sie hinsichtlich Kompaktheit und leichter Erfaßbarkeit textuellen Ausgaben überlegen sind.

In Anbetracht der großen Datenmengen spielt schließlich auch die Protokollierung zeitlicher Abläufe auf den Robotern und deren anschließende verlangsamte Auswertung eine wichtige Rolle. Dabei sollte die Protokollierung möglichst „verlustfrei“, d.h. ohne einen signifikanten Verbrauch von Rechenzeit erfolgen.

Diese Forderungen wirkten sich maßgeblich auf das Design der in dieser Arbeit entwickelten Objektmodelle aus. Ausdruck dessen ist die Integration von grundlegenden Mechanismen für die Datenprotokollierung, die Ausgabe von Debug- und Statusinformationen und die Visualisierung ikonischer und textueller Sensordaten (lokal oder z.B. mit Hilfe eines Monitorprogramms auf einem anderen Rechner). Darüber hinaus werden Klassen und Verfahren bereitgestellt, die es dem Entwickler bzw. Anwender erlauben, mit dem laufenden System zu kommunizieren, z.B. um bestimmten Objekten gezielt Nachrichten zu schicken oder um die Ausgabewerkzeuge im laufenden System den aktuellen Aufgaben entsprechend umkonfigurieren zu können.

Kapitel 3

Modellierung von Zeit

Ausgehend von der Bedeutung der physikalischen Größe Zeit für dynamische Systeme und Echtzeitanwendungen wird in diesem Kapitel ein konsistentes System für die Notation und Repräsentation von Zeitangaben und zeitlichen Ausdrücken eingeführt. Hierin wird zwischen absoluten und relativen Zeiten sowie zwischen Zeitpunkten und Intervallen unterschieden. Dabei werden elementare Operationen über den verschiedenen Zeitwerten definiert, wobei der Modellierung der Realzeit besondere Bedeutung beigemessen wird.

3.1 Temporale Aspekte in Echtzeitsystemen

In der Literatur spielen temporale Aspekte vor allem im Zusammenhang mit Planungsverfahren eine Rolle, etwa zur Festlegung von Aktionsabfolgen oder bei der Ressourcenvergabe [Dor89, Win94]. Kaum beachtet wurden dagegen bisher Aspekte, wie die explizite Repräsentation von Datenmeßzeiten und die Modellierung des Zeitverhaltens von Sensoren, sowie die Berücksichtigung dieser Größen innerhalb der Sensordatenverarbeitung. Notwendig wird dies z.B., um die Daten verschiedener Sensoren, die zu unterschiedlichen Zeiten aufgenommen wurden, exakt aufeinander abstimmen zu können oder um die Meßzeit in die Lokalisation und Bewegungsprädiktion von Objekten der Szene einfließen zu lassen. Im folgenden werden verschiedene Zeitaspekte, wie sie in dynamischen Anwendungen auftreten, aufgelistet:

Abtastzeitpunkt diskreter Signale: Die im Rechner zu modellierenden Sensordaten können als Meßgrößen (diskreter) Signale aufgefaßt werden. Durch die explizite Repräsentation der Meßzeit lassen sich die Sensordaten wie auch die aus ihnen bestimmten Merkmale einander und den realen Vorgängen der Szene zuordnen, unabhängig vom Zeitpunkt ihrer Bereitstellung. Das Alter spielt darüber hinaus bei der Bewertung der Glaubwürdigkeit der Daten eine wichtige Rolle.

Interpolation und Prädiktion: Für die Interpolation oder Prädiktion von Sensorwerten werden mehrere Datenwerte mit ihren jeweiligen Abtastzeiten benötigt. Eine Interpolation ist z.B. notwendig, um die Daten unterschiedlicher, nicht synchronisierter Sensoren zusammenzuführen. Die Prädiktion von Bildmerkmalen ermöglicht es u.a., gezielt Suchbereiche für den nächsten Zyklus festzulegen.

Zeitabhängige Datenaktualisierung: Das Datenalter stellt ein wichtiges Kriterium dar, um zu entscheiden, ob bei einem Sensordatenzugriff der alte Wert noch gültig ist oder ob ein neuer Wert gemessen bzw. berechnet werden muß.

Zeitcharakteristik von Sensoren: Während Sensoren ihre Daten bereitstellen, vergeht in aller Regel Zeit. Des weiteren besitzen viele Sensoren verschiedene Arbeitsmodi, durch die gesteuert wird, ob die Daten kontinuierlich oder nur bei Bedarf bereitgestellt werden. Daraus ergeben sich für unterschiedliche Sensoren in den einzelnen Modi spezifische Zeitcharakteristiken. Diese können als Grundlage für die Planung der Arbeitsweise der Sensoren und der Datenzugriffe herangezogen werden.

Zeitbedarf von Operatoren: Bildverarbeitungsoperatoren und Datenanalysemethoden benötigen Rechenzeit. Durch die Auswertung von Operatorenlaufzeiten und des Alters der bereitgestellten Daten kann überprüft werden, ob zeitliche Restriktionen eingehalten werden. In Abhängigkeit davon können geeignete Verfahren und Operatoren ausgewählt oder die Auflösung bzw. Genauigkeit der Daten eingeschränkt werden.

Dynamik der Szene: Zeitangaben fließen direkt in die Berechnung von Geschwindigkeiten bewegter Objekte der Szene ein. Diese sind für die Bewegungs- oder Aktionsplanung von Robotern von Bedeutung.

Echtzeitplanung Bei der Bewegungs- und Aktionsplanung sind unter Umständen zeitliche Restriktionen oder Ressourcenengpässe zu beachten. Diese sind in geeigneter Weise zu repräsentieren und mit den realen zeitlichen Abläufen in Zusammenhang zu bringen.

In den folgenden Abschnitten werden — motiviert durch die vielfältigen Anwendungsfelder im Rahmen der dynamischen Sensordatenauswertung — Ausdrucksmittel für die Repräsentation von Zeitgrößen untersucht sowie elementare Operationen über ihnen definiert.

3.2 Repräsentation von Zeit

3.2.1 Überblick

Für die Repräsentation zeitlicher Ausdrücke und Restriktionen kann zwischen quantitativen und qualitativen Ausdrucksformen unterschieden werden. Diese basieren auf den Grundelementen Zeitpunkt und Intervall und können durch einen Menge von Grundrelationen zueinander in Beziehung gesetzt werden. In [Win94] werden verschiedene Repräsentationsformen gegenübergestellt und drei von ihnen (Intervallalgebra [All83, All84], temporales Constraint-Satisfaction-Problem [DMP91] und Mengen von Set-of-Possibility-Occurences [Rit86]) zu einem neuen Modell, dem Zeitkern, vereinigt. Dies hat zum Ziel, Beschränkungen einzelner Methoden durch die anderen Repräsentationsformen auszugleichen. Für Aufgaben aus den Bereichen temporales Schließen und Planen steht damit ein mächtiges Modell zur Verfügung, das im Gegensatz zu den anderen Modellen in der Lage ist, qualitative und quantitative Ausdrücke sowohl mit Zeitpunkten als auch mit Intervallen darzustellen und vorgegebene Restriktionen geeignet zu propagieren.

Im Bereich der Bildfolgenanalyse sind die Anforderungen an die Repräsentation von Zeit allerdings etwas anderer Natur. Arithmetische Operationen auf Zeitpunkten und Intervallen haben hier eine größere Bedeutung als das Propagieren temporaler Constraints. Aus diesem Grund soll hier eine Zeitrepräsentation entwickelt und verwendet werden, die auf Zeitpunkten und Intervallen basiert und über diesen Grundelementen arithmetische Operationen definiert. Wird für Planungsaufgaben oder die Propagierung zeitlicher Restriktionen die erweiterte Funktionalität des Zeitkern benötigt, ist eine spätere Umwandlung der Repräsentationsform möglich.

Grundelemente für die Repräsentation von Zeit

Die Grundelemente für die Repräsentation von Zeit sind **Zeitpunkt** und **Intervall**. Dabei wird ein Intervall durch zwei Zeitpunkte, die Anfangs- und die Endzeit begrenzt. Alternativ kann ein Intervall auch durch einen Zeitpunkt und die Dauer repräsentiert werden.

Zeitangaben können absolut als Realzeit oder relativ erfolgen, aufgrund der unterschiedlichen Semantik soll dieser Unterschied hier explizit modelliert werden. Zum einen sind nicht alle arithmetischen Operationen auf absoluten Zeiten sinnvoll und zum anderen ist die Bedeutung und damit auch der Typ der Ergebnisse arithmetischer Operationen durch die Semantik der Eingangsgrößen vorgegeben, was die Interpretation der Ergebnisse erleichtert.

In dieser Arbeit sollen einfache Zeitpunkte als Basiselemente für die Zeitmodellierung genügen. Dieser Ansatz läßt sich durch alternative Elemente erweitern. Werden die Zeitpunkte beispielsweise durch Zeitschranken (vgl. [Dor89]) ersetzt, ist es möglich, Bereiche für das Eintreffen oder die Dauer von Ereignissen zu definieren. Dadurch können zeitliche Aussagen auch zu nicht vollständig bestimmten Intervallen angeben werden, was vor allem für Planungsaufgaben interessant ist. Dieser Ansatz ließe sich noch weiter verallgemeinern, indem mit beliebigen Funktionen die Genauigkeit jeder einzelnen Zeitangabe modelliert wird.

Granularität und Zeiteinheiten

Im Gegensatz zu anderen Arbeiten (z.B. [Dor89]) soll hier nicht von einem abstrakten Granularitätsinterval als kleinste darzustellende Zeiteinheit ausgegangen werden. Zeit wird als physikalische Größe mit der Basiseinheit Sekunde (oder einer davon abgeleiteten Einheit) modelliert, d.h. die Einheit ist fester Bestandteil jeder Zeitangabe. Das hat den Vorteil, daß sich der äußere Prozeß und die physikalischen Rahmenbedingungen einer Echtzeitanwendung (Zeitcharakteristiken der Sensoren, Bewegungsdaten von Objekten, usw.) direkt in dem vorgestellten Zeitsystem ausdrücken lassen. Weiterhin haben damit alle Zeitangaben sofort einen realen Bezug.

Der Vorteil des Granularitätsintervalls, nämlich die einfache Skalierbarkeit einer Lösung auf neue Problembereiche, die lediglich ein anderes Grundintervall besitzen, kommt hier nicht zum tragen, da in einer Echtzeitanwendung sehr viele unterschiedliche Problemfelder mit unabhängigem Zeitverhalten zusammentreffen, die sich nicht gemeinsam skalieren lassen.

3.2.2 Relative Zeitpunkte

Mit relativen Zeitpunkten kann das Eintreffen von Ereignissen in bezug auf ein anderes Ereignis beschrieben werden. Auch die Dauer einer Aktion läßt sich damit ausdrücken, sie ergibt sich aus dem Endzeitpunkt der Aktion in Relation zu deren Beginn. Dynamische Objekt- und Sy-

stemeigenschaften lassen sich mit ihrer Hilfe ebenfalls ausdrücken. Relativzeiten, die auf anderen Systemen ermittelt wurden, können auch bei nicht synchronisierten Uhren direkt verwendet werden. Darüber hinaus können sie im Gegensatz zu absoluten Zeiten gemittelt, vervielfacht oder aufaddiert werden.

Die Menge der Relativzeitpunkte $\mathcal{T}_r$ wird wie folgt beschrieben, wobei τ als Symbol für einen relativen Zeitpunkt steht:

$$\mathcal{T}_r = \{\tau : \tau = v\,[\mathbf{E}]; v \in \mathcal{R};\ [\mathbf{E}] \in \{\mathsf{s}, \mathsf{ms}, \mu\mathsf{s}, \ldots\}\}$$

$\mathcal{R}$ bezeichnet die Menge der reellen Zahlen, v den Zeitwert und $[\mathbf{E}]$ steht für die Einheit, die ein wesentlicher Bestandteil der Zeitangabe ist. Basiseinheit ist die Sekunde (s), prinzipiell können aber auch alle davon abgeleiteten Einheiten verwendet werden.

3.2.3 Absolute Zeitpunkte

Absolute Zeiten sind dadurch gekennzeichnet, daß sie fest in der realen Zeit verankert sind, d.h. sie entsprechen einer realen Datum-/Uhrzeit-Kombination. Um das Rechnen mit Zeitangaben zu vereinfachen, erfolgt die Modellierung der Zeit jedoch nicht mit Tag, Monat, Jahr usw., sondern in Form eines einzigen Zeitwerts, der mit einer Einheit (üblicherweise Sekunden) versehen auf einer Zeitskala abgebildet wird. Diese Zeitskala hat ihren Nullpunkt in einem konkreten Zeitpunkt, wodurch der Bezug zwischen dem Zeitwert und der Realzeit hergestellt wird. Die Menge der Absolutzeitpunkte $\mathcal{T}_a$ kann wie folgt beschrieben werden, wobei t als Symbol für absolute Zeitpunkte verwendet wird:

$$\begin{aligned}
\mathcal{T}_a &= \{t : t = v\,[\mathbf{E}]\big|_{\mathrm{T}}; v \in \mathcal{R};\ [\mathbf{E}] \in \{\mathsf{s}, \mathsf{ms}, \mu\mathsf{s}, \ldots\}; \mathrm{T} \in \mathcal{D}\} \\
\mathcal{D} &= \{\mathrm{T} : \mathrm{T} = (Y, M, D, h, m, s);\ \mathrm{Jahr}: \quad Y \in \mathcal{G}, \\
&\qquad \mathrm{Monat}: \quad M \in \mathcal{N},\ 1 \le M \le 12, \\
&\qquad \mathrm{Tag}: \quad D \in \mathcal{N},\ 1 \le D \le 31, \\
&\qquad (Y, M, D)\ \mathrm{ist\ g\ddot{u}ltiges\ Datum}, \\
&\qquad \mathrm{Stunde}: \quad h \in \mathcal{N},\ 0 \le h < 24, \\
&\qquad \mathrm{Minute}: \quad m \in \mathcal{N},\ 0 \le m < 60, \\
&\qquad \mathrm{Sekunde}: \quad s \in \mathcal{R},\ 0 \le s < 60\}.
\end{aligned}$$

$\mathcal{N}$ bezeichnet die Menge der natürlichen Zahlen, $\mathcal{G}$ die der ganzen Zahlen und $\mathcal{R}$ die reellen Zahlen. $\mathcal{D}$ steht für die Menge gültiger Datum-/Uhrzeit-Angaben. Diese setzen sich aus Jahr, Monat, Tag und Uhrzeit zusammen und stehen für einen realen, im Kalender möglichen Zeitpunkt.

Jeder Zeitwert t entspricht damit einer absoluten Datum-/Uhrzeit-Angabe. T legt fest, wie die Zeitskala in der realen Zeit verankert ist, bestimmt also den Nullpunkt, auf den sich t bezieht. In unterschiedlichen Systemen können verschiedene Zeitskalen verwendet werden. Insbesondere können sich auch die Zeitanker von Teilsystemen eines Gesamtsystems unterscheiden. Ist dieser Bezug eindeutig oder irrelevant, kann die Angabe von T auch wegfallen. Sollen absolute Zeitwerte aus unterschiedlichen Systemen miteinander verknüpft werden, müssen die Zeitwerte der beiden Systeme synchronisiert werden. Das kann dadurch geschehen, daß die Relativzeit zwischen den beiden Zeitankern $\tau_{\mathrm{T}_{1,2}}$ ermittelt und auf alle Werte der anzupassenden Skala aufaddiert wird:

$$t_i\big|_{\mathrm{T}_1} = t_i\big|_{\mathrm{T}_2} + \tau_{\mathrm{T}_{1,2}}, \quad \tau_{\mathrm{T}_{1,2}} = \mathrm{T}_2 - \mathrm{T}_1$$

Die Vorschrift zum Ermitteln der Differenzzeit zwischen zwei konkreten Datumsangaben wird als bekannt vorausgesetzt und soll hier nicht explizit ausgeführt werden. Weiterhin muß gewährleistet sein, daß alle Zeitwerte in der gleichen Einheit angegeben werden. In [Jos99] wurde ein entsprechendes Verfahren für den Zeitabgleich autonomer Robotersysteme implementiert. Für die weiteren Betrachtungen wird davon ausgegangen, daß die Zeiten, die über eine Operation $(\mathcal{O}_\mathcal{P})$ miteinander in Beziehung gesetzt werden, den gleichen Zeitanker besitzen und in Sekunden angegeben werden:

$$t_1(\mathcal{O}_\mathcal{P})\, t_2 \quad \Longrightarrow \quad t_1 = v_1\,[\mathbf{E_1}]\Big|_{\mathrm{T}_1} \wedge t_2 = v_2\,[\mathbf{E_2}]\Big|_{\mathrm{T}_2} \wedge \mathrm{T}_1 = \mathrm{T}_2 \wedge [\mathbf{E_1}] = [\mathbf{E_2}] = \mathsf{s}$$

3.2.4 Intervalle

Intervalle erlauben die Repräsentation von Zeiträumen. Mit ihnen lassen sich Beginn, Dauer und Ende einer Aktion oder eines Zustands mit nur einem Datentyp beschreiben ebenso wie Zeitspannen, in denen bestimmte Ereignisse auftreten können oder müssen. Gegenüber einfachen Zeitpunkten erlauben zusätzliche Intervalloperationen Aussagen über die zeitliche Relation verschiedener Aktionen, beispielsweise hinsichtlich deren Aufeinanderfolge oder zeitlichen Überlappung. Intervalle können als Menge aller Zeitpunkte zwischen Start- und Endzeitpunkt einschließlich dieser beiden Zeiten interpretiert werden.

Intervalle können wie Zeitpunkte im absoluten oder einem relativen Zeitsystem angegeben werden. Absolute Intervalle beschreiben Beginn, Dauer und Ende real ablaufender, geplanter oder prognostizierte Vorgänge. Mit ihnen lassen sich weiterhin Toleranzbereiche für das Eintreffen von Ereignissen formulieren. Relative Intervalle erlauben dagegen z.B., das charakteristische Zeitverhalten eines Systems zu beschreiben. Dargestellt und gebildet werden Intervalle wie folgt:

Absolute Intervalle

$$\mathcal{I}_a = \{i \,:\, i = [i^-, i^+]\Big|_{\mathrm{T}};\; i^-, i^+ \in \mathcal{T}_a;\; i^- \leq i^+;\; \mathrm{T} \in \mathcal{D}\}$$

Dabei bezeichnet i^- den Startzeitpunkt und i^+ den Endzeitpunkt des Intervalls. Sie werden als absolute Zeitpunkte mit der gleichen Zeitbasis T angegeben. Als Symbol für Absolutintervalle wird i verwendet.

Relative Intervalle

$$\mathcal{I}_r = \{\iota \,:\, \iota = [\iota^-, \iota^+];\; \iota^-, \iota^+ \in \mathcal{T}_r;\; \iota^- \leq \iota^+\}$$

Auch hier bezeichnet ι^- den Startzeitpunkt und ι^+ den Endzeitpunkt des Intervalls. Allerdings stellen die Zeiten hier relative Zeitpunkte dar. Die Kennzeichnung von Relativintervallen erfolgt durch das Symbol ι.

3.2.5 Spezielle Intervalle

Im Bereich des temporalen Schließens spielen neben den vollständig quantitativ bestimmten Intervallen auch qualititative Intervallangaben eine wichtige Rolle. Always bezeichnet den Zeitraum, der alle möglichen Zeitpunkte vereinigt, Never ($i = \emptyset$) ist dagegen der Zeitraum, zu

dem kein einziger Zeitpunkt gehört. Never kann auch für die Bezeichnung inkonsistenter Intervalldefinitionen ($i^- > i^+$) herangezogen werden. Intervallmengen können mit Hilfe von Intervallrelationen in Abhängigkeit zu anderen Intervallen definiert werden. Diese lassen sich wieder durch ein Intervall oder aber eine Liste von Intervallen beschreiben. Schließlich können Intervalle offen sein, d.h. bei ihnen ist nur eine Grenze — Beginn: $\mathsf{From}(t)$ oder Ende: $\mathsf{To}(t)$ — festgelegt.

3.2.6 Bestimmung von Absolutzeiten

Die Realzeit t_{REAL} beschreibt die kontinuierlich voranschreitende Zeit, wobei alle Zeitpunkte eine Art Lebenszyklus bestehend aus Zukunft, Gegenwart und Vergangenheit durchlaufen. Die Gegenwart ist der gerade aktuelle Zeitpunkt und wird mit t_{NOW} bezeichnet. Vergangene Aktionen werden durch Zeiten $t < t_{\mathsf{NOW}}$ charakterisiert. Diese sind fest und können nicht mehr verschoben werden. Im Gegensatz dazu sind Ereignisse in der Zukunft ($t > t_{\mathsf{NOW}}$) noch offen.

Die aktuelle Realzeit kann mit Hilfe einer Uhr mit einer bestimmten Granularität τ_{gran} sowie einer gewissen Genauigkeit τ_ε gemessen werden, der aktuell gemessene Wert soll hier mit t_{now} bezeichnet werden. Die Granularität bezeichnet die Zeitdauer, die maximal vergehen kann, bevor eine erneute Messung einen anderen Zeitwert liefert. Sie kann in den hier betrachteten Systemen als vernachlässigbar klein angenommen werden, da die Dauer aller Aktionen $\tau_d(A)$ deutlich über der gegebenen Granularität liegt:

$$\begin{aligned} t_{\mathsf{REAL}}(E_1) = t_{\mathsf{REAL}}(E_2) &\iff E_1 \text{ und } E_2 \text{ treten exakt gleichzeitig auf} \\ t(E_1) = t(E_2) &\implies |t_{\mathsf{REAL}}(E_1) - t_{\mathsf{REAL}}(E_2)| < \tau_{\mathsf{gran}} \end{aligned}$$

$$\forall A: \quad \tau_d(A) \gg \tau_{\mathsf{gran}}$$

$E_1, E_2 \in$ Ereignisse, $A \in$ Aktionen, $\tau_d(A) =$ Dauer der Aktion A

Ungenauigkeiten bei der Zeitmessung entstehen zum einen durch Abweichung der angenommenen Zeitbasis einer Zeitskala von der tatsächlichen Zeitbasis (Skalierungsfehler der Uhr). Diese Abweichung kann durch eine Fehlerkonstante im Zeitanker modelliert werden:

$$t = v\,[\mathbf{E}]\,\Big|_{\mathrm{T}+\tau_{\varepsilon_c}}$$

Dieser Fehler wirkt sich insbesondere dann aus, wenn Zeiten unterschiedlicher Systeme, die mit eigenen Uhren ausgestattet sind, in Beziehung gesetzt werden sollen, selbst wenn diese Systeme an und für sich die gleiche Zeitbasis verwenden, wie das z.B. bei allen UNIX-Systemen mit $\mathrm{T} = [1.1.1970, 0.00\text{Uhr}, \text{GMT}]$ der Fall ist. Weiterhin kann die Abweichung der Uhr von der Realzeit einen nicht konstanten Anteil beinhalten, d.h. der Fehler τ_ε verändert sich mit der Zeit. Diese Veränderung soll hier jedoch als so gering angesehen werden, daß sie vernachlässigt werden kann. Es interessiert i.d.R. auch nicht die absolute Realzeit einer Aktion, sondern nur der Zeitversatz zu anderen Systemen. Dieser kann aber bei Bedarf jederzeit neu ermittelt werden.

Des weiteren kann die Zeitmessung durch den Meßprozeß selbst verfälscht werden. Zeitmessungen von äußeren Ereignissen beruhen i.d.R. darauf, daß *gleichzeitig* mit dem Ereignis E der aktuelle Zeitpunkt t_{now} ermittelt wird und die gemessene Zeit mit der Ereigniszeit gleichgesetzt wird: $t(E) = t_{\mathsf{now}}$. Dabei kann in vielen Fällen die Annahme der Gleichzeitigkeit

von Messung und Ereignis, etwa durch Verzögerungen des Meßaufbaus, nicht aufrechterhalten werden. Ähnliches gilt auch für die vom System selbst initiierten Aktionen, für die zwar der Zeitpunkt des Einleitens der Aktion ermittelt werden kann, die eigentliche Aktion aber möglicherweise erst später beginnt. Da diese Verzögerungen in dynamischen Systemen durchaus relevant sein können, sollte das Zeitverhalten des Meßaufbaus und das des zu messenden Systems möglichst genau modelliert werden können.

Schließlich kann in einem Computersystem — unabhängig von verfahrensbedingten Fehlern — auch die Zeitmessung selbst kleinen Ungenauigkeiten unterworfen sein. Die Ursachen für Abweichungen zwischen dem Abfragen der Uhrzeit und dem Zugriff auf die Uhr findet man in möglichen Compileroptimierungen, Pipelining und der Parallelausführung mehrerer Programmzweige, im asynchronen Zugriff auf die Systemuhr sowie in möglichen Wartezyklen beim Buszugriff. Dieser Fehler ist nur schwer zu modellieren, liegt allerdings auch nur im Bereich weniger Taktzyklen und damit weit unter den relevanten Bearbeitungszeiträumen.

3.3 Operationen über Zeitpunkten und Intervallen

Zeitoperationen können verschiedenen Kategorien oder Klassen zugeordnet werden, die sich hinsichtlich der Semantik der Operationen sowie der Datentypen der Eingangs- und Ausgangszeiten unterscheiden. *Vergleichsoperationen* liefern einen Wahrheitswert $b \in \mathcal{B} = \{\mathsf{true}, \mathsf{false}\}$ zurück. Dafür werden Zeitwerte über ihre Ordnungsrelationen miteinander in Beziehung gesetzt. Mit Hilfe *arithmetischer Operationen* lassen sich Zeitwerte verschieben, in Relation zueinander setzen oder skalieren. *Konstruktionsoperatoren* erlauben es, aus Zeitwerten neue Zeitwerte zu konstruieren und *Zugriffsoperationen* schließlich ermöglichen den Zugriff auf einzelne Merkmale von Zeitwerten, insbesondere von Intervallen.

Für alle Operationen mit absoluten Zeitangaben wird vorausgesetzt, daß die Zeiten, die miteinander in Beziehung gesetzt werden, den gleichen Zeitanker besitzen. Ist dies nicht gegeben, muß zuvor ein Angleich der Skalen erfolgen. Weiterhin wird vorausgestzt, daß alle Zeitwerte in der gleichen Einheit, in Sekunden $[\mathsf{s}]$ angegeben werden.

Schließlich soll auf mögliche Spezialfälle bei verschiedenen Operationen hingewiesen werden. Diese ergeben sich dadurch, daß für Relativzeiten $0[\mathsf{s}]$ und für absolute Zeiten t_{now} angegeben werden kann. Dort wo es sinnvoll erscheint, sollen dafür spezielle Operationen definiert werden.

3.3.1 Vergleichsoperationen

Vergleichsoperationen erlauben es, Aussagen über die Reihenfolge des Eintreffens von Ereignissen, über die Gleichzeitigkeit, Überlagerung oder Aufeinanderfolge sowie die Dauer von Aktionen zu formulieren, und den Wahrheitsgehalt dieser Aussagen zu bestimmen. Es können Zeitpunkte sowohl mit anderen Zeitpunkten als auch mit Intervallen sowie Intervalle untereinander in Relation gesetzt werden. Dabei gibt es jedoch keine vernünftige Interpretation für direkte Vergleiche von absoluten mit relativen Zeiten, die Zeiten müssen also beide der gleichen Klasse angehören.

Vergleich von Zeitpunkten

$$(\mathcal{O}_\mathcal{P}) \in \{<, \leq, =, \geq, >, \neq\} : \quad \begin{array}{l} \mathcal{T}_a \times \mathcal{T}_a \longmapsto \mathcal{B}, \quad t_1, t_2 \in \mathcal{T}_a\,, b \in \mathcal{B} \\ \mathcal{T}_r \times \mathcal{T}_r \longmapsto \mathcal{B}, \quad \tau_1, \tau_2 \in \mathcal{T}_r, b \in \mathcal{B} \end{array}$$

$$\begin{array}{lll} \text{absolut:} & b := t_1\,(\mathcal{O}_\mathcal{P})\,t_2 : & t_i = v_i\,[\mathbf{E}]\Big|_{\mathrm{T}}, i \in \{1,2\} \quad \leftrightarrow \quad b := v_1\,(\mathcal{O}_\mathcal{P})\,v_2 \\ \text{relativ:} & b := \tau_1\,(\mathcal{O}_\mathcal{P})\,\tau_2 : & \tau_i = v_i\,[\mathbf{E}], \quad i \in \{1,2\} \quad \leftrightarrow \quad b := v_1\,(\mathcal{O}_\mathcal{P})\,v_2 \end{array}$$

Der Vergleich zweier absoluter Zeitpunkte liefert eine Aussage darüber, in welcher Reihenfolge die Zeitpunkte aufgetreten sind bzw. auftreten werden. Die Interpretation der Operatorsymbole läßt sich direkt auf die Zeitwerte v übertragen, erfolgt also in der gewohnten Weise, z.B. bedeutet $t_1 < t_2 = \mathsf{true}$, daß $v_1 < v_2$, also daß t_1 vor t_2 auftritt.

Mögliche Spezialfälle ergeben sich bei $t_2 = t_{\mathsf{now}}$ und $\tau_2 = 0$: Die Operatoren für absolute Zeitpunkte testen somit, ob der betrachtete Zeitpunkt in der Zukunft, Gegenwart oder Vergangenheit liegt. Dabei hängt das Ergebnis vom konkreten Anfragezeitpunkt ab.

$$(\mathcal{O}_\mathcal{P}) \in \{\mathsf{is}_+, \mathsf{is}_0, \mathsf{is}_-\} : \quad \begin{array}{l} \mathcal{T}_a \longmapsto \mathcal{B}, \quad t \in \mathcal{T}_a\,, b \in \mathcal{B} \\ \mathcal{T}_r \longmapsto \mathcal{B}, \quad \tau \in \mathcal{T}_r, b \in \mathcal{B} \end{array}$$

absolut:			relativ:		
$b := \mathsf{is}_+(t)$	$\leftrightarrow$	$b := t > t_{\mathsf{now}}$	$b := \mathsf{is}_+(\tau)$	$\leftrightarrow$	$b := \tau > 0$
$b := \mathsf{is}_0(t)$	$\leftrightarrow$	$b := t = t_{\mathsf{now}}$	$b := \mathsf{is}_0(\tau)$	$\leftrightarrow$	$b := \tau = 0$
$b := \mathsf{is}_-(t)$	$\leftrightarrow$	$b := t < t_{\mathsf{now}}$	$b := \mathsf{is}_-(\tau)$	$\leftrightarrow$	$b := \tau < 0$

Relationen zwischen Intervallen

$$(\mathcal{O}_\mathcal{P}) \in \{\mathsf{eq}, \mathsf{m}, \mathsf{mi}, \mathsf{b}, \mathsf{a}, \mathsf{o}, \mathsf{oi}, \mathsf{d}, \mathsf{di}, \mathsf{s}, \mathsf{si}, \mathsf{f}, \mathsf{fi}\} : \quad \begin{array}{l} \mathcal{I}_a \times \mathcal{I}_a \longmapsto \mathcal{B}, \quad i_1, i_2 \in \mathcal{I}_a, b \in \mathcal{B} \\ \mathcal{I}_r \times \mathcal{I}_r \longmapsto \mathcal{B}, \quad \iota_1, \iota_2 \in \mathcal{I}_r, b \in \mathcal{B} \end{array}$$

absolut:

$b := i_1\,(\mathsf{eq})\,i_2$	$\leftrightarrow$	$b := i_2\,(\mathsf{eq})\,i_1$	$\leftrightarrow$	$b := i_1^- = i_2^- \wedge i_1^+ = i_2^+$
$b := i_1\,(\mathsf{m})\,i_2$	$\leftrightarrow$	$b := i_2\,(\mathsf{mi})\,i_1$	$\leftrightarrow$	$b := i_1^+ = i_2^-$
$b := i_1\,(\mathsf{b})\,i_2$	$\leftrightarrow$	$b := i_2\,(\mathsf{a})\,i_1$	$\leftrightarrow$	$b := i_1^+ < i_2^-$
$b := i_1\,(\mathsf{o})\,i_2$	$\leftrightarrow$	$b := i_2\,(\mathsf{oi})\,i_1$	$\leftrightarrow$	$b := i_1^- < i_2^- \wedge i_2^- < i_1^+ \wedge i_1^+ < i_2^+$
$b := i_1\,(\mathsf{d})\,i_2$	$\leftrightarrow$	$b := i_2\,(\mathsf{di})\,i_1$	$\leftrightarrow$	$b := i_1^- > i_2^- \wedge i_1^+ < i_2^+$
$b := i_1\,(\mathsf{s})\,i_2$	$\leftrightarrow$	$b := i_2\,(\mathsf{si})\,i_1$	$\leftrightarrow$	$b := i_1^- = i_2^- \wedge i_1^+ < i_2^+$
$b := i_1\,(\mathsf{f})\,i_2$	$\leftrightarrow$	$b := i_2\,(\mathsf{fi})\,i_1$	$\leftrightarrow$	$b := i_1^- > i_2^- \wedge i_1^+ = i_2^+$

Der Vergleich zweier absoluter Intervalle liefert eine Aussage darüber, in welcher Reihenfolge die Intervalle auftreten und ob sie sich überlappen. Die Operationen $(\mathcal{O}_\mathcal{P})$ entsprechen den 13 Grundrelationen zwischen Intervallen nach [All83]: *equals* (eq), *meets* (m), *is met* (mi), *before* (b), *after* (a), *overlaps* (o), *is overlapped* (oi), *during* (d), *contains* (di), *starts* (s), *is started* (si), *finishes* (f), *is finished* (fi). Sie stützen sich auf die zuvor beschriebenen Vergleichsoperationen zwischen Zeitpunkten ab. Hier wurden nur die Intervallgrundrelationen definiert, abgeleitete Relationen können durch boolsche Operationen über den Ergebnissen b gebildet werden.

Die Vergleichsoperationen für relative Intervalle werden hier nicht explizit angegeben, da sie in der exakt gleichen Weise gebildet werden, wie die für absolute Intervalle. In obiger

Definition sind lediglich alle i durch ι zu ersetzen. Der Vergleich zweier Relativintervalle liefert eine Aussage darüber, wie die Intervalle zueinander liegen und ob sie sich überlappen. Zu den Operationen gilt analog das zuvor für absolute Intervalle Gesagte.

Relationen zwischen Zeitpunkt und Intervall

$$(\mathcal{O}_\mathcal{P}) \in \{<, \mathsf{s}, \in, \mathsf{f}, >\}: \quad \begin{array}{ll} \mathcal{T}_a \times \mathcal{I}_a \longmapsto \mathcal{B}, & t \in \mathcal{T}_a\,, i \in \mathcal{I}_a, b \in \mathcal{B} \\ \mathcal{T}_r \times \mathcal{I}_r \longmapsto \mathcal{B}, & \tau \in \mathcal{T}_r, \iota \in \mathcal{I}_r, b \in \mathcal{B} \end{array}$$

$$(\mathcal{O}_\mathcal{P}) \in \{<, \mathsf{fi}, \ni, \mathsf{si}, >\}: \quad \begin{array}{ll} \mathcal{I}_a \times \mathcal{T}_a \longmapsto \mathcal{B}, & i \in \mathcal{I}_a, t \in \mathcal{T}_a\,, b \in \mathcal{B} \\ \mathcal{I}_r \times \mathcal{T}_r \longmapsto \mathcal{B}, & \iota \in \mathcal{I}_r, \tau \in \mathcal{T}_r, b \in \mathcal{B} \end{array}$$

absolut:

$$\begin{array}{lcllcl} b := t < i & \leftrightarrow & b := i > t & \leftrightarrow & b := t < i^- \\ b := t\ (\mathsf{s})\ i & \leftrightarrow & b := i\ (\mathsf{si})\ t & \leftrightarrow & b := t = i^- \\ b := t \in i & \leftrightarrow & b := i \ni t & \leftrightarrow & b := i^- \leq t \wedge t \leq i^+ \\ b := t\ (\mathsf{f})\ i & \leftrightarrow & b := i\ (\mathsf{fi})\ t & \leftrightarrow & b := t = i^+ \\ b := t > i & \leftrightarrow & b := i < t & \leftrightarrow & b := t > i^+ \end{array}$$

Der Vergleich zwischen einem Zeitpunkt und einem Intervall liefert eine Aussage darüber, ob ein Zeitpunkt vor ($<$), im ($\in$) oder nach ($>$) dem Intervall auftritt, bzw. ob er mit dem Start- oder Endzeitpunkt zusammenfällt (s, f bzw. si, fi). Die für absolute Intervalle angegebenen Relationen gelten auch für Relativintervalle. Bei der obigen Definition sind alle t durch τ und alle i durch ι zu ersetzen.

3.3.2 Arithmetische Operationen

Mit den in diesem Abschnitt vorgestellten arithmetischen Zeitoperationen wird ein Instrumentarium für das Rechnen mit Zeitangaben bereitgestellt. Im Gegensatz zu den Vergleichsoperationen können mit arithmetischen Operationen Relativ- mit Absolutzeiten verknüpft werden. Allerdings sind auch hier nicht alle Operationen sinnvoll. Beispielsweise gibt es keine vernünftige Interpretation für die Addition von Absolutzeiten. An dieser Stelle sollen nur sinnvolle Operationen, d.h. Operationen, für die es eine vernünftige Interpretation gibt, definiert werden.

Verzichtet werden soll hier auf die Definition weiterer physikalischer Größen, die die Zeit beinhalten, wie Geschwindigkeit, Beschleunigung oder Frequenz.

Verhältnis von Relativzeitpunkten

$$(\mathcal{O}_\mathcal{P}) \in \{/\}: \quad \mathcal{T}_r \times \mathcal{T}_r \longmapsto \mathcal{R}, \quad \tau_1, \tau_2 \in \mathcal{T}_r,\ r \in \mathcal{R}$$

$$r := \tau_1 \,/\, \tau_2 :\ \tau_i = v_i\,[\mathbf{E}],\ i \in \{1, 2\} \quad \rightarrow \quad r :=\ v_1 \,/\, v_2$$

Diese Operation dient in erster Linie dem quantitativen Vergleich zweier Zeiträume. Für Absolutzeiten ist eine solche Operation nicht sinnvoll.

Skalierung von Relativzeiten

$$(\mathcal{O}_\mathcal{P}) \in \{*, /\}: \quad \begin{array}{ll} \mathcal{T}_r \times \mathcal{R} \longmapsto \mathcal{T}_r, & \tau, \tau' \in \mathcal{T}_r,\ r \in \mathcal{R} \\ \mathcal{I}_r \times \mathcal{R} \longmapsto \mathcal{I}_r, & \iota, \iota' \in \mathcal{I}_r,\ r \in \mathcal{R} \end{array}$$

$$\begin{array}{lllll} \text{Zeitpunkte:} & \tau' := \tau * r\ (= r * \tau) & : \tau = v\,[\mathbf{E}] & \rightarrow & \tau' := (v\ *\ r)\,[\mathbf{E}] \\ & \tau' := \tau \,/\, r & : \tau = v\,[\mathbf{E}] & \rightarrow & \tau' := (v \,/\, r)\,[\mathbf{E}] \\ \text{Intervalle:} & \iota' := \iota * r\ (= r * \iota) & : \iota = [\iota^-, \iota^+] & \rightarrow & \iota' := [\iota^- * r,\ \iota^+ * r] \\ & \iota' := \iota \,/\, r & : \iota = [\iota^-, \iota^+] & \rightarrow & \iota' := [\iota^- \,/\, r,\ \iota^+ \,/\, r] \end{array}$$

Skalierungs- und Verhältnisoperationen sind nur für Relativzeiten sinnvoll. Zusammen mit der weiter unten beschriebenen Intervalladdition erlaubt die Skalierung von Intervallen deren Mittelung, beispielsweise für eine Berechnung eines durchschnittlichen Zeitverhaltens.

Addition und Subtraktion von Relativzeiten

$$(\mathcal{O}_\mathcal{P}) \in \{+, -\}: \quad \begin{array}{ll} \mathcal{T}_r \times \mathcal{T}_r \longmapsto \mathcal{T}_r, & \tau_1, \tau_2, \tau' \in \mathcal{T}_r \\ \mathcal{I}_r \times \mathcal{T}_r \longmapsto \mathcal{I}_r, & \iota, \iota' \in \mathcal{I}_r,\ \tau \in \mathcal{T}_r \\ \mathcal{I}_r \times \mathcal{I}_r \longmapsto \mathcal{I}_r, & \iota_1, \iota_2, \iota' \in \mathcal{I}_r \end{array}$$

$$\begin{array}{lllll} \mathcal{T}_r \times \mathcal{T}_r: & \tau' := \tau_1 + \tau_2\ (= \tau_2 + \tau_1) & : \tau_{1,2} = v_{1,2}\,[\mathbf{E}] & \rightarrow & \tau' := (v_1 + v_2)\,[\mathbf{E}] \\ & \tau' := \tau_1 - \tau_2 & : \tau_{1,2} = v_{1,2}\,[\mathbf{E}] & \rightarrow & \tau' := (v_1 - v_2)\,[\mathbf{E}] \\ \mathcal{I}_r \times \mathcal{T}_r: & \iota' := \iota + \tau\ (= \tau + \iota) & : \iota = [i^-, i^+] & \rightarrow & \iota' := [i^- + \tau,\ i^+ + \tau] \\ & \iota' := \iota - \tau & : \iota = [i^-, i^+] & \rightarrow & \iota' := [i^- - \tau,\ i^+ - \tau] \\ \mathcal{I}_r \times \mathcal{I}_r: & \iota' := \iota_1 + \iota_2\ (= \iota_2 + \iota_1) & : \iota_{1,2} = [\iota_{1,2}^-, \iota_{1,2}^+] & \rightarrow & \iota' := [\iota_1^- + \iota_2^-,\ \iota_1^+ + \iota_2^+] \end{array}$$

Die Addition oder Subtraktion von Relativzeiten läßt sich zum einen als Verschieben eines Zeitpunkts um eine Relativzeit interpretieren und zum anderen als Berechnung einer Gesamtdauer aus den Zeiträumen einzelner Teilprozesse. Die Addition von zwei Intervallen ist notwendig, um ein mittleres Zeitverhalten, beispielsweise aus mehreren Messungen zu berechnen. Für die Subtraktion von Intervallen gibt es keine offensichtliche Interpretation. Da auch nicht garantiert ist, daß dabei wieder ein gültiges Intervall ι mit $\iota^- \leq \iota^+$ herauskommt, soll diese Operation hier nicht definiert werden.

Differenzen von Absolutzeiten

$$(\mathcal{O}_\mathcal{P}) \in \{-\}: \quad \begin{array}{ll} \mathcal{T}_a \times \mathcal{T}_a \longmapsto \mathcal{T}_r, & t_1, t_2 \in \mathcal{T}_a,\ \tau \in \mathcal{T}_r \\ \mathcal{I}_a \times \mathcal{T}_a \longmapsto \mathcal{I}_r, & i \in \mathcal{I}_a,\ t \in \mathcal{T}_a,\ \iota \in \mathcal{I}_r \end{array}$$

$$\begin{array}{lllll} \mathcal{T}_a \times \mathcal{T}_a: & \tau := t_1 - t_2 : & t_i = v_i\,[\mathbf{E}]\Big|_\mathrm{T}, i \in \{1, 2\} & \rightarrow & \tau := (v_1 - v_2)\,[\mathbf{E}] \\ \mathcal{I}_a \times \mathcal{T}_a: & \iota := i - t & : i = [i^-, i^+]\Big|_\mathrm{T} & \rightarrow & \tau := [i^- - t, i^+ - t]\Big|_\mathrm{T} \end{array}$$

Die Differenz zweier Absolutzeiten liefert eine Relativzeit, die als Dauer eines Prozesses oder als Versatz eines Ereignisses relativ zu einem anderen interpretiert werden kann. Ein Intervall wird durch Abziehen einer Absolutzeit zu dieser in Relation gesetzt. Für die Addition von Absolutzeiten sowie für Differenzen zwischen Intervallen gibt es keine vernünftige Interpretation.

Spezialfall ($t_1 = t_{\mathsf{now}}$): Wird ein Absolutzeitpunkt auf die aktuelle Zeit t_{now} gesetzt, erhält man als Spezialfall der zuvor definierten Operatoren Funktionen für die Berechnung des Alters von Zeitangaben. Das Alter τ_{age} eines Intervalls wird hier über dessen Beginn definiert, τ_{ago} beschreibt dagegen, wie lang das Intervallende zurück liegt. Für Zeitpunkte fallen diese beide Zeiträume zusammen:

$$(\mathcal{O}_\mathcal{P}) \in \{\tau_{\mathsf{age}}, \tau_{\mathsf{ago}}\}: \quad \begin{array}{ll} \mathcal{T}_a \longmapsto \mathcal{T}_r, & t \in \mathcal{T}_a,\ \tau \in \mathcal{T}_r \\ \mathcal{I}_a \longmapsto \mathcal{T}_r, & i \in \mathcal{I}_a,\ \tau \in \mathcal{T}_r \end{array}$$

$$\begin{array}{lll} \tau := \tau_{\mathsf{age}}(t), \quad \tau := \tau_{\mathsf{ago}}(t) & \rightarrow & \tau := t_{\mathsf{now}} - t \\ \tau := \tau_{\mathsf{age}}(i) & \rightarrow & \tau := t_{\mathsf{now}} - i^- \\ \tau := \tau_{\mathsf{ago}}(i) & \rightarrow & \tau := t_{\mathsf{now}} - i^+ \end{array}$$

Verschieben von Absolutzeiten

$$(\mathcal{O}_\mathcal{P}) \in \{+, -\}: \quad \begin{array}{ll} \mathcal{T}_a \times \mathcal{T}_r \longmapsto \mathcal{T}_a, & t, t' \in \mathcal{T}_a,\ \tau \in \mathcal{T}_r \\ \mathcal{I}_a \times \mathcal{T}_r \longmapsto \mathcal{I}_a, & i, i' \in \mathcal{I}_a,\ \tau \in \mathcal{T}_r \end{array}$$

$$\begin{array}{lll} \mathcal{T}_a \times \mathcal{T}_r: & t' := t + \tau\ (= \tau + t): & t = v_a\,[\mathbf{E}]\Big|_{\mathrm{T}}, \tau = v_r\,[\mathbf{E}] \rightarrow t' := (v_a + v_r)\,[\mathbf{E}]\Big|_{\mathrm{T}} \\ & t' := t - \tau: & t = v_a\,[\mathbf{E}]\Big|_{\mathrm{T}}, \tau = v_r\,[\mathbf{E}] \rightarrow t' := (v_a - v_r)\,[\mathbf{E}]\Big|_{\mathrm{T}} \\ \mathcal{I}_a \times \mathcal{T}_r: & i' := i + \tau\ (= \tau + i): & i = [i^-, i^+]\Big|_{\mathrm{T}} \rightarrow i' := [i^- + \tau, i^+ + \tau]\Big|_{\mathrm{T}} \\ & i' := i - \tau: & i = [i^-, i^+]\Big|_{\mathrm{T}} \rightarrow i' := [i^- - \tau, i^+ - \tau]\Big|_{\mathrm{T}} \end{array}$$

Die Addition von relativen mit absoluten Zeiten läßt sich auf zwei Weisen interpretieren, zum einen als Verschieben des absoluten Zeitpunktes um eine Relativzeit (z.B. was war vor einer Minute: $t_{\mathsf{now}} - \tau, \tau = 60\,\mathrm{s}$), zum anderen als Projektion oder Übertragung charakteristischer Zeiten in einen real ablaufenden Prozeß ($t_{1,2,3} = t_{\mathrm{start}} + \tau_{1,2,3}$). Das Verschieben eines absoluten Intervalls um eine Relativzeit läßt sich z.B. als Übertragung eines gemessenen Ereignisses auf einen späteren Zyklus interpretieren, etwa um dessen Verlauf zu prädizieren.

Projektion von Relativintervallen

$$(\mathcal{O}_\mathcal{P}) \in \{+\}: \quad \mathcal{I}_r \times \mathcal{T}_a \longmapsto \mathcal{I}_a, \quad \iota \in \mathcal{I}_r,\ t \in \mathcal{T}_a,\ i' \in \mathcal{I}_a$$

$$i' := \iota + t\ (= t + \iota): \ \iota = [\iota^-, \iota^+] \quad \rightarrow \quad i' := [t + \iota^-, t + \iota^+]$$

Eine Projektion stellt die Übertragung eines mit relativen Zeitangaben beschriebenen Vorgangs in einen realen Prozeß dar. Die Projektion von Relativzeitpunkten wurde bereits im Rahmen der zuvor beschriebenen Addition von Zeitpunkten definiert.

3.3.3 Konstruktionsoperationen

Mit Hilfe von Konstruktionsoperationen lassen sich aus einzelnen Zeitpunkten oder Intervallen neue Intervalle bilden. Zeitpunkte können dafür als Anfangs-, Mittel- oder Endpunkte der zu konstruierenden Intervalle interpretiert werden. Darüber hinaus können Relativzeiten die Dauer des Intervalls bestimmen.

Die meisten Intervallkonstruktoren können in einer restriktiveren und einer toleranteren Variante definiert werden. Die tolerante Version interpretiert immer den kleineren Zeitpunkt als Anfangszeit und den größeren als Endzeitpunkt, erzeugt also in jedem Fall ein gültiges Intervall. Bei der restriktiven Version wird dagegen das Never-Intervall $\emptyset$ zurückgeliefert, wenn der übergebene Startzeitpunkt größer als der Endzeitpunkt ist.

Statt der Operatorschreibweise soll hier die Funktionsschreibweise verwendet werden. Der Index bezeichnet die Semantik der Parameter. s steht für *start*, f für *finish*, c für *center* und d für *duration*. Die restriktive Version der Operatoren wird mit einem '!' im Index gekennzeichnet.

Start- und Endzeitpunkt

$$(\mathcal{O}_\mathcal{P}) \in \{\iota_{\mathsf{sf}}, \iota_{\mathsf{sf!}}\}: \quad \mathcal{T}_r \times \mathcal{T}_r \longmapsto \mathcal{I}_r, \quad \tau_1, \tau_2 \in \mathcal{T}_r,\ \iota \in \mathcal{I}_r$$
$$(\mathcal{O}_\mathcal{P}) \in \{i_{\mathsf{sf}}, i_{\mathsf{sf!}}\}: \quad \mathcal{T}_a \times \mathcal{T}_a \longmapsto \mathcal{I}_a, \quad t_1, t_2 \in \mathcal{T}_a,\ i \in \mathcal{I}_a$$

$$\begin{array}{lll}
\text{relativ}: & \iota := \iota_{\mathsf{sf}}(\tau_1, \tau_2)\ (= \iota_{\mathsf{sf}}(\tau_2, \tau_1)) & \rightarrow \quad \iota := [\min(\tau_1, \tau_2), \max(\tau_1, \tau_2)] \\
& \iota := \iota_{\mathsf{sf!}}(\tau_1, \tau_2) & \rightarrow \quad \iota := \begin{cases} [\tau_1, \tau_2] & \text{für} \quad \tau_1 \leq \tau_2 \\ \emptyset & \text{für} \quad \tau_1 > \tau_2 \end{cases} \\
\text{absolut}: & i := i_{\mathsf{sf}}(t_1, t_2)\ (= i_{\mathsf{sf}}(t_2, t_1)) & \rightarrow \quad i := [\min(t_1, t_2), \max(t_1, t_2)] \\
& i := i_{\mathsf{sf!}}(t_1, t_2) & \rightarrow \quad i := \begin{cases} [t_1, t_2] & \text{für} \quad t_1 \leq t_2 \\ \emptyset & \text{für} \quad t_1 > t_2 \end{cases}
\end{array}$$

Die Übergabe von Start- und Endzeitpunkt stellt die einfachste Form dar, ein Intervall zu erzeugen. In der toleranten Version i_{sf} bzw. ι_{sf} spielt die Reihenfolge der Parameter keine Rolle, in der restriktiven Form $i_{\mathsf{sf!}}$ ($\iota_{\mathsf{sf!}}$) muß $\tau_1 \leq \tau_2$ bzw. $t_1 \leq t_2$ gewährleistet sein, damit ein gültiges Intervall erzeugt wird.

Zeitpunkt und Dauer

$$(\mathcal{O}_\mathcal{P}) \in \{\iota_{\mathsf{sd}}, \iota_{\mathsf{sd!}}, \iota_{\mathsf{fd}}, \iota_{\mathsf{fd!}}, \iota_{\mathsf{cd}}, \iota_{\mathsf{cd!}}\}: \quad \mathcal{T}_r \times \mathcal{T}_r \longmapsto \mathcal{I}_r, \quad \tau, \tau_d \in \mathcal{T}_r,\ \iota \in \mathcal{I}_r$$
$$(\mathcal{O}_\mathcal{P}) \in \{i_{\mathsf{sd}}, i_{\mathsf{sd!}}, i_{\mathsf{fd}}, i_{\mathsf{fd!}}, i_{\mathsf{cd}}, i_{\mathsf{cd!}}\}: \quad \mathcal{T}_a \times \mathcal{T}_r \longmapsto \mathcal{I}_a, \quad t \in \mathcal{T}_a,\ \tau_d \in \mathcal{T}_r,\ i \in \mathcal{I}_a$$

absolut:		für $\tau_d \geq 0$ s	für $\tau_d < 0$ s
$i := i_{\mathsf{sd}}(t, \tau_d)$	$\rightarrow$	$i := [t, t + \tau_d]$	$i := [t + \tau_d, t]$
$i := i_{\mathsf{fd}}(t, \tau_d)$	$\rightarrow$	$i := [t - \tau_d, t]$	$i := [t, t - \tau_d]$
$i := i_{\mathsf{cd}}(t, \tau_d)$	$\rightarrow$	$i := [t - \tau_d/2, t + \tau_d/2]$	$i := [t + \tau_d/2, t - \tau_d/2]$

restriktiv:

$$i := i_{\mathsf{Op!}}(t, \tau_d), \quad \rightarrow \quad i := i_{\mathsf{Op}}(t, \tau_d) \qquad i := \emptyset \qquad \text{mit } \mathsf{Op} \in \{\mathsf{sd}, \mathsf{fd}, \mathsf{cd}\}$$

Diese Konstruktoroperatoren erlauben es, ein Intervall relativ zu einem gegebenen Zeitpunkt t (bzw. τ) zu erzeugen. Dabei kann dieser Zeitpunkt den Anfang (sd), das Ende (fd) oder die Mitte (cd) des Intervalls markieren. An dieser Stelle werden aufgrund der besseren Übersicht nur die Konstruktoren für absolute Intervalle angegeben. Für die Definition der Operatoren für Relativintervalle sind lediglich alle i durch ι und alle t durch τ zu substituieren. Für die

Spezialfälle $t = t_{\mathsf{now}}$ und $\tau = 0\,\mathsf{s}$ werden eigene Operatoren definiert. Der erste Parameter entfällt in diesem Fall:

absolut, $t = t_{\mathsf{now}}$:		für $\tau_d \geq 0\,\mathsf{s}$	für $\tau_d < 0\,\mathsf{s}$
$i := i_{\mathsf{sd}}(\tau_d)$	$\rightarrow$	$i := [t_{\mathsf{now}}, t_{\mathsf{now}} + \tau_d]$	$i := [t_{\mathsf{now}} + \tau_d, t_{\mathsf{now}}]$
$i := i_{\mathsf{fd}}(\tau_d)$	$\rightarrow$	$i := [t_{\mathsf{now}} - \tau_d, t_{\mathsf{now}}]$	$i := [t_{\mathsf{now}}, t_{\mathsf{now}} - \tau_d]$
$i := i_{\mathsf{cd}}(\tau_d)$	$\rightarrow$	$i := [t_{\mathsf{now}} - \tau_d/2, t_{\mathsf{now}} + \tau_d/2]$	$i := [t_{\mathsf{now}} + \tau_d/2, t_{\mathsf{now}} - \tau_d/2]$

relativ, $\tau = 0\,\mathsf{s}$:		für $\tau_d \geq 0\,\mathsf{s}$	für $\tau_d < 0\,\mathsf{s}$
$\iota := \iota_{\mathsf{sd}}(\tau_d)$	$\rightarrow$	$\iota := [0\,\mathsf{s}, \tau_d]$	$\iota := [\tau_d, 0\,\mathsf{s}]$
$\iota := \iota_{\mathsf{fd}}(\tau_d)$	$\rightarrow$	$\iota := [-\tau_d, 0\,\mathsf{s}]$	$\iota := [0\,\mathsf{s}, -\tau_d]$
$\iota := \iota_{\mathsf{cd}}(\tau_d)$	$\rightarrow$	$\iota := [-\tau_d/2, \tau_d/2]$	$\iota := [\tau_d/2, -\tau_d/2]$

restriktiv :

$i := i_{\mathsf{Op!}}(\tau_d)$	$\rightarrow$	$i := i_{\mathsf{Op}}(\tau_d)$	$i := \emptyset$	mit $\mathsf{Op} \in \{\mathsf{sd}, \mathsf{fd}, \mathsf{cd}\}$
$\iota := \iota_{\mathsf{Op!}}(\tau_d)$	$\rightarrow$	$\iota := \iota_{\mathsf{Op}}(\tau_d)$	$\iota := \emptyset$	

Zeitpunkt und Intervall

$$(\mathcal{O}_{\mathcal{P}}) \in \{\iota_{\mathsf{gap}}, \iota_{\mathsf{span}}, \iota_{\mathsf{divs}}, \iota_{\mathsf{divf}}\} : \quad \mathcal{I}_r \times \mathcal{T}_r \longmapsto \mathcal{I}_r, \quad \iota, \iota' \in \mathcal{I}_r,\ \tau \in \mathcal{T}_r$$
$$(\mathcal{O}_{\mathcal{P}}) \in \{i_{\mathsf{gap}}, i_{\mathsf{span}}, i_{\mathsf{divs}}, i_{\mathsf{divf}}\} : \quad \mathcal{I}_a \times \mathcal{T}_a \longmapsto \mathcal{I}_a, \quad i, i' \in \mathcal{I}_a,\ t \in \mathcal{T}_a$$

absolut :		für $t < i^-$	für $t \in i$	für $t > i^+$
$i' := i_{\mathsf{gap}}(i, t)$	$\rightarrow$	$i' := [t, i^-]$	$i' := \emptyset$	$i' := [i^+, t]$
$i' := i_{\mathsf{span}}(i, t)$	$\rightarrow$	$i' := [t, i^+]$	$i' := [i^-, i^+]$	$i' := [i^-, t]$
$i' := i_{\mathsf{divs}}(i, t)$	$\rightarrow$	$i' := \emptyset$	$i' := [i^-, t]$	$i' := [i^-, i^+]$
$i' := i_{\mathsf{divf}}(i, t)$	$\rightarrow$	$i' := [i^-, i^+]$	$i' := [t, i^+]$	$i' := \emptyset$

Auch hier wurden nur die Definitionen für die absoluten Intervallkonstruktoren angegeben, da sich die Konstruktoren für relative Intervalle aus diesen einfach durch Austausch von i mit ι und t mit τ ableiten lassen.

Das vom i_{gap}-Operator konstruierte Intervall bezeichnet den Zwischenraum zwischen Zeitpunkt und Intervall. Liegt der angegebene Zeitpunkt innerhalb des Intervalls, gibt es keinen Zwischenraum, folglich wird $\emptyset$ zurückgeliefert. Der gesamte Zeitraum, der von den angegebenen Zeiten überspannt wird, wird vom i_{span}-Operator geliefert. Liegt der angegebene Zeitpunkt innerhalb des Intervalls, teilen die $i_{\mathsf{div}*}$-Operatoren schließlich das Intervall an der durch den Zeitpunkt definierten Stelle in zwei Teilintervalle auf. i_{divs} liefert das erste Teilintervall, i_{divf} das zweite.

Neben den hier angegeben Konstruktoren lassen sich mit i_{sf} und $i_{\mathsf{sf!}}$ bzw. ι_{sf} und $\iota_{\mathsf{sf!}}$ weitere Intervalle direkt aus dem angegeben Zeitpunkt t bzw. τ und einem der Intervallzeitpunkte i^-, i^+ bzw. ι^- oder ι^+ bilden.

Zwei Intervalle

$$(\mathcal{O}_{\mathcal{P}}) \in \{\iota_{\mathsf{span}}, \iota_{\mathsf{gap}}, \iota_{\mathsf{over}}\} : \quad \mathcal{I}_r \times \mathcal{I}_r \longmapsto \mathcal{I}_r, \quad \iota_1, \iota_2, \iota' \in \mathcal{I}_r$$
$$(\mathcal{O}_{\mathcal{P}}) \in \{i_{\mathsf{span}}, i_{\mathsf{gap}}, i_{\mathsf{over}}\} : \quad \mathcal{I}_a \times \mathcal{I}_a \longmapsto \mathcal{I}_a, \quad i_1, i_2, i' \in \mathcal{I}_a$$

absolut :

$$i' := i_{\mathsf{span}}(i_1, i_2) \;\rightarrow\; i' := [\min(i_1^-, i_2^-), \max(i_1^+, i_2^+)]$$

$$i' := i_{\mathsf{gap}}(i_1, i_2) \;\rightarrow\; i' := \begin{cases} [\min(i_1^+, i_2^+), \max(i_1^-, i_2^-)] & \text{für} \quad i_1^+ < i_2^- \vee i_2^+ < i_1^- \\ \emptyset & \text{sonst} \end{cases}$$

$$i' := i_{\mathsf{over}}(i_1, i_2) \;\rightarrow\; i' := \begin{cases} [\max(i_1^-, i_2^-), \min(i_1^+, i_2^+)] & \text{für} \quad i_1^- \leq i_2^+ \wedge i_2^- \leq i_1^+ \\ \emptyset & \text{sonst} \end{cases}$$

Mit Hilfe dieser Operatoren lassen sich Zeiträume ausdrücken, die durch das Verhältnis zweier Intervalle zueinander definiert werden. Im einzelnen sind das der gesamte, von beiden Intervallen überspannte Zeitraum ($i_{\mathsf{span}}, \iota_{\mathsf{span}}$), der Zwischenraum zwischen den Intervallen, insofern es einen gibt ($i_{\mathsf{gap}}, \iota_{\mathsf{gap}}$), und — wenn es diesen Zwischenraum nicht gibt — der Bereich, in dem sich beide Intervalle überlagern ($i_{\mathsf{over}}, \iota_{\mathsf{over}}$). Weitere Intervalle, wie z.B. Intervalle über beide Anfangs- oder beide Endzeitpunkte, lassen sich leicht mit Hilfe der zeitpunktbasierten Konstruktoren i_{sf} und $i_{\mathsf{sf!}}$ bzw. ι_{sf} und $\iota_{\mathsf{sf!}}$ bilden. So erzeugt beispielsweise $i_{\mathsf{sf}}(i_1^-, i_2^-)$ ein Intervall über beide Anfangspunkte, und $i_{\mathsf{sf!}}(i_1^+, i_2^+)$ erzeugt, falls der erste Endpunkt vor dem zweiten liegt, ein Intervall über beide Endpunkte. Alle für absolute Intervalle angegebenen Definitionen sind auch auf Relativintervalle übertragbar.

3.3.4 Zugriffsoperationen

Abgeschlossen werden soll dieser Abschnitt mit der Definition von Operationen, die den Zugriff auf die charakteristischen Intervallgrößen ermöglichen. Diese Größen sind der Anfangs-, der Endzeitpunkt und die Dauer.

Anfangszeitpunkt

$$(\mathcal{O}_\mathcal{P}) \in \{\tau_{\mathsf{s}}\}: \quad \mathcal{I}_r \longmapsto \mathcal{T}_r,\ \iota \in \mathcal{I}_r, \tau \in \mathcal{T}_r,\ : \quad \text{relativ:} \quad \tau := \tau_{\mathsf{s}}(\,\iota\,) \;\rightarrow\; \tau := \iota^-$$
$$(\mathcal{O}_\mathcal{P}) \in \{t_{\mathsf{s}}\}: \quad \mathcal{I}_a \longmapsto \mathcal{T}_a,\ i \in \mathcal{I}_a, t \in \mathcal{T}_a,\ : \quad \text{absolut:} \quad t := t_{\mathsf{s}}(\,i\,) \;\rightarrow\; t := i^-$$

Endzeitpunkt

$$(\mathcal{O}_\mathcal{P}) \in \{\tau_{\mathsf{f}}\}: \quad \mathcal{I}_r \longmapsto \mathcal{T}_r,\ \iota \in \mathcal{I}_r,\ \tau \in \mathcal{T}_r,\ : \quad \text{relativ:} \quad \tau := \tau_{\mathsf{f}}(\,\iota\,) \;\rightarrow\; \tau := \iota^+$$
$$(\mathcal{O}_\mathcal{P}) \in \{t_{\mathsf{f}}\}: \quad \mathcal{I}_a \longmapsto \mathcal{T}_a,\ i \in \mathcal{I}_a,\ t \in \mathcal{T}_a,\ : \quad \text{absolut:} \quad t := t_{\mathsf{f}}(\,i\,) \;\rightarrow\; t := i^+$$

Dauer

$$(\mathcal{O}_\mathcal{P}) \in \{\tau_{\mathsf{d}}\}: \quad \begin{array}{l} \mathcal{I}_r \longmapsto \mathcal{T}_r,\ \iota \in \mathcal{I}_r,\ \tau \in \mathcal{T}_r,\ : \quad \text{relativ:} \quad \tau := \tau_{\mathsf{d}}(\,\iota\,) \;\rightarrow\; \tau := \iota^+ - \iota^- \\ \mathcal{I}_a \longmapsto \mathcal{T}_r,\ i \in \mathcal{I}_a,\ \tau \in \mathcal{T}_r,\ : \quad \text{absolut:} \quad \tau := \tau_{\mathsf{d}}(\,i\,) \;\rightarrow\; \tau := i^+ - i^- \end{array}$$

Kapitel 4

Dynamische Basiselemente in Bildfolgenprogrammen

Nachdem im vorhergehenden Kapitel ein Instrumentarium für die Repräsentation von Zeit und zeitlichen Ausdrücken entwickelt wurde, werden nun die dynamischen Basiselemente von Bildfolgen- und Sensordatenprogrammen modelliert und hinsichtlich ihres Zeitverhaltens charakterisiert. Das Ziel ist dabei nicht die Einführung von speziell auf die Bildverarbeitung zugeschnittenen Datentypen wie Bild- oder Regionensequenzen, sondern vielmehr die Modellierung allgemeiner Eigenschaften von Datenfolgen und Funktionen, unabhängig von einem bestimmten Anwendungsgebiet, konkreten Datentypen und Datenverarbeitungsmethoden. Damit werden die Voraussetzungen geschaffen für die Repräsentation und effiziente Entwicklung von Programmen, mit denen dynamische Sensordaten, insbesondere Bildfolgen verarbeitet werden können.

Die Folgen oder Sequenzen der zyklisch zu aktualisierenden Daten bilden den Ausgangspunkt der Untersuchungen. Die Datenwerte einer solchen Sequenz beschreiben *zusammen* den Verlauf eines diskreten Signals bzw. der daraus extrahierten Merkmale. Sie werden in dieser Arbeit als Sequenz bezeichnet und gemeinsam als *ein* Datenobjekt $\mathbf{S}$ modelliert.

Mit Hilfe von Sensoren, Funktionen und Operatoren werden neue Datenwerte bereitgestellt, verarbeitet und ausgewertet. Auch wenn diese Funktionen jedesmal mit anderen Werten aufgerufen werden, besitzen sie einen relativ konstanten Verarbeitungskontext, der durch einen Datenflußgraphen beschrieben werden kann. Für die Einbindung der Operatoren in einen solchen Graphen spielt die Funktionalität nur eine untergeordnete Rolle. Dem wird mit einem abstrakten Funktionsmodell entsprochen. Die so modellierten Operatorobjekte werden in dieser Arbeit als *Funktoren* $\mathbf{F}$ bezeichnet. Sie bilden die zweite zentrale Objektgruppe, für die in diesem Kapitel geeignete Modelle bereitgestellt werden.

Die Schnittstelle zur Umgebung und damit zum äußeren Prozeß bilden Sensoren, indem sie regelmäßig bestimmte Zustandsgrößen der Umgebung messen, aufbereiten und dem System für die weitere Verarbeitung oder Auswertung zur Verfügung stellen. Da die physikalischen Sensoren vom Softwaresystem über Funktionen angesprochen werden, können sie in den Datenfluß- und Verarbeitungskontext ebenfalls mit Hilfe von Funktoren $\mathbf{F}$ integriert werden. Dabei ist es unerheblich, in welchem Umfang sie bereits eine Datenvorverarbeitung durchführen. Das Zeitverhalten einer solchen Einbindung und das Format der bereitgestellten Daten hängt jedoch nicht mehr nur von der Arbeitsweise des physikalischen Sensors ab, sondern auch von den nach-

folgenden Signaltransformationen und Datenverarbeitungsmethoden. Dem wird durch das Modell des *logischen* oder *virtuellen Sensors* entsprochen. Dieses Modell erlaubt die Repräsentation allgemeiner Sensoreigenschaften, etwa des Zeitverhaltens oder der Fähigkeit, kontinuierlich bzw. auf Anfrage Daten bereitzustellen. Es abstrahiert dabei von den konkreten physikalischen Zusammenhängen. Somit aber kann jede Funktionseinheit, die die Daten eines physikalischen oder logischen Sensors aufbereitet, weiterverarbeitet und anderen Komponenten zur Verfügung stellt, wiederum als (logischer) Sensor aufgefaßt werden. Diese Betrachtungsweise wird im vorgestellten Ansatz dadurch unterstützt, daß der Zugriff auf alle Funktoren gleich ist, unabhängig davon, ob sie Funktionen oder Sensoren kapseln. Alle Funktor-Ausgangsdaten $\{\mathbf{S}_{\mathbf{Out}}\}$ können somit auch als Sensordaten interpretieren werden.

Abgeschlossen wird die Liste der Basiselemente durch ein Agentenmodell $\mathbf{A}$. Als Agenten werden aktive Objekte bezeichnet, die basierend auf ihren rollenspezifischen Fähigkeiten sowie explizit angesammelten Wissens autonom Entscheidungen treffen und das System gezielt beeinflussen können. Das in dieser Arbeit vorgestellte Agentenmodell begnügt sich mit elementaren Datenstrukturen und Fähigkeiten für die flexible Ansteuerung und Konfiguration der aus Datensequenzen und Funktoren bestehenden Datenflußgraphen. Dadurch kann die Sensordatenverarbeitung von anderen Programmkomponenten vollkommen losgelöst und als logischer Sensor betrachtet werden.

Die in diesem Abschnitt definierten Objektmodelle bilden die Grundlage für die Softwarerepräsentation von Datensequenzen und Funktoren. Aus diesem Grund werden für diese Objekte neben den Attributen elementare Aktionen definiert. Sie erlauben es, zwischen dem mit Hilfe von Gleichungen beschreibbaren Datenfluß und konkreten Vorgängen wie Datenzugriffen und Funktionsaufrufen zu unterscheiden. Durch die Definition von Aktionen wird der Objektaspekt der Daten- und Funktionsmodelle unterstrichen. Die Aktionen benötigen Rechenzeit und können weitere Aktionen der beteiligten Objekte nach sich ziehen.

Die in diesem Kapitel definierten Objekte mit ihren temporalen Attributen werden im Kapitel 5 hinsichtlich ihres Zeitverhaltens charakterisiert. Die Umsetzung funktionaler Beschreibungen in Datenflußgraphen und deren Ansteuerung ist Gegenstand des Kapitels 6.

4.1 Daten- und Verarbeitungszyklen

Die rechnerbasierte Verarbeitung von Sensordaten ist dadurch gekennzeichnet, daß die Zugriffe auf die Sensoren und die Szene nicht kontinuierlich, sondern nur zu diskreten Zeitpunkten entsprechend der Daten- bzw. Meßzyklen erfolgen. Damit läßt sich für jede Datensequenz eine Folge von Meßzeitpunkten $T_{\mathsf{g}} = \{t^i_{\mathsf{g}}\}$ (*grab time*) bestimmen. Die Auswertung der Daten erfolgt ebenfalls zyklisch. Diese Bearbeitungszyklen sind an die diskreten Zeitpunkte $T_{\mathsf{c}} = \{t^i_{\mathsf{c}}\}$ (*cycle time*) gebunden sind. Verschiedene Komponenten oder einzelne Sensoren können dabei ihre eigenen, von anderen Komponenten weitestgehend unabhängigen Iterationszyklen besitzen: $T_{\mathsf{g,Camera}} \neq T_{\mathsf{g,Odometry}} \neq T_{\mathsf{c,SelfLoc}} \neq T_{\mathsf{c,Calib}}$.

Um zu beschreiben, auf welchen Zyklus sich bestimmte Angaben beziehen, können die Iterationen von Beginn an durchgezählt werden, oder es wird vom aktuellen Zyklus (I) ausgegangen, und die Daten werden in Relation zu diesem gesetzt. Der aktuelle Zyklus wird im ersten Fall mit dem Index $i = I$ (absolut) und im zweiten mit $i = 0$ (relativ) bezeichnet. Die

Elemente einer Folge werden mit einer hochgestellten Zyklusnummer unterschieden. Diese laufen entweder absolut von 1 bis zum aktuellen Zyklus I oder relativ von $-I+1$ bis 0. Bei der relativen Indizierung werden mit jedem neuen Zyklus alle Indizes der Folge dekrementiert. Eine Folge von Meßzeitpunkten wird damit wie folgt dargestellt:

$$\begin{array}{ll} \text{absolut:} & T_{\mathsf{g}} = \{t_{\mathsf{g}}^1, t_{\mathsf{g}}^2, \ldots, t_{\mathsf{g}}^{I-1}, t_{\mathsf{g}}^I, \ldots\} \qquad t_{\mathsf{g}}^k < t_{\mathsf{g}}^l \leftrightarrow k < l \\ \text{relativ:} & T_{\mathsf{g}} = \{t_{\mathsf{g}}^{1-I}, t_{\mathsf{g}}^{2-I}, \ldots, t_{\mathsf{g}}^{-1}, t_{\mathsf{g}}^0, \ldots\} \\ & \text{neues Element } t_{\mathsf{g}}\text{: } t_{\mathsf{g}} > t_{\mathsf{g}}^0 \quad \Rightarrow \quad \forall t_{\mathsf{g}}^i \in T_{\mathsf{g}} : \; i := i-1; \; t_{\mathsf{g}}^0 := t_{\mathsf{g}}; \; I := I+1; \end{array}$$

Wird eine Datenfolge — etwa bei der Offline-Bildfolgenanalyse — als ganzes betrachtet, stellt die absolute Zählung u.U. die geeignetere Zählweise dar. Online-Datenzugriffe lassen sich jedoch besser mit Hilfe der relativen Zählung beschreiben, da alle Aktionen vom aktuellen Aufnahme- bzw. Bearbeitungszyklus ausgehen. Zu diesem Zeitpunkt ist immer nur ein Ausschnitt der Folge, d.h. die jüngste Vergangenheit interessant und sichtbar. Es interessiert i.d.R. nicht, das wievielte Datum gerade bearbeitet wird, sondern wie alt die Daten sind, und ob sie aktuell sind. Dafür ist es notwendig, einen Zusammenhang herzustellen zwischen dem inneren Zustand einer solchen Folge und den von außen vorgegebenen Bearbeitungszyklen. Die absoluten Indizes sind dafür kaum geeignet — zum einen müßten alle zugreifenden Komponenten die Historie, d.h. die Zyklen der Sensordaten von Beginn an mitverfolgen, zum anderen können sich diese Indizes für verschiedene Daten, die unabhängig voneinander bereitgestellt werden, unterscheiden. Relative Indizes im Zusammenhang mit Zeitinformationen sind für diese Aufgabe besser geeignet, da sie unabhängig von der Historie der beteiligten Objekte sind.

4.2 Datenwerte und Sequenzen

4.2.1 Das Sequenzmodell

Ein zentraler Aspekt des hier vorgestellten Ansatzes ist, daß der Signalcharakter von Datenfolgen erhalten bleibt und explizit modelliert wird. Ein neuer Datenwert überschreibt daher nicht einfach seinen Vorgänger, sondern wird einer Folge mit den zuvor bereits aufgenommenen Werten als neues Element hinzugefügt. Damit sind ältere Werte eines Datums auch weiterhin für andere Komponenten sichtbar. Mit den temporalen Attributen der einzelnen Datenwerte und der Sequenz als Ganzes wird darüber hinaus die Grundlage für zeitgebundene Datenzugriffe gelegt.

Dynamische Datenfolgen werden als Sequenz $\mathbf{S}$ bezeichnet, ihre Elemente heißen Sequenzwerte S. In ihrer Gesamtheit beschreiben sie eine bestimmte Zustandsgröße der Szene über die Zeit. Die Sequenzwerte dienen dabei als Containerobjekt für die konkreten Daten. Neben den reinen Datenwerten enthalten sie allgemeine, die Datenaufnahmen und -verarbeitung charakterisierende Zeitwerte (vgl. dazu Abschnitt 5.1):

$$S = (V, \mathbf{T}, t_{\mathsf{g}}, t_{\mathsf{s}}), \; V \in \mathbf{T}, \; t_{\mathsf{g}}, t_{\mathsf{s}} \in \mathcal{T}_a, \quad \text{mit} \quad \begin{array}{l} V = \text{Datenwert bzw. -objekt} \\ \mathbf{T} = \text{Typ des Datums} \\ t_{\mathsf{g}} = \text{Datenmeßzeit (}\textit{grab time}\text{)} \\ t_{\mathsf{s}} = \text{Datenbereitstellungszeit (}\textit{supply time}\text{)} \end{array}$$

Erweitert werden kann dieses Modell durch zusätzliche Attribute, wie einen Verweis auf den verwendeten Sensor $\mathbf{F}$, die Dauer der Datenerfassung τ_{g} oder den Zyklus t_{c}, in dem das Datum ermittelt wurde.

Für die Kennzeichnung der einzelne Datenelemente innerhalb einer Folge gelten die gleichen Konventionen wie für die zuvor beschriebenen Zeitfolgen: positive Indizes bezeichnen den fortlaufenden Index seit Beginn der Messung, negative Indizes und der Index 0 beziehen sich auf den aktuellen Iterationsschritt. Eine Datensequenz $\mathbf{S}$ läßt sich somit zuerst einmal wie folgt beschreiben:

$$\mathbf{S} = \{S^i\} = \{S^{1-I}, \ldots, S^{-1}, S^0, \ldots\} = \{S^1, \ldots, S^{I-1}, S^I, \ldots\}$$

Weiterhin soll gelten, daß alle Sequenzwerte ein und derselben Sequenz nur Daten des gleichen Typs $\mathbf{T}$ enthalten. Allerdings soll die Möglichkeit bestehen, undefinierte Sequenzwerte der Sequenz hinzuzufügen, etwa um zu kennzeichnen, daß ein Merkmal aus einem bestimmten Bild nicht extrahiert werden konnte. Dies erlaubt es, eine zu einem bestimmten Zeitpunkt durchgeführte Messung, die erfolglos war, von den Zeiten, für die keine Messung erfolgte, zu unterscheiden. Dafür wird ein spezielles, typloses Datenobjekt $\emptyset$ verwendet.

$$\begin{aligned} \forall S^i \in \mathbf{S} \mid S^i = (V^i, \mathbf{T}^i, t^i_{\mathsf{g}}, t^i_{\mathsf{s}}): \quad & \mathbf{T}^i = \mathbf{T} = \mathbf{T}(\mathbf{S}) && \wedge \\ & V^i \in \mathbf{T} \vee V^i = \emptyset && \wedge \\ & i_1 \begin{smallmatrix} < \\ = \\ > \end{smallmatrix} i_2 \;\leftrightarrow\; t^{i_1}_{\mathsf{g}} \begin{smallmatrix} < \\ = \\ > \end{smallmatrix} t^{i_2}_{\mathsf{g}} \;\leftrightarrow\; t^{i_1}_{\mathsf{s}} \begin{smallmatrix} < \\ = \\ > \end{smallmatrix} t^{i_2}_{\mathsf{s}} \end{aligned}$$

Datensequenzen können mit einem Namen versehen werden, der die Semantik der Daten beschreibt. Optional kann nach einem Doppelpunkt der Typ der Daten folgen:

$$\begin{aligned} \mathbf{S}_{\mathbf{Name}[:\mathbf{Type}]} &= \{\ldots, S^{-1}_{Name[:Type]}, S^0_{Name[:Type]}, \ldots\} = \{S^i_{Name[:Type]}\} \\ S^i_{Name:Type} &= (V^i_{Name}, \mathbf{T}_{Type}, t^i_{\mathsf{g}}, t^i_{\mathsf{s}}) \\ \text{Beispiele:} \quad & \mathbf{S}_{\mathbf{Camera:Image}} = \mathbf{S}_{\mathbf{Cam:I}} = \mathbf{S}_{\mathbf{Cam}}, \quad \mathbf{S}_{\mathbf{Edge:Line}}, \quad \mathbf{S}_{\mathbf{ROI:Region}} \end{aligned}$$

Neben dem eigentlichen Datenverlauf wird für jede Sequenz ihr Bearbeitungskontext und der aktuelle Status modelliert. Der Kontext wird von verschiedenen Funktoren gebildet: $\mathbf{F}_{\mathbf{upd}}$ kapselt eine Funktion bzw. einen Sensor, mit dem der Sequenz neue Werte hinzugefügt werden. Der Funktor $\mathbf{F}_{\mathbf{int}}$ dient zur Interpolation von fehlenden Datenwerten. Weitere Funktoren $\{\mathbf{F}_{\mathbf{dep}}\}^n$ können nach jeder Aktualisierung der Sequenz aufgerufen werden. Dabei kann zwischen Operatoren, die nach einer erfolgreichen Aktualisierung $\{\mathbf{F}_{\mathbf{dep}_+}\}^{n_+}$, und welchen, die bei mißglückten Aktualisierungsversuchen ($V = \emptyset$) aufgerufen werden $\{\mathbf{F}_{\mathbf{dep}_-}\}^{n_-}$, unterschieden werden ($\{\mathbf{F}_{\mathbf{dep}_+}\} \cup \{\mathbf{F}_{\mathbf{dep}_-}\} = \{\mathbf{F}_{\mathbf{dep}}\}$).

Der Sequenzstatus wird durch den aktuellen Zyklus (Index I und Zeit t_{c}) sowie die zuletzt aufgerufenen Sequenzaktionen $\{\mathsf{M}\}$ — bestehend aus einer Aktionskennung M, dem aktuellen Bearbeitungsstand state und der Zykluszeit t_{c} — beschrieben (vgl. Abschnitt 4.2.2). Ausgestattet mit diesen Attributen ergibt sich das folgende, erweiterte Sequenzmodell $\mathbf{S}$:

$$\mathbf{S} = \Big(\{S\}^I, \mathbf{F}_{\mathbf{upd}}, \mathbf{F}_{\mathbf{int}}, \{\mathbf{F}_{\mathbf{dep}}\}^n, I, t_{\mathsf{c}}, \{\mathsf{M}\}, \ldots \Big) \quad \text{mit} \quad \mathsf{M} = (M, \mathsf{state}, t_{\mathsf{c}})$$

M : Objektmethode bzw. Aktion

$\mathsf{state} \in \{\mathsf{started}, \mathsf{running}, \mathsf{canceled}, \mathsf{finished}, \mathsf{error}\}$

4.2.2 Elementare Zugriffsmethoden und Aktionen

Damit eine Funktion $F(V_1, \ldots, V_m)$ mit den Daten $V_j : 1 \leq j \leq m$ tatsächlich rechnen kann, müssen zuerst aus den entsprechenden Sequenzen $\mathbf{S}_{\mathbf{X_j}}$ die Sequenzwerte S_j und aus diesen die Datenwerte V_j gelesen werden. Durch das Sequenzmodell werden dafür verschiedene Zugriffsmethoden $\mathsf{get}(\mathbf{S}, i)$ definiert. $\mathbf{S}_{\mathbf{X}}^{(i)}$ stellt dafür eine alternative, kompaktere Schreibweise dar. Durch sie kann zusätzlich ausgedrückt werden, daß mehrere Sequenzwerte einer Sequenz gleichzeitig zu verarbeiten sind: $\mathbf{S}_{\mathbf{X}}^{(0,-1,\ldots)} \equiv \mathbf{S}_{\mathbf{X}}^{(0)}, \mathbf{S}_{\mathbf{X}}^{(-1)}, \ldots$ Diese Schreibweise ist jedoch nicht mit S_X^i zu verwechseln. Während ersteres eine Aktion der Sequenz beschreibt, um auf einen Sequenzwert zuzugreifen, bezeichnet letzteres den Sequenzwert selbst und damit ein Datum.

Die einfachste Form des Datenzugriffs ist $\mathsf{get}_{[]}(\mathbf{S}_{\mathbf{X}}, i)$ — oder kurz $\mathbf{S}_{\mathbf{X}}^{[i]}()$. Diese Methode geht vom aktuellen Zustand der Sequenz aus und liefert direkt einen Wert aus der Liste der verfügbaren Sequenzwerte.

Um bei späteren Zugriffen den gleichen Wert zu erhalten wie zuvor, kann anstelle der aktuellen Zeit t_{now} auch eine Referenzzeit $t : t \leq t_{\mathsf{now}}$ als Bezugsgröße angegeben werden: $\mathbf{S}_{\mathbf{X}}^{[i]}(t)$. In diesem Fall liefert die Anfrage das zur angegebenen Referenzzeit jüngste Element der Sequenz. Bei diesen Datenzugriffen wird nicht von der Datenmeßzeit t_{g} ausgegangen, sondern von der Bereitstellungszeit t_{s} des Datenwerts (vgl. dazu Kapitel 5.1).

Der Index i bezieht sich bei allen Anfragen immer auf den (im Sinne des Zugriffs) aktuellen Wert; $i = 0$ bezeichnet also den im Moment der Anfrage (bzw. zur angegebenen Referenzzeit) letzten Wert in der Sequenz. Ältere Werte werden mit negativen Indizes i bezeichnet.

Einfache Sequenzwertabfrage:

$S := \mathsf{get}_{[]}(\mathbf{S}_{\mathbf{X}}, i) = \mathbf{S}_{\mathbf{X}}^{[i]}() :$

Sequenzwert:	$S := S_X^i$
Datenwert:	$V(S) = V_X^i = V(S_X^i)$
Typ:	$\mathbf{T}(S) = \mathbf{T}_X^i = \mathbf{T}(S_X^i) = \mathbf{T}(\mathbf{S}_{\mathbf{X}})$
Meßzeit:	$t_{\mathsf{g}}(S) = t_{\mathsf{g},\mathsf{X}}^i = t_{\mathsf{g}}(S_X^i) = t_{\mathsf{g}}(V_X^i)$
Bereitstellungszeit:	$t_{\mathsf{s}}(S) = t_{\mathsf{s},\mathsf{X}}^i = t_{\mathsf{s}}(S_X^i)$

Einfache Sequenzwertabfrage mit Referenzzeit:

$$S := \mathsf{get}_{[]}(\mathbf{S}_{\mathbf{X}}, i, t) = \mathbf{S}_{\mathbf{X}}^{[i]}(t) : \;\; S := S_X^k : \quad t_{\mathsf{s}}(S_X^k) < t \quad \wedge \quad t_{\mathsf{s}}(S_X^{k+1}) > t$$

In vielen Fällen reicht eine einfache Sequenzwertabfrage nicht aus, da bei dieser Zugriffsart der Bezug zwischen den Daten und einem Bearbeitungszyklus fehlt. Dafür muß der Datenzugriff direkt an einen bestimmten Zeitwert wie die Datenmeßzeit t_{g} oder Zykluszeit t_{c} gekoppelt werden. Nur mit Hilfe dieser Zugriffsverfahren ist es möglich, bestimmte Datenzeiten einzufordern, um so die Konsistenz und Aktualität der Daten sicherzustellen. Auch in diesem Fall ist es möglich, auf vergangene Werte zuzugreifen, wobei der Index in Relation zu dem durch die angegebene Zeit definierten Sequenzwert gesetzt wird. Die Zugriffsmethoden werden mit $\mathsf{get}_{\mathsf{t}}(\mathbf{S}_{\mathbf{X}}, t)$ bzw. $\mathsf{get}_{\mathsf{t}}(\mathbf{S}_{\mathbf{X}}, t, i)$ bezeichnet, mit den Kurzformen $\mathbf{S}_{\mathbf{X}}(t)$ und $\mathbf{S}_{\mathbf{X}}^i(t)$. Für den Index $i = 0$ sind beide Formen identisch: $\mathsf{get}_{\mathsf{t}}(\mathbf{S}_{\mathbf{X}}, t, 0) = \mathsf{get}_{\mathsf{t}}(\mathbf{S}_{\mathbf{X}}, t) = \mathbf{S}_{\mathbf{X}}^0(t) = \mathbf{S}_{\mathbf{X}}(t)$.

Die Methode $\mathsf{get}_{\mathsf{t}}()$ stellt dabei lediglich die Basisfunktion für zeitabhängige Datenzugriffe dar. Durch spezielle, davon abgeleitete Methoden kann das Zugriffsverhalten für Daten, die so (noch) nicht in der Sequenz enthalten sind, gesteuert werden. Sind alle Sequenzwerte zu alt, kann durch *insistente* Zugriffe mit $\mathsf{get}_{\mathsf{t}}^{\triangleleft}()$ die Aktualisierung der Sequenz aktiv herbeigeführt werden. Eine andere Möglichkeit ist, den Zugriff mit einem *Fehlerstatus* abzubrechen:

$\mathsf{get}_\mathsf{t}^?()$. $\mathsf{get}_\mathsf{t}^{\bowtie}()$ *wartet* dagegen, daß die Sequenz von einer anderen Instanz aus aktualisiert wird. Schließlich ist auch die *Prädiktion* des fehlenden Datums aus den alten Werten möglich: $\mathsf{get}_\mathsf{t}^{\sim}()$. Um statt der Prädiktion eine *Interpolation* des gesuchten Datums aus zwei die Anfragezeit flankierenden Werten vornehmen zu können, kann auch hier auf die Aktualisierung der Sequenz gewartet $\mathsf{get}_\mathsf{t}^{\widetilde{\bowtie}}()$ oder diese aktiv gefordert werden $\mathsf{get}_\mathsf{t}^{\widetilde{\triangleleft}}()$.

Wurde die Sequenz nach dem angegebenen Zeitpunkt bereits aktualisiert und enthält sowohl Daten, die vor, als auch welche, die nach diesem Zeitpunkt gemessen wurden, können diese, falls es für deren Datentyp sinnvoll ist, direkt interpoliert werden: $\mathsf{get}_\mathsf{t}^{\sim}()$. Darüber hinaus kann auch einfach ein Zugriff auf einen der benachbarten Datenwerte erfolgen: $\mathsf{get}_\mathsf{t}^{<}()$ und $\mathsf{get}_\mathsf{t}^{>}()$ oder die Methode mit einem entsprechenden Fehlerwert abgebrochen werden $\mathsf{get}_\mathsf{t}^?()$. Für die Interpolation oder Prädiktion von Datenwerten ist die Angabe eines Relativindexes nicht sinnvoll, bei allen anderen Zugriffen bezieht sich der Index auf den Wert, der der angegebenen Zeit entsprechen würde. Dieser muß dafür nicht notwendigerweise selbst in der Sequenz enthalten sein.

Zeitgebundene Sequenzwertabfrage:

$S := \mathsf{get}_\mathsf{t}(\mathbf{S_X}, t) = \mathbf{S_X}(t)$:

$S := \mathsf{get}_\mathsf{t}(\mathbf{S_X}, t, i) = \mathbf{S}_\mathbf{X}^i(t)$ (mit Relativindex i) :

$\exists S_X^k \in \mathbf{S_X} \mid t_\mathsf{g}(S_X^k) \mathrel{\widehat{=}} t$:

- S hat der Zeit entsprechenden Wert:
  $S := S_X^k$; bzw. $S := S_X^{k+i}$;

$\forall S_X^k \in \mathbf{S_X} \mid t_\mathsf{g}(S_X^k) < t$:

- S ist nicht aktuell genug:

$i = 0$

- aktueller Wert gefordert:
  - Abbrechen: $\mathsf{get}_\mathsf{t}^?(\mathbf{S_X}, t)$:
    $S := \emptyset$;
  - Warten: $\mathsf{get}_\mathsf{t}^{\bowtie}(\mathbf{S_X}, t)$:
    DO
    WAITFOR : $\mathsf{M} = (\mathsf{set}, \mathsf{finished}, t_\mathsf{g})$;
    WHILE $(t_\mathsf{g} < t)$;
    $S := \mathsf{get}_\mathsf{t}^{>}(\mathbf{S_X}, t)$;
  - Aktualisieren: $\mathsf{get}_\mathsf{t}^{\triangleleft}(\mathbf{S_X}, t)$:
    $\mathsf{update}(\mathbf{S_X}, t)$;
    $S := \mathsf{get}_\mathsf{t}^{\bowtie}(\mathbf{S_X}, t)$;
  - Prädizieren: $\mathsf{get}_\mathsf{t}^{\sim}(\mathbf{S_X}, t)$:
    $S := \mathsf{int}(\mathbf{S_X}, t)$;
  - Interpolieren: $\mathsf{get}_\mathsf{t}^{\widetilde{\bowtie}}(\mathbf{S_X}, t)$:
    WAITFOR : $\mathsf{set}(\mathbf{S_X}, t_\mathsf{g})$;
    $S := \mathsf{int}(\mathbf{S_X}, t)$;

    $\mathsf{get}_\mathsf{t}^{\widetilde{\triangleleft}}(\mathbf{S_X}, t)$:
    $\mathsf{update}(\mathbf{S_X}, t)$;
    $S := \mathsf{int}(\mathbf{S_X}, t)$;

$i < 0$

- alter Wert gefordert:
  $S := S_X^{i+1}$;

$\exists S_X^k \in \mathbf{S_X} \mid t_\mathsf{g}(S_X^k) > t \wedge t_\mathsf{g}(S_X^{k-1}) < t$:

- kein Datum mit der geforderten Meßzeit:
  - Abbrechen: $\mathsf{get}_\mathsf{t}^?(\mathbf{S_X}, t, i)$, $\mathsf{get}_\mathsf{t}^{\triangleleft}(\mathbf{S_X}, t, i)$: $S := \emptyset$;
  - Vorgänger: $\mathsf{get}_\mathsf{t}^{<}(\mathbf{S_X}, t, i)$: $S := S_X^{k-1+i}$;
  - Nachfolger: $\mathsf{get}_\mathsf{t}^{>}(\mathbf{S_X}, t, i)$, $\mathsf{get}_\mathsf{t}^{\bowtie}(\mathbf{S_X}, t, i)$: $S := S_X^{k+i}$;
  - Interpolieren: $\mathsf{get}_\mathsf{t}^{\sim}(\mathbf{S_X}, t, i)$: $S := \mathsf{int}(\mathbf{S_X}, t)$;

Welche Zeitwerte im einzelnen von Bedeutung sind, wird in den folgenden Kapiteln genauer untersucht. Der Zugriff auf die eigentlichen Datenwerte eines Sequenzwertes erfolgt, wie oben bereits gezeigt, einheitlich mit $V(S)$ und bleibt damit vollkommen unabhängig von der Methode, mit der der Sequenzwert bereitgestellt wurde.

Neben den reinen Zugriffsmethoden werden weitere Aktionen für Sequenzen definiert. Dazu gehören vor allem die Aktualisierung der Sequenz: $\mathsf{update}()$ und die Interpolation fehlender Werte: $\mathsf{int}()$. Weiterhin muß es den Funktoren möglich sein, über eine geeignete $\mathsf{set}()$-Methode der Sequenz neue Datenwerte hinzuzufügen. Dafür werden zuerst die internen Statusgrößen und die Datenliste angepaßt: $\mathsf{add}()$. Daran anschließend hat die Sequenz die Möglichkeit, andere Funktoren zu triggern, um so die neuen Sequenzwerte im Datenflußgraphen weiterzuleiten: $\mathsf{prop}()$.

Setzen eines neuen Sequenzwerts:

$\mathsf{set}(\mathbf{S_X}, t, S):$ $\mathsf{add}(\mathbf{S_X}, t, S);$
- Aktualisieren der internen Sequenzdaten

$\mathsf{prop}(\mathbf{S_X});$
- Weiterreichen des neuen Sequenzelementes

Einen neuen Sequenzwert der internen Liste hinzufügen:

$\mathsf{add}(\mathbf{S_X}, t, S):$ $\forall S_X^i \in \mathbf{S_X} : \; i := i - 1;$
$S_X^0 := S;$
$I := I + 1;$

Propagieren eines neues Sequenzwertes:

$\mathsf{prop}(\mathbf{S_X}):$ $\forall \mathbf{F}_{\mathbf{dep_j}} \in \{\mathbf{F_{dep}}\}^n : \; \mathsf{call}(\mathbf{F}_{\mathbf{dep_j}}, t_{\mathsf{g}}(S^0));$
- Aufruf aller von $\mathbf{S_X}$ abhängigen Funktionen

Aktualisieren der Sequenz einleiten:

$\mathsf{update}(\mathbf{S_X}, t):$ $\mathsf{call}(\mathbf{F_{upd}}, t);$
- Aufruf des entsprechenden Sensors $\mathbf{F_{upd}}$, dieser setzt mit $\mathsf{set}()$ den neuen Wert.

Interpolation von Sequenzwerten:

$S := \mathsf{int}(\mathbf{S_X}, t) = \mathbf{S}_{\mathbf{X}}^{\approx}(t):$ $\mathsf{call}(\mathbf{F_{int}}, t);$
- Aufruf des Interpolationsfunktors $\mathbf{F_{int}}(S_1, S_2, t)$

mit $S_1 := S_X^i$ und $S_2 := S_X^{i+1}$: $\quad t_{\mathsf{g}}(S_X^i) < t < t_{\mathsf{g}}(S_X^{i+1})$

4.3 Sensoren, Operatoren und Funktoren

4.3.1 Physikalische Sensoren

Eine zentrale Rolle in den hier betrachteten Systemen spielen Sensoren. Gemeinsam mit den Aktoren bilden sie die Schnittstelle zwischen der Umgebung und dem Rechner. Ihre Aufgabe ist es, Daten, die den aktuellen Zustand der Szene beschreiben, zu ermitteln und dem Softwaresystem zur Verfügung zu stellen. Dafür greifen sie zu diskreten Zeitpunkten auf die Szene zu,

messen den Wert oder Verlauf physikalischer Größen und stellen das Ergebnis in digitalisierter Form dem Rechner zur Verfügung.

Im engeren Sinn versteht man unter Sensoren lediglich den Meßfühler, also nur den Teil eines Meßgerätes, der unmittelbar der Umgebung ausgesetzt wird und die zu beobachtende physikalische Größe in eine leichter weiterzuverarbeitende Form — meist in ein elektrisches Signal — umwandelt. Zu diesen relativ einfachen physikalischen Sensoren gehören beispielsweise Taster oder Drucksensoren, Temperaturmeßfühler, CCD-Elemente und Infrarotsensoren.

Häufig wird, insbesondere im Zusammenhang mit elektronischer Datenverarbeitung, der Begriff des physikalischen Sensors weiter gefaßt und auch für die Bezeichnung von mehr oder weniger komplexen Geräten verwendet, die die gemessenen Szenendaten bereits in eine für den Rechner geeignete oder zumindest in eine standardisierte Form umwandeln. Neben den Meßfühlern für die Daten- oder Signaltransformation können diese Geräte Komponenten enthalten, die die Messung durchführen, steuern sowie bereits eine gewisse Aufbereitung der Daten vornehmen, etwa um die Daten leichter handhabbar zu machen, sie zu sequentialisieren oder um sie an abstraktere Standards anzupassen. Typische Vertreter derartiger Sensorgeräte sind CCD-Kameras oder Laserrangefinder.

In jedem Fall muß ein Sensor, der in einem Computersystem verwendet werden soll, in geeigneter Weise repräsentiert werden, damit die Software darauf zugreifen und die Werte abfragen kann. Zwischen dem eigentlichen Hardwaresensor und den durch die Software auswertbaren Daten liegen i.d.R. verschiedene Repräsentationsebenen. Beim Sensor „Kamera“ sind das beispielsweise die optische Projektion auf den CCD-Chip, das Video-Signal, die Digitalisierung im Framegrabber und schließlich die Bildrepräsentation entsprechend der verwendeten Programmiersprache bzw. Softwarebibliothek. Jeder Wechsel der Repräsentationsform kann mit bestimmten Veränderungen und Modifikationen der Sensordaten einhergehen. Dies wirkt sich i.d.R. sowohl auf die Meßwerte als auch auf die Zeitcharakteristik der Daten aus.

4.3.2 Logische Sensoren

Die zuvor beschriebenen komplexen Sensorgeräte beschreiben bereits den Übergang zu einem abstrakteren Sensorbegriff — den logischen oder auch virtuellen Sensoren. Die wichtigste Eigenschaft, die alle Sensoren besitzen, ist, daß sie, indem sie eine bestimmte Zustandsgröße der Szene erfassen und in eine geeignete Form umwandeln, Informationen über die Umgebung liefern. Welche Arbeitsschritte oder Operationen dafür notwendig sind, und ob die Daten eine direkte physikalische Entsprechung besitzen, ist für die Konsumenten der Daten i.d.R. unerheblich. So kann der Sensor zum einen eine einfache softwaretechnische Aufbereitung der Informationen und deren Bereitstellung in Form von Datenobjekten vornehmen, er kann aber auch beliebig komplexe Abfolgen von Datentransformation durchführen.

Als Konsequenz daraus können alle Geräte und Operatoren, die Daten über die Umgebung liefern, als (logische) Sensoren betrachtet werden können. Das schließt physikalische Sensoren mit einer entsprechenden Softwareeinbindung ebenso ein, wie nachgeschaltete Softwaremodule, die die Sensordaten aufbereiten oder die Daten bereits entsprechend einer gestellten Aufgabe modifizieren, filtern oder segmentieren. Allgemein lassen sich Sensoren somit über Funktionen F repräsentieren, die in irgendeiner Weise vom Zustand der Szene abhängen. Eine solche Zustandsgröße kann z.B. das mittels Projektion der Objekte einer Szene s durch ein optisches System entstehende Abbild E sein oder die Entfernung d zwischen einem Ultraschallsensor und

dem nächsten Hindernis. Möglich sind aber auch virtuelle Sensoren, die z.B. in Videobildern $\{V\}$ Grauwertkanten oder den optischen Fluß ermitteln:

Optische Projektion in die Bildebene:	F_{Camera}	$\equiv F_{Camera}(E,t)$	$\equiv F_{Camera}(s,t)$
Entfernung zum Ultraschallsensor:	F_{Sonar}	$\equiv F_{Sonar}(d,t)$	$\equiv F_{Sonar}(s,t)$
Grauwertkanten:	F_{Edges}	$\equiv F_{Edges}(V,t)$	$\equiv F_{Edges}(s,t)$
Optischer Fluß:	$F_{OptFlow}$	$\equiv F_{OptFlow}(V,V^{-1},t)$	$\equiv F_{OptFlow}(s,t)$

4.3.3 Operatoren und Funktionen

Für Komponenten, die die von einem Sensordatenmodul bereitgestellten Datensequenzen weiterverarbeiten oder analysieren, bietet die Sensorsichtweise auf die Daten eine adäquate Darstellungsform, da bei ihr von der internen Funktionsweise und der konkreten Realisierung der Sensordatenmodule abstrahiert wird. Für die Realisierung dieser Module spielt die Bereitstellung, Auswahl und Art der Einbindung von geeigneten Funktionalitäten jedoch eine wichtige Rolle. Softwarebibliotheken stellen grundlegende Verfahren in Form von Operatoren oder Funktionen — die Begriffe sollen hier synonym verwendet werden — zur Verfügung. Mit Hilfe von Bildverarbeitungsoperatoren lassen sich beispielsweise aus einer Videobildfolge bestimmte Bildmerkmale extrahieren, um sie anderen Komponenten zur Verfügung zu stellen. Derartige konkrete Operatoren werden hier mit F bezeichnet. Sie sind dadurch gekennzeichnet, daß sie m Eingangsdatenwerten $\{V_{in}\}^m$ in n Ausgangsdatenwerte $\{V_{out}\}^n$ abbilden. $m \times n$ bezeichnet die *Kardinalität* des Funktors. Die für die Eingangs- und die Ausgangsdaten geforderten Datentypen bilden dessen *Signatur*.

$$
\begin{array}{lll}
F^{m\times n}: & \{V_{in} |\ V_{in} \in \mathbf{T}_{in}\}^m \longmapsto \{V_{out} |\ V_{out} \in \mathbf{T}_{out}\}^n, & \\
& m \times n & \text{– Kardinalität,} \\
& (\{\mathbf{T}_{in}\}^m \times \{\mathbf{T}_{out}\}^n) & \text{– Signatur.}
\end{array}
$$

Neben den zu verarbeitenden Sensordaten können Operatoren zusätzliche Steuerparameter besitzen, mit denen sich ihre Arbeitsweise modifizieren, einstellen oder parametrisieren läßt. Dabei kann es sich beispielsweise um Filtermasken oder Schwellwerte für Bildverarbeitungsoperatoren handeln. Diese spielen für die hier betrachteten Datenflußbeziehungen zwischen dynamischen Datensequenzen keine Rolle. Änderungen dieser Parameter können dabei als Austausch bzw. als erneute Instanziierung der gesamten Funktion interpretiert und so aus der Datenflußebene herausgehalten werden. Eine Funktion $F(V_{Image}, \mathsf{Thresh})$ beispielsweise läßt sich auch als Menge von Funktionen $\{F_{Thresh}(V_{Image})\}$ darstellen, die jeweils nur noch vom Eingangsbild V_{Image} abhängen. Ändert sich im i-ten Zyklus der Parameter Thresh von th_1 auf th_2, entspricht das einem Austausch der Funktion $F^i = F_{Thresh=th_1}(V_{Image})$ durch $F^{i+1} = F_{Thresh=th_2}(V_{Image})$, wobei beide Funktoren die gleiche Kardinalität 1×1 und die gleiche Signatur $(\{\mathbf{T}_{Image}\} \times \{\mathbf{T}_{Region}\})$ besitzen.

4.3.4 Das Funktormodell

Die konkreten Funktionalitäten der Operatoren und Sensoren spielen in dieser Arbeit keine besondere Rolle — ebensowenig wie die von den Sequenzwerten gekapselten konkreten Datentypen und -werte (vgl. Abschnitt 4.2). Hier interessieren vor allem allgemeine Beschreibungsformen, mit denen sich die Beziehungen zwischen den verschiedenen Datensequenzen

und Operationen darstellen lassen. Dafür werden die Funktionalitäten, die die Daten ineinander überführen, als abstrakte Operatoren modelliert. Sie werden zum einen durch ihre Aufgabe — etwa „extrahiere im jeweils aktuellen Bild den Ball“ — und zum anderen durch ihren Kontext im Datenflußgraphen — z.B. zwischen Bild- und Ballsequenz, eventuell mit einem Suchbereich als weiterer Eingangsgröße — charakterisiert, nicht jedoch durch ihre interne Arbeitsweise. Diese abstrakten Operatoren fungieren dabei als eine Art Platzhalter oder Container für konkrete Funktionen. Aufgrund dieser Aufgabe sowie der für Objekte typischen Eigenschaft, mit anderen Objekten Beziehungen eingehen zu können, sollen sie von den eigentlichen Funktionen unterschieden und im folgenden als *Funktoren* $\mathbf{F}$ bezeichnet werden.[1]

Ein Funktor $\mathbf{F}$ beschreibt damit zum einen eine Klasse der für eine bestimmte Aufgabe einsetzbaren Funktionen $F : F \in \mathbf{F}$; zum anderen kapselt er für jeden konkreten Funktoraufruf im Zyklus i eine bestimmte Funktion aus dieser Klasse: $F^i = F(t_\mathsf{c}^i);\ \{F^i\} \subset \mathbf{F}$. Formal kann diese Funktionsklasse durch die Signatur und die Kardinalität der Funktionen beschrieben werden. Diese Größen sind für den Funktor und alle gekapselten Funktionen gleich. In der Praxis wird die Funktionsmenge aufgrund der gestellten Aufgabe und der unterschiedlichen Funktionalitäten weiter eingeschränkt.

Wie schon die zuvor beschriebenen Sequenzen werden auch Funktoren über ihren Kontext charakterisiert. Dazu gehören in erster Linie die M Eingangs- und N Ausgangsdatensequenzen $\{\mathbf{S_{In}}\}^M, \{\mathbf{S_{Out}}\}^N$. Da die von einem Funktor $\mathbf{F}$ gekapselten Funktionen F auch ältere Werte einer Sequenz als Eingangsdaten verarbeiten können sowie aufgrund der verschiedenen Zugriffsmöglichkeiten auf die Sequenzen, ist eine Abbildungsvorschrift $\mathbf{Get}$ notwendig, die beschreibt, welche Daten aus den M Eingangsdatensequenzen die m Eingangsdaten von F bilden. Für die Ausgangsdaten ist eine vergleichbare Abbildungsvorschrift nicht notwendig, da diese Abbildung immer eindeutig ist: der k-te Ausgangswert von F wird in die k-te Ausgangssequenz des Funktors $\mathbf{S_{Out_k}}$ als aktueller Wert eingefügt. Hinzu kommen schließlich Statusinformationen über den aktuellen Zyklus (I und t_c) und die laufenden Aktionen mit ihrem jeweiligen Bearbeitungsstand $\{\mathsf{M}\}$. Mit diesen Attributen ergibt sich nun das folgende erweiterte Funktormodell:

$$
\begin{aligned}
\mathbf{F}^{m\times n} = \quad & \Big(\{F^{m\times n}\}, \{\mathbf{S_{In}}\}^M, \{\mathbf{S_{Out}}\}^N, \mathbf{Get}, (\{\mathbf{T}_{In}\}^m \times \{\mathbf{T}_{Out}\}^n), I, t_\mathsf{c}, \{\mathsf{M}\}, \dots \Big) \\
\mathbf{Get}: \quad & (\,\{\mathbf{S_{In}}\}^M,\ i:\ -i \in \mathcal{N},\ \mathsf{get}()\,) \longmapsto \{S_{in}\}^m \\
& S_{in_j} \overset{\mathsf{get}()}{:=} \mathbf{S}^{(i)}_{\mathbf{In_{j'}}} \quad \text{mit: } j, j', -i \in \mathcal{N}, 1 \le j \le m, 1 \le j' \le M
\end{aligned}
$$

Die unmittelbar aus dem Datenfluß ableitbaren Beziehungen zwischen Sequenzen und Funktoren können nun wie folgt dargestellt werden. Im einfachsten Fall berechnet sich ein neuer Wert S^0_{Out} einer Ausgangssequenz $\mathbf{S_{Out}}$ mit Hilfe der aktuellen Funktion F^I aus dem aktuellen Wert S^0_{In} genau einer anderen Sequenz $\mathbf{S_{in}}$. Beschrieben wird diese Beziehung durch einen Funktor $\mathbf{F}^{1\times 1}_{\mathbf{X}}$ wie folgt:

$$
\mathbf{S_{Out}} = \mathbf{F_X}(\mathbf{S_{In}}) \iff S^0_{Out} = \mathbf{F_X}(S^0_{In}) \iff V^0_{Out} = F^I(V^0_{In}),\ F^I \in \mathbf{F}^{1\times 1}_{\mathbf{X}}
$$

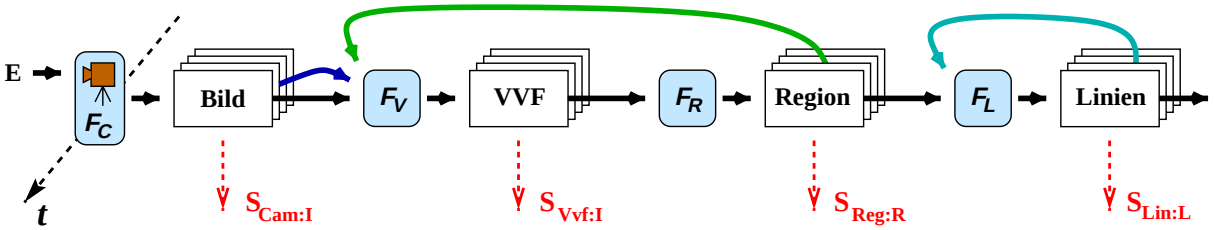
Oft hängt eine Sequenz jedoch nicht von nur *einer* anderen Sequenz und auch nicht nur von den gerade aktuellen, sondern auch von älteren Werten ab. Letzteres wird direkt an der Sequenz

[1] Der Begriff *Funktor* wurde in Anlehnung an den von James O. Coplien verwendeten Namen für Funktionsobjekte *functor* [Cop92] verwendet.

angegeben: z.B. $\mathbf{F}(\mathbf{S}_{\mathbf{In}}^{(0,-1)}) \Leftrightarrow \mathbf{F}(S_{In}^{0}, S_{In}^{-1})$. Darüber hinaus kann ein Funktor mehrere Sequenzen mit neuen Daten versorgen. Allgemein läßt sich der Zusammenhang zwischen Sequenzen und Funktoren somit wie folgt darstellen:

$$
\begin{aligned}
& \{\mathbf{S_{Out_1}}, \ldots, \mathbf{S_{Out_N}}\} = \mathbf{F}(\mathbf{S}_{\mathbf{In_1}}^{(0,-1,\ldots)}, \mathbf{S}_{\mathbf{In_2}}^{(0,-1,\ldots)}, \ldots, \mathbf{S}_{\mathbf{In_M}}^{(0,-1,\ldots)}) \\
\Longleftrightarrow \quad & \{S_{Out_1}^{0}, \ldots, S_{Out_N}^{0}\} = \mathbf{F}(S_{In_1}^{0}, S_{In_1}^{-1}, \ldots,\ S_{In_2}^{0}, S_{In_2}^{-1}, \ldots,\ \ldots,\ S_{In_M}^{0}, S_{In_M}^{-1}, \ldots) \\
& = \{S_{out_1}, \ldots, S_{out_n}\} = \mathbf{F}(S_{in_1}, S_{in_2}, \ldots, S_{in_m}) \\
\Longleftrightarrow \quad & \{V_{Out_1}^{0}, \ldots, V_{Out_N}^{0}\} = F^{I}(V_{In_1}^{0}, V_{In_1}^{-1}, \ldots,\ V_{In_2}^{0}, V_{In_2}^{-1}, \ldots,\ \ldots,\ V_{In_M}^{0}, V_{In_M}^{-1}, \ldots) \\
& = \{V_{out_1}, \ldots, V_{out_n}\} = F^{I}(V_{in_1}, V_{in_2}, \ldots, V_{in_m})
\end{aligned}
$$

Dabei ist zu beachten, daß ein Funktor $\mathbf{F}$ mit M Eingangssequenzen eine Funktion F mit m Eingangsdatenwerten kapselt. Da für jede Ausgangssequenz nur ein aktuelles Datum berechnet wird, ist die Anzahl n der Ausgangsdaten der Funktion gleich der Anzahl N der Ausgangssequenzen des dazugehörigen Funktors.



$\mathbf{S_{Cam}} = \mathbf{F_C}()$ $S_{Cam}^{0} = \mathbf{F_C}()$	$\mathbf{S_{Vvf}} = \mathbf{F_V}(\mathbf{S}_{\mathbf{Cam}}^{(0,-1)}, \mathbf{S}_{\mathbf{Reg}}^{(-1)})$ $S_{Vvf}^{0} = \mathbf{F_V}(S_{Cam}^{0}, S_{Cam}^{-1}, S_{Reg}^{-1})$	$\mathbf{S_{Reg}} = \mathbf{F_R}(\mathbf{S_{Vvf}})$ $S_{Reg}^{0} = \mathbf{F_R}(S_{Vvf}^{0})$	$\mathbf{S_{Lin}} = \mathbf{F_L}(\mathbf{S_{Reg}}, \mathbf{S}_{\mathbf{Lin}}^{(-1)})$ $S_{Lin}^{0} = \mathbf{F_L}(S_{Reg}^{0}, S_{Lin}^{-1})$

Abbildung 4.1: Beziehungen zwischen Sequenzen und Operatoren.

Abb. 4.1 veranschaulicht die Zusammenhänge an einem Beispiel. Wie dort gezeigt wird, erfolgt die Einbindung von Sensoren (hier eine Kamera) in das System ebenfalls mit Hilfe von Funktoren ($\mathbf{F_C}$). Sie sind dadurch gekennzeichnet, daß die Menge der Eingangsdatensequenzen leer ist: $\{\mathbf{S_{In}}\}^{M} = \{V_{in}\}^{m} = \emptyset \Leftrightarrow M = m = 0$. Die so gekapselten Funktionen $F^{0\times n}$ zeichnen sich dadurch aus, daß sie von einer bestimmten Zustandsgröße der Szene und der Zeit abhängen, nicht jedoch von anderen Sequenzdaten V_{in}:

$$
\begin{aligned}
\mathbf{S_{Cam}} = \mathbf{F_C}() : \quad & \forall F \in \mathbf{F_C} : \quad F = F_C(E,t) \quad = F_C(s,t) \\
\mathbf{S_{Dist}} = \mathbf{F_{Sonar}}() : \quad & \forall F \in \mathbf{F_{Sonar}} : \quad F = F_{Sonar}(d,t) = F_{Sonar}(s,t)
\end{aligned}
$$

Dabei läßt sich, wie Abb. 4.2 zeigt, für jeden beliebigen Funktor $\mathbf{F}_{\mathbf{X}}^{m\times n}$ ein äquivalenter Funktor $\mathbf{F}_{\mathbf{X'}}^{0\times n}$ konstruieren, der direkt von den Zustandsgrößen der Szene abhängt. Für potentielle Konsumenten von Daten ist es i.d.R. nicht relevant, mit welcher Funktion und aus welchen Eingangsdaten ein Funktor seine Ausgangsdaten bereitstellt, was die Äquivalenz von Funktoren und (logischen) Sensoren zeigt.

$$
\left.
\begin{aligned}
\mathbf{S_{Edge}} &= \mathbf{F_{Edge}}(\mathbf{S_{Cam}}) \\
\mathbf{S_{Cam}} &= \mathbf{F_C}()
\end{aligned}
\right\} \quad \equiv \quad \mathbf{S_{Edge}} = \mathbf{F_{Edge'}}()
$$

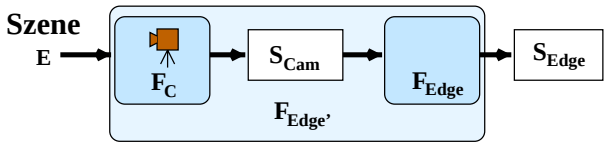


Abbildung 4.2: Äquivalenz zwischen Funktoren und (logischen) Sensoren.

Schließlich kann bei Funktoren auch die Menge der Ausgangsdatensequenzen leer sein: $\{\mathbf{S_{Out}}\}^N = \{\mathbf{S_{out}}\}^n = \emptyset \Leftrightarrow N = n = 0$. Die von diesen Funktoren gekapselten Funktionen $F^{m\times 0}$ transformieren die dynamischen Eingangsdaten nicht in eine andere Datensequenz. Statt dessen erfolgt durch sie z.B. die Aufbereitung der Daten für deren Visualisierung, die Modifikation und Erweiterung von langlebigen Systemdaten, Modellen oder des Systemwissens sowie das Agieren in der Szene mit Hilfe von Aktoren.

4.3.5 Elementare Aufrufmethoden und Aktionen für Funktoren

Die wichtigste Aktion eines Funktors ist das Ausführen der mit ihm assoziierten, aktuell gültigen Funktion F^I. Dafür stellt der Funktor nach außen die $\mathsf{call}()$-Methode bereit.

Bevor nun allerdings $F^I(V_{in_1}, \ldots, V_{in_m})$ aufgerufen werden kann, müssen die m Eingangsdaten $\{V_{in}\}^m$ von den entsprechenden Datensequenzen geholt werden. Für diese sowie weitere Vorbereitungen des Funktionsaufrufs wird die Aktion $\mathsf{pre}()$ definiert. Stehen alle Eingangsdaten zur Verfügung, erfolgt mit Hilfe der Aktion $\mathsf{do}()$ die Abarbeitung der Funktion F^I. Um dabei eine einheitliche und typunabhängige Aufrufschnittstelle zu erhalten, und um auf die mit den Daten assoziierten Zeitinformationen zugreifen zu können, erfolgt die Datenübergabe an die $\mathsf{do}()$-Methode in Form von Sequenzwerten $\mathsf{do}() : \{S_{in}\}^m \mapsto \{S_{out}\}^n$. Auf die eigentlichen Datenwerte $\{V_{in}\}^m$ wird erst innerhalb der Aktion zugegriffen. Die berechneten Daten werden zusammen mit den entsprechenden Zeitinformationen vor dem Verlassen der $\mathsf{do}()$-Methode in Sequenzwerte $\{S_{out}\}^n$ eingetragen. In einem letzten Schritt müssen die neuen Sequenzwerte den entsprechenden Ausgangsdatensequenzen hinzugefügt werden. Diese Aufgabe übernimmt die Aktion $\mathsf{post}()$.

Aufruf des Funktors:

$\mathsf{call}(\mathbf{F_X}, t)$:

$$\{S_{in}\}^m := \mathsf{pre}(\mathbf{F_X}, t);$$
$$\{S_{out}\}^n := \mathsf{do}(\mathbf{F_X}, t, \{S_{in}\}^m);$$
$$\mathsf{post}(\mathbf{F_X}, t, \{S_{out}\}^n);$$

Vorbereitung des Funktionsaufrufs:

$\mathsf{pre}(\mathbf{F_X}, t)$:

$$\forall j, 1 \le j \le m : \quad S_{in_j} = S^i_{In_{j'}} := \mathsf{get}_{[]}(\mathbf{S_{In_{j'}}}, i); \quad \text{oder}$$
$$S_{in_j} = S^i_{In_{j'}} := \mathsf{get}_{\mathsf{t}}(\mathbf{S_{In_{j'}}}, t, i);$$

- Zugriff auf die Eingangsdatensequenzen entsprechend der $\mathbf{Get}$-Relation

Funktionsaufruf:

$\mathsf{do}(\mathbf{F_X}, t, \{S_{in}\}^m)$: $\forall j, 1 \leq j \leq m : \; V_{in_j} := V(S_{in_j})$;

• Zugriff auf die Datenwerte und Typkontrolle $\mathbf{T}(S_{in_j})$

$\{V_{out}\}^n := F(\{V_{in_j}\}^m, \{t_{\mathsf{g}}(V_{in_j})\}^m, \ldots)$;

• Aufruf der Funktion F

$\forall k, 1 \leq k \leq n : \; S_{out_k} := (V_{out_k}, \mathbf{T}_{out_k}, t_{\mathsf{g}}(V_{out_k}), t_{\mathsf{s}} = t_{\mathsf{now}})$;

• Sequenzwerte aus Ergebnissen erzeugen

Aufbereitung der Aufrufergebnisse:

$\mathsf{post}(\mathbf{F_X}, t)$: $\forall k, 1 \leq k \leq n : \mathsf{set}(\mathbf{S_{Out_k}}, t, S_{out_k})$

• Aktualisieren der Ausgangsdatensequenzen

4.4 Agenten

4.4.1 Das Agentenmodell

Agenten zeichnen sich nach [Lüc00] vor allem dadurch aus, daß sie autonom und aktiv sind, Wissen über ihre Domäne besitzen, eine bestimmte Rolle einnehmen, mit anderen Objekten kommunizieren können, und daß ein Agentenzyklus ihre prinzipielle Arbeitsweise bestimmt. Sie besitzen ein großes Maß an Flexibilität und Lernfähigkeit, wodurch sie gut skalierbar und adaptierfähig sind. Dies prädestiniert sie in besonderer Weise für die Bearbeitung inhärent verteilter Aufgaben in komplexen Systemen.

Der Zusammenhang zwischen den die Bildfolgenauswertung beschreibenden Datenflußgraphen und Softwareagenten ergibt sich aus zwei wesentlichen Aufgaben, die Agenten übernehmen können. Gegenstand der ersten Aufgabe ist die Steuerung und Überwachung der zyklisch durchzuführenden Sensordatenbereitstellung und -verarbeitung. Eine zweite Aufgabe beinhaltet die bei Bedarf durchzuführende (Re-) Konfiguration des Datenflußgraphen, um z.B. auf Veränderungen hinsichtlich der zur Verfügung stehenden Ressourcen so zu reagieren, daß deren optimale Ausnutzung garantiert werden kann. Darüber hinaus ist ihr Einsatz auch in anderen Programmkomponenten oder Abstraktionsebenen möglich, beispielsweise für die Steuerung und Überwachung eines Planungsmoduls oder einer Roboteransteuerung sowie für die Parallelisierung von Bildverarbeitungsoperatoren auf der Operatorebene.

Hier soll eine relativ einfache Modellierung von Agenten genügen. So sollen Agenten in der Lage sein, eine Menge von Datensequenzen $\{\mathbf{S_{dep}}\}^m$ zyklisch zu aktualisieren sowie daran anschließend eine Menge von Funktoren $\{\mathbf{F_{dep}}\}^n$ abzuarbeiten. Der Beginn eines Zyklus kann an bestimmte Bedingungen $\mathbf{C} \in \mathcal{B}$ geknüpft werden, beispielsweise das Ende des vorhergehenden Zyklus, den Ablauf eines Timers, um eine Mindestzyklusdauer garantieren zu können, oder das Eintreffen eines externen Triggerimpulses, um die Datenverarbeitung mit dem äußeren Prozeß synchronisieren zu können. Weiterhin wird ein Agent durch seinen aktuellen Arbeitsmodus μ beschrieben — ein Agent kann aktiv, schlafend (bzw. wartend oder angehalten), deaktiviert oder terminiert sein.

Die Abarbeitung der Agentenzyklen erfolgt ausschließlich im aktiven Modus ($\mu = \mathsf{active}$). Ist die Aufrufbedingung nicht erfüllt, kann der Agent allerdings den Modus wechseln und seine Arbeit unterbrechen. Dafür gibt es prinzipiell zwei Möglichkeiten: er kann deaktiviert oder in

den schlafenden Modus versetzt werden. In beiden Fällen bleibt der interne Zustand des Agenten erhalten. Unterschiede ergeben sich bei der Art der Reaktivierung. Ein schlafender Agent (μ = sleeping) kann prinzipiell durch beliebige Signale oder Ereignisse aufgeweckt werden, insofern ein Mechanismus installiert wurde, der dafür sorgt, daß der Agent (z.B. mit Hilfe eines *Event-Handlers*) über das Ereignis informiert wird. Diese Ereignisse können vollkommen unabhängig vom Agenten erzeugt werden und sie müssen auch nicht für ihn bestimmt sein. Sie können beispielsweise von einem Timer, einem externen Signal oder einem Impulsgeber stammen. Im Gegensatz dazu kann ein deaktivierter Agent (μ = deactive) in den aktiven Modus nur durch eine explizit an ihn gerichtete Aktivierungsaufforderung wechseln. Terminierung beendet die Arbeit des Agenten vollständig und löscht den Agenten mit allen Zustandsgrößen und Assoziationen zu anderen Objekten.

Der aktuelle Bearbeitungsstand von Agenten wird schließlich durch Angaben zum aktuellen Zyklus (I, t_c) und den laufenden Aktionen mit dem jeweiligen Bearbeitungsstand $\{\mathsf{M}\}$ modelliert:

$$\mathbf{A} = \Big(\mathbf{C}, \{\mathbf{S_{dep}}\}^m, \{\mathbf{F_{dep}}\}^n, \mu, I, t_c, \{\mathsf{M}\}, \dots \Big) \quad \text{mit}$$
$$\mu \in \{\mathsf{active}, \mathsf{sleeping}, \mathsf{deactive}, \mathsf{terminated}\}$$

Da konkrete, wissensbasierte Planungsvorgänge nicht Gegenstand dieser Arbeit sein sollen, wird an dieser Stelle auf mögliche Erweiterungen der Agenten um Komponenten für die Rekonfiguration des Datenflußgraphen verzichtet. Zu diesen Komponenten zählen beispielsweise Listen der zu verknüpfenden Daten- und Funktorobjekte oder eine Wissensbasis, die die verfügbaren Ressourcen und die anwendbaren Operatoren beschreibt.

4.4.2 Elementare Aktionen von Agenten

Die elementaren Aktionen eines Agenten, die das Arbeitsschema der Agenten bestimmen, sind der Start eines einzelnen neuen Zyklus callCycle() sowie die kontinuierliche Arbeit mit run(). Intern wird zwischen der Aktualisierung der Datensequenzen callSequ() und dem Aufruf der assoziierten Funktoren callFunc() unterschieden. Im kontinuierlichen Modus werden vor dem Beginn eines neuen Zyklus in preCycle() die Aufrufbedingungen $\mathbf{C}$ überprüft und mit $\mathsf{get}_{t_c}()$ die aktuelle Zykluszeit, die z.B. von einem externen Prozeß vorgegeben werden kann, ermittelt. Die Methode postCycle(), die nach Beendigung der Zyklusaktionen aufgerufen wird, erlaubt es schließlich, die Ergebnisse oder das Zeitverhalten der Zyklusaktionen auszuwerten. Dieser Mechanismus kann auf einzelne oder auch alle Sequenz- und Funktoraufrufe ausgedehnt werden, indem diese mit einer Vor- und einer Nachbereitungsroutine assoziiert werden. Dies ermöglicht beispielsweise, zwischen den einzelnen Aktionen das Einhalten von Zeitrestriktionen zu überwachen, das System zu modifizieren oder neu zu konfigurieren, einzelne, weniger wichtige Aktionen auszulassen oder den aktuellen Zyklus abzubrechen.

Bearbeitung eines einzelnen Agentenzyklus:

callCycle($\mathbf{A_X}, t$) : callSequ($\mathbf{A_X}, t$);
callFunc($\mathbf{A_X}, t$);

Kontinuierliche Agentenarbeit starten:

$\mathsf{run}(\mathbf{A_X})$: $\mu := \mathsf{active}$;
`WHILE` $(\mu = \mathsf{active})$:
$t_c := \mathsf{preCycle}(\mathbf{A_X})$;
$\mathsf{callCycle}(\mathbf{A_X}, t_c)$;
$\mathsf{postCycle}(\mathbf{A_X}, t_c)$;

Zyklusvorbereitung:

$\mathsf{preCycle}(\mathbf{A_X})$: `WAITFOR` : $\mathbf{C} = \mathsf{true}$;
return $\mathsf{get}_{t_c}(\mathbf{A_X})$;

Sequenzen aktualisieren:

$\mathsf{callSequ}(\mathbf{A_X}, t)$: $\forall \mathbf{S_{dep_j}} \in \{\mathbf{S_{dep}}\}^m : \mathsf{update}(\mathbf{S_{dep_j}}, t)$

Funktoren aufrufen:

$\mathsf{callFunc}(\mathbf{A_X}, t)$: $\forall \mathbf{F_{dep_k}} \in \{\mathbf{F_{dep}}\}^n : \mathsf{call}(\mathbf{F_{dep_k}}, t)$

Die Methoden $\mathsf{notify}()$ und $\mathsf{setMode}()$ erlauben es, den Agenten aktiv durch den externen Prozeß zu triggern und ihn so an diesen zu koppeln. Beliebige externe Objekte — z.B. Funktoren $\mathbf{F}$ oder Sensordaten $\mathbf{S}$ — können diese Methoden aufrufen und den Agenten über Ereignisse und externe Statusänderungen unterrichten oder aber ihn explizit aktivieren, aufwecken, deaktivieren oder beenden.

Kapitel 5

Zeitaspekte in der Bildfolgenauswertung

Eine bedeutende Rolle bei der Modellierung der zuvor beschriebenen Basiselemente spielen die verschiedenen zeitrelevanten Aspekte des dynamischen Systems. Dabei sind aus Sicht der Sensordatenverarbeitung im einzelnen die folgenden Zeitdaten zu repräsentieren:

- Die Meßzeit, also der Moment t_{g} oder Zeitraum i_{g} der Datenaufnahme stellt ein zentrales Merkmal der Sensordaten dar. Über sie läßt sich das Alter der Daten bestimmen. Weiterhin lassen sich so die aus den Sensordaten extrahierten Informationen zeitlich in den äußeren Prozeß einordnen und zueinander in Beziehung setzen.
- Während die Realzeit kontinuierlich abläuft, erfolgt die Verarbeitung der diskreten Sensordaten in Verarbeitungszyklen mit der Zykluszeit t_{c}. Damit soll der Zeitpunkt bezeichnet werden, zu dem ein Bearbeitungszyklus initiiert wird. Zugriffe auf zusammengehörige Daten erfolgen wie auch die Datenverarbeitung selbst i.d.R. sequentiell, d.h. Datenmessung und -verarbeitung müssen in einer geeigneten Weise synchronisiert werden. Die aus den Daten extrahierten Informationen stehen immer erst mit einer gewissen Verzögerung τ_{age} zur Verfügung.
- Die Rechenzeit der Operatoren τ_{call} ist eine wichtige Größe im Datenverarbeitungsprozeß. Durch sie wird das Mindestalter, mit dem Daten zur Verfügung gestellt werden können, bestimmt. Um vorgegebene Restriktionen hinsichtlich des Datenalters einhalten zu können, ist eine Analyse des Verbrauchs der Ressource Rechenzeit elementar.
- Physikalische und logische Sensoren zeichnen sich häufig durch ein charakteristisches Zeitverhalten aus. So können Sensordaten häufig nur mit einer maximalen Datenrate bereitgestellt werden. Die Daten haben ein gewisses Alter, bis sie im System verfügbar sind. Und bei Zugriffen auf einen Sensor, der nicht kontinuierlich arbeitet oder noch nicht initialisiert wurde, stehen die Daten erst mit einer gewissen Verzögerung zur Verfügung. Ist das Zeitverhalten des Sensors bekannt, kann der Meßvorgang hinsichtlich der Wartezeit auf die Daten und deren Alter beim Zugriff optimiert sowie die Genauigkeit der Messungen erhöht werden.
- Zeitliche Ausdrücke können für die Steuerung von Datenzugriff und -aktualisierung verwendet werden. Dabei soll das System selbständig entscheiden können, ob die vorliegenden Daten zu alt oder noch aktuell genug sind. Neben der Datenmeßzeit und der

Zykluszeit ist hierfür vor allem ein Zeitraum von Bedeutung, der beschreibt, wie lang Daten nach ihrer Aktualisierung gültig sind.

In den folgenden Abschnitten sollen diese Zeitaspekte genauer untersucht und den Anforderungen entsprechend modelliert werden.

5.1 Zeitwerte für die Charakterisierung von Daten

5.1.1 Meß- oder Aufnahmezeit

Neben dem eigentlichen Meßwert stellt die Zeit, zu der eine Datenmessung bzw. -aufnahme erfolgt, ein wichtiges Merkmal der zu bearbeitenden Daten dar. Bei einigen Sensoren, wie z.B. CCD-Kameras erfolgt die Datenerfassung über einen gewissen Zeitraum, d.h. während eines Meßintervalls i_{g} (Belichtungs- oder Shutterzeit). Der eigentliche Meßwert ergibt sich dann — abhängig vom jeweiligen Sensortyp — durch Mittelung, Integration oder Differenzierung der Sensoreingabe über diesen Zeitraum. Andere Sensoren tasten in einem vernachlässigbar kurzem Intervall ihr Eingangssignal ab, bzw. messen mit Hilfe eines kurzen Meßimpulses (z.B. Ultraschallsensoren). Bei ihnen kann von einem Zeitpunkt als Meßzeit t_{g} ausgegangen werden.

Oft ist es sinnvoll, selbst wenn die Messung über einen gewissen Zeitraum erfolgte, die eigentliche Meßzeit auf einen Zeitpunkt zu reduzieren, beispielsweise wenn die Meßdauer vernachlässigbar kurz oder nicht relevant ist, durch die Reduzierung also kein nennenswerter Informationsverlust auftritt. Ist die Aufnahmedauer für alle Messungen konstant, was häufig der Fall ist, kann sie auch gesondert, als Merkmal des Sensors modelliert werden (vgl. Abschnitt 5.3). Dadurch ist sie im System auch dann verfügbar, wenn die Meßzeit auf einen Zeitpunkt reduziert wurde.

Hier sollen sowohl Möglichkeiten für die Modellierung von Meßintervallen als auch von einzelnen Meßzeitpunkten angegeben werden, wobei das Meßintervall die allgemeinere Form darstellt. $t_{\mathsf{g.start}}(S^i_X)$ und $t_{\mathsf{g.end}}(S^i_X)$ bezeichnen den Beginn und das Ende der i-ten Messung der Sequenz $\mathbf{S_X}$. Das in Abschnitt 4.2.1 angegebene Modell für Sequenzwerte bleibt davon unberührt, da mit Hilfe eines Meßzeitpunktes t_{g} und der Meßdauer τ_{g} auch Meßintervalle angegeben werden können. Als Meßzeitpunkt soll hier das Ende der Messung betrachtet werden: $t_{\mathsf{g}} = t_{\mathsf{g.end}}$. Dieser Zeitpunkt stellt darüber hinaus den Bezugspunkt für die Bestimmung des Alters einer Messung — und damit der gemessenen Daten — dar.

Meßintervall:	$i^i_{\mathsf{g,X}}$	$= i_{\mathsf{g}}(S^i_X)$	$= [i^-_{\mathsf{g,X^i}}, i^+_{\mathsf{g,X^i}}]$	$= [t_{\mathsf{g.start}}(S^i_X), t_{\mathsf{g.end}}(S^i_X)]$
Meßzeitpunkt:	$t^i_{\mathsf{g,X}}$	$= t_{\mathsf{g}}(S^i_X)$	$= i^+_{\mathsf{g,X^i}}$	$= t_{\mathsf{g.end}}(S^i_X)$
Meßdauer:	$\tau^i_{\mathsf{g,X}}$	$= \tau_{\mathsf{g}}(S^i_X)$	$= \tau_{\mathsf{d}}(i^i_{\mathsf{g,X}})$	$= t_{\mathsf{g.end}}(S^i_X) - t_{\mathsf{g.start}}(S^i_X)$
Alter der Messung:	$\tau^i_{\mathsf{ago,g,X}}$	$= \tau_{\mathsf{ago}}(t^i_{\mathsf{g,X}})$	$= \tau_{\mathsf{ago}}(i^i_{\mathsf{g,X}})$	$= t_{\mathsf{now}} - t_{\mathsf{g.end}}(S^i_X)$

Ist im System nur ein unabhängiger Sensor vorhanden, oder wird aus dem Zusammenhang klar, auf welche Sequenz die Angaben sich beziehen, kann der Index X bei der Angabe der Meßzeit auch entfallen.

Nicht zu verwechseln mit der Datenmeßzeit i_{g} bzw. t_{g} ist die Datenbereitstellungzeit t_{s}. Erst ab diesem Zeitpunkt stehen die Daten im System tatsächlich zur Verfügung. Diese Zeit hängt

von den Rechenzeiten der verwendeten Operatoren, den zur Verfügung stehenden Ressourcen und der Funktionsweise der Sensoren ab.

Datenbereitstellung: $t^i_{\mathsf{s},\mathsf{X}} = t_{\mathsf{s}}(S^i_X) \quad (\text{mit } t_{\mathsf{s}}(S^i_X) > t_{\mathsf{g}}(S^i_X))$
Datenverzögerung: $\tau^i_{\mathsf{s},\mathsf{X}} = t^i_{\mathsf{s},\mathsf{X}} - t^i_{\mathsf{g},\mathsf{X}}$

5.1.2 Propagierung von Meßzeiten

Die Datenmeßzeit ist nicht nur für die Daten, die unmittelbar von einem Sensor geliefert werden, charakteristisch, sondern auch für alle aus diesen Daten abgeleiteten Merkmale, unabhängig davon, wann diese berechnet werden oder auf sie zugegriffen wird. Ignoriert man beispielsweise bei der kamerabasierten Lokalisation schnell bewegter Objekte den Zeitversatz zwischen Bildaufnahme und Auswertung der extrahierten Merkmale, kann dies die Meßergebnisse deutlich verfälschen, insbesondere dann, wenn die Kamera sich in dieser Zeitspanne selber auch bewegt hat. Komponenten, die die Positionen anderer Objekte in der Szene auswerten, müssen daher berücksichtigen, daß einzelne Merkmale oder die beobachteten Objekte sich nicht unbedingt zum Zeitpunkt des Datenzugriffs an der berechneten Position befinden, sondern im Moment der Datenmessung durch den physikalischen Sensor. Dieses Problem tritt auch beim Ermitteln der Position des Sensors selbst auf, z.B. wenn diese mit Hilfe der Roboterodometrie bestimmt wird. Neben dem Alter der Daten ist zu berücksichtigen, daß die Odometriedaten im allgemeinen nicht mit den Bilddaten zeitlich zusammenfallen müssen.

Daher ist es eine wichtige Aufgabe der mit der Bereitstellung der Daten beauftragten Funktoren, ihre Ausgangsdaten mit einem korrekten Zeitstempel zu versehen. Dieser hängt vom Zeitverhalten der zugrundeliegenden physikalischen Sensoren bzw. den zu verknüpfenden Eingangsdaten ab. Funktoren, die physikalische Sensoren in das System einbinden, müssen, wenn diese nicht von sich aus einen Zeitstempel liefern, deren Zeitverhalten kennen und modellieren. Andere Funktoren müssen die Zeitstempel der Ausgangsdaten aus denen der Eingangsdaten generieren.

Berechnet sich ein Merkmal lediglich aus dem aktuellen Wert genau einer anderen Sequenz, ist diese Aufgabe trivial. Der Zeitstempel des Eingangsdatums überträgt sich direkt auf das Ausgangsdatum, d.h. die Meßzeit kann auf neu berechnete Merkmale einfach propagiert werden. Fließen mehrere Sequenzwerte, die alle den gleichen Zeitstempel besitzen, in die Berechnung eines neuen Datums ein, ist die Zeitpropagierung genauso einfach:

$$\begin{aligned} S_{Out} &= \mathbf{F}(S^0_{In}) && \Longrightarrow \quad i_{\mathsf{g},\mathsf{Out}} := i_{\mathsf{g},\mathsf{In}} \\ \{S_{Out}\}^N &= \mathbf{F}(S^0_{In_1}, \ldots, S^0_{In_M}) \wedge i_{\mathsf{g},\mathsf{In}_1} = \cdots = i_{\mathsf{g},\mathsf{In}_\mathsf{M}} = i_{\mathsf{g},\mathsf{In}} && \Longrightarrow \quad i_{\mathsf{g},\mathsf{Out}_\mathsf{k}} := i_{\mathsf{g},\mathsf{In}} \\ & && \qquad \text{für } \forall k,\ 1 \leq k \leq N \end{aligned}$$

Werden die Ausgangsdaten eines Funktors allerdings aus verschiedenen, unterschiedlich alten Daten bzw. aus einer Datensequenz, die über einen gewissen Zeitraum beobachtet wurde, berechnet, wirkt sich dies i.d.R. auch auf die Zeitstempel der zu berechnenden Daten aus. Hierbei hängt die Methode, mit der die Propagierung der Zeitstempel erfolgen soll, von der Semantik der Daten und den Operatoren, die diese verknüpfen, ab. Wird z.B. aus zwei aufeinanderfolgenden Positionsmerkmalen die Geschwindigkeit eines Objekts bestimmt, basiert die Geschwindigkeit auf einem Meßintervall, das beide Positionsbestimmungen einschließt. Das kann bei der Interpretation eine wichtige Information darstellen, etwa wenn es um die Frage der

Gültigkeit der Geschwindigkeitswerte geht. In anderen Fällen spielt dagegen die Zeit einzelner Daten keine Rolle, so daß diese auch nicht propagiert werden darf.

Allgemein kann eine differenzierte Propagierung der Meßzeiten mit Hilfe einer Funktion f_{i_g}, bzw. bei mehreren Ausgangsdaten durch eine Menge von Funktionen $\{f_{i_\mathsf{g}}\}^N$ beschrieben werden. Diese können unter Berücksichtigung von speziellem Domänenwissen und der Funktionsweise der verwendeten Operatoren, den Einfluß der einzelnen Eingangsdaten auf das Ergebnis richtig beurteilen und die Eingangszeitstempel in der gewünschten Form verknüpfen.

$$i_{\mathsf{g,Out}} = f_{i_\mathsf{g}}(i_\mathsf{g}(S_{in_1}), \ldots, i_\mathsf{g}(S_{in_m}))$$

In vielen Fällen ist eine solch detaillierte Beschreibung jedoch nicht notwendig. Zugriffe auf ältere Datenwerte folgen i.d.R. bestimmten Anwendungsmustern, die sich verallgemeinern lassen und für die eine Standardvorgehensweise im Umgang mit Zeitstempeln definiert werden kann. In einer ersten Propagierungsmethode bestimmt der von den Zeitstempeln aller Eingangsdaten überspannte Zeitraum die Meßzeit der Ausgangsdaten:

$$\{S_{Out}\}^N = \mathbf{F}(S_{in_1}, \ldots, S_{in_m}) \quad \Longrightarrow \quad i_{\mathsf{g,Out}} = i_\mathsf{g}(S_{Out}) = i_{\mathsf{span}}(i_\mathsf{g}(S_{in_1}), \ldots, i_\mathsf{g}(S_{in_m}))$$
$$\text{mit :} \quad i_{\mathsf{span}}(i_1, i_2, \ldots, i_m) = i_{\mathsf{span}}(\ldots i_{\mathsf{span}}(i_{\mathsf{span}}(i_1, i_2), i_3), \ldots i_m)$$

Als typisches Anwendungsmuster für dieses Propagierungsverfahrens kommen differenzierende und mittelnde Operatoren in Frage. In diese Klasse gehören Operatoren, mit deren Hilfe z.B. in aufeinanderfolgenden Bildern Änderungen oder aus einer Folgen von Positionswerten Geschwindigkeiten bestimmt werden. Sie können aber auch für eine Reduzierung des Rauschens die letzten n Bilder mitteln. Die Meßzeit für die so berechneten Daten entspricht dem Gesamtzeitraum über den Eingangsmessungen, was vernünftig ist, da das Ergebnis auf dem Verlauf von Szenenzuständen über diesen Zeitraum basiert. Gekennzeichnet wird diese Operatorklasse durch mehrere unterschiedlich alte Datenwerte aus ein und derselben Eingangssequenz $\mathbf{S}_{\mathbf{In}}^{(0,-1,\ldots)}$:

$$S_{Out} = \mathbf{F}(S_{In}^0, S_{In}^{-1}) \quad \Longrightarrow \quad i_{\mathsf{g,Out}} = i_\mathsf{g}(S_{Out}) = i_{\mathsf{span}}(i_\mathsf{g}(S_{In}^0), i_\mathsf{g}(S_{In}^{-1})) = [i^-_{In^{-1}}, i^+_{In^0}]$$

Als zweite Operatorklasse mit dieser Propagierungsmethode sollen integrierende Operatoren betrachtet werden. Mit ihrer Hilfe werden fortlaufend die Daten einer Sequenz aufaddiert, z.B. um Objekte über eine gewisse Zeit zu zählen, oder um den Systemzustand über einen längeren Zeitraum zu mitteln. Auch hier ist es naheliegend, als Meßdauer die Gesamtzeit seit Beginn der Integration zu betrachten. Für diese Operatoren ist charakteristisch, daß ein Eingangsdatum ein zuvor bestimmter Wert einer der Ausgangssequenzen des Operators ist $\mathbf{S}_{\mathbf{In_j}} = \mathbf{S}_{\mathbf{Out_k}}$:

$$S_{Out} = \mathbf{F}(S_{In}^0, S_{Out}^{-1}) \quad \Longrightarrow \quad i_{\mathsf{g,Out}} = i_\mathsf{g}(S_{Out}) = i_{\mathsf{span}}(i_\mathsf{g}(S_{In}^0), i_\mathsf{g}(S_{Out}^{-1})) = [i^-_{Out^{-1}}, i^+_{In^0}]$$
$$\text{Initialisierung:} \quad i^1_{\mathsf{g,Out}} = [i^-_{In^1}, i^+_{In^1}]$$
$$\text{nach Rekursion:} \quad i^i_{\mathsf{g,Out}} = [i^-_{In^1}, i^+_{In^i}]$$

Im Gegensatz dazu gibt es eine Reihe von Anwendungsfällen, in denen eine Propagierung der Eingangszeiten auf die Ausgangsdaten nicht sinnvoll ist. Das betrifft z.B. Operatoren, die die von integrierenden Operatoren erzeugten Daten für das Bestimmen aktueller Merkmale weiter verwenden. So fließen in einen Kalman-Filter immer auch die alten Werte als Prädiktion in die aktuelle Berechnung ein — durch die Rekursion bei der Zeitpropagierung würden also alle so bestimmten Daten immer einen Meßzeitraum vom Beginn der ersten bis zur aktuellen

Messung umfassen. Auch wenn eine solche Interpretation theoretisch begründbar wäre, aus praktischer Sicht ist sie nicht sinnvoll.

Manche Eingangsdaten stellen lediglich Hilfsgrößen für die Bearbeitung der zu analysierenden Daten dar. Dabei kann es sich z.B. um Suchbereiche, prädizierte Merkmale oder Mittelungen vergangener Systemzustände als Vergleichsbasis handeln. Diese Hilfsgrößen werden zwar für die Berechnung der neuen Sequenzwerte herangezogen — beispielsweise um zu bestimmen, wo im Bild nach Merkmalen zu suchen ist —, wann und auf welcher Grundlage diese Daten ermittelt wurden, ist an dieser Stelle aber nicht mehr relevant. Die Meßzeiten von Hilfsgrößen werden daher ignoriert, und für diese Anwendungsfälle wird standardmäßig ein alternativer Propagierungsmechanismus verwendet. Bei diesem legt nur der erste Eingabeparameter den Zeitstempel für die Ausgabedaten fest:

$$S_{Out} = \mathbf{F}(S_{In}; S_{H_1}, \ldots, S_{H_{m-1}}) \quad \Longrightarrow \quad i_{\mathsf{g,Out}} = i_{\mathsf{g}}(S_{Out}) = i_{\mathsf{g}}(S_{In})$$

Mit diesen beiden Propagierungsverfahren läßt sich ein Großteil aller Anwendungsfälle abdecken. Welches Verfahren eingesetzt werden muß, kann anhand der Eingangsdatensequenzen automatisch entschieden werden. Nicht abgedeckt wird hierbei allerdings, wenn ein Operator die nacheinander gemessenen Daten mehrerer unabhängiger Sensoren direkt weiterverarbeitet. In diesem Fall kann jedoch die Auswertung der Eingangsmeßzeiten durch eine Funktion $f_{i_{\mathsf{g}}}$ vorgegeben werden. Die Ansätze dieses Abschnittes zusammenfassend ergibt sich damit für die Bestimmung der Ausgangszeitstempel eines Funktors $\mathbf{F_X} : \{S_{out}\}^n = \mathbf{F_X}(\{S_{in}\}^m)$ der folgende, in der post()-Methode des Funktors zu integrierende Algorithmus:

$$\begin{array}{ll}
\mathsf{post}(\mathbf{F_X}, t): & \texttt{IF } (\exists i_{\mathsf{g,out}} = i_{\mathsf{g}}(S_{out_k}), 1 \le k \le n : i_{\mathsf{g,out}} = \mathsf{undefined}) : \\
& \quad i_{\mathsf{g,out}} := i_{\mathsf{g}}(S_{in_1}); \\
& \quad \forall S_{in_j}, 2 \le j \le m : \\
& \quad\quad \texttt{IF } (\mathbf{S_{In_j}} = \mathbf{S_{In_1}} \vee \mathbf{S_{In_j}} \in \{\mathbf{S_{Out}}\}^n) : \\
& \quad\quad\quad i_{\mathsf{g,out}} := i_{\mathsf{span}}(i_{\mathsf{g}}(S_{in_j}), i_{\mathsf{g,out}}); \\
& \quad \forall S_{out_k}, 1 \le k \le n : \\
& \quad\quad \texttt{IF } (i_{\mathsf{g}}(S_{out_k}) = \mathsf{undefined}) : \quad i_{\mathsf{g}}(S_{out_k}) := i_{\mathsf{g,out}}; \\
& \forall S_{out_k}, 1 \le k \le n : \quad \mathsf{set}(\mathbf{S_{Out_k}}, t, S_{out_k})
\end{array}$$

5.1.3 Datenzyklen

Die Aufnahme und die Bearbeitung der kontinuierlich zu erfassenden dynamischen Daten erfolgt in Zyklen, wobei in vielen Arbeiten von einer konstanten Zykluszeit ausgegangen wird, die von der Datenrate der Sensoren bestimmt wird. So wird die Echtzeit-Bildfolgenverarbeitung in der Literatur häufig mit Bildverarbeitung in Videorate gleichgesetzt und die Berechnung direkt an den Datenzyklus gekoppelt:

$$\begin{array}{lll}
\text{konstanter Datenzyklus:} & \tau_{\mathsf{cg}} = t^i_{\mathsf{g}} - t^{i-1}_{\mathsf{g}} & \\
\text{und Arbeitszyklus:} & = \tau_{\mathsf{c}} = const. & \\
\text{Videorate:} & \tau_{\mathsf{cg,PAL}} = 40\ \mathsf{ms} & (PAL - Norm) \\
& \tau_{\mathsf{cg,NTSC}} = 33.33\ \mathsf{ms} & (NTSC - Norm)
\end{array}$$

Für viele Anwendungen ist diese Forderung jedoch zu strikt und die explizite Kopplung zwischen Datenaufnahme und Datenverarbeitung unnötig oder sogar hinderlich. So ist häufig

weder eine konstante Zykluszeit noch das Einhalten der Videorate für das Erfüllen der gestellten Aufgabe zwingend notwendig. In Abhängigkeit von den konkreten Daten und ihrem Informationsgehalt kann die Auswertung mehr oder weniger aufwendig sein, zusätzliche Aufgaben mit höherer Priorität können die Sensordatenauswertung bremsen, und schließlich können verschiedene Teilaufgaben unterschiedliche Anforderungen an die Genauigkeit oder Sicherheit der Daten stellen.

Geht man aber von einer variablen Zyklusdauer aus, wird sie zu einer Eigenschaft des konkreten Zyklus, die erst nach dessen Beendigung, also wenn der nächste Zyklus beginnt, feststeht. Die Zyklusdauer ist so vor allem von statistischem Wert oder kann z.B. im Nachhinein für die Überprüfung des Einhaltens vorgegebener Restriktionen verwendet werden.

Um weitestgehend unabhängige Komponenten für die Datenbereitstellung und -auswertung modellieren zu können, soll im folgenden zwischen *Datenzyklen* und *Arbeitszyklen* unterschieden werden. Verschiedene Daten können unterschiedliche Datenzyklen besitzen, und die Auswertung und Weiterverarbeitung der Daten kann in davon unabhängigen Arbeitszyklen erfolgen. Die Datenzyklen definieren sich über die Aufnahmezeiten, die Arbeitszyklen über den Beginn der jeweiligen Iteration zur Datenauswertung. Auch wenn die Arbeitszyklen eines Moduls $\mathcal{M}_{\mathsf{M}}$ in erster Linie die Programmkomponenten und nicht so sehr die Szenendaten charakterisieren, soll deren Definition hier vorweggenommen werden (vgl. Abschnitt 5.2).

Datenzyklus:	$i^i_{\mathsf{cg,X}}$	$= i_{\mathsf{cg}}(S^i_X)$	$= [t^i_{\mathsf{g,X}}, t^{i+1}_{\mathsf{g,X}}]$	$= [i^+_{\mathsf{g,X}^i}, i^+_{\mathsf{g,X}^{i+1}}]$
Zyklusdauer:	$\tau^i_{\mathsf{cg,X}}$	$= \tau_{\mathsf{cg}}(S^i_X)$	$= \tau_d(\, i^i_{\mathsf{cg,X}}\,)$	$= t^{i+1}_{\mathsf{g,X}} - t^i_{\mathsf{g,X}}$
Meßpause:	$\tau^i_{\mathsf{cg'},\mathsf{X}}$	$= \tau_{\mathsf{cg'}}(S^i_X)$	$= \tau^i_{\mathsf{cg,X}} - \tau^{i+1}_{\mathsf{g,X}}$	$= i^-_{\mathsf{g,X}^{i+1}} - i^+_{\mathsf{g,X}^i}$
mittlere Zyklusdauer:	$\bar{\tau}^i_{\mathsf{cg,X}}$	$= \bar{\tau}_{\mathsf{cg}}(S^i_X)$	$= \frac{1}{i}\sum_{l=1}^{i} \tau^l_{\mathsf{cg,X}}$	$= \frac{1}{i}(t^{i+1}_{\mathsf{g,X}} - t^1_{\mathsf{g,X}})$
Arbeitszyklus:	$i^i_{\mathsf{c,M}}$	$= i_{\mathsf{c}}(\mathcal{M}^i_{\mathsf{M}})$	$= [t^i_{\mathsf{c.start,M}}, t^{i+1}_{\mathsf{c.start,M}}]$	
Zykluszeit:	$t^i_{\mathsf{c,M}}$	$= t_{\mathsf{c}}(\mathcal{M}^i_{\mathsf{M}})$		

Für einen abgeschlossenen Zyklus gibt es verschiedene Möglichkeiten, Sequenzwerte für Zeitpunkte, die zwischen den beiden Messungen liegen und diskretisierungsbedingt fehlen, zu bestimmen. Sie können zurück auf den zum Anfragezeitpunkt aktuellen Meßwert, auf den nächstliegenden Zeitpunkt oder auf einen aus den bekannten Werten interpolierten Wert gesetzt werden. Dabei haben alle drei Methoden aufgrund der speziellen Vor- und Nachteile ihre Berechtigung:

Vorhergehender Wert: $\mathbf{S}(t) := \mathbf{S}(t^{i-1}_{\mathsf{g}}) = S^{i-1}$ für $i^-_{\mathsf{g,i-1}} \le t < i^-_{\mathsf{g,i}}$

Vorteile: Es werden für eine Anfragezeit immer die gleichen Daten geliefert, unabhängig davon, ob der Zyklus noch offen, d.h. aktuell ist oder ob er durch eine zwischenzeitliche Aktualisierung abgeschlossen wurde.

Nachteile: Von allen Methoden gibt es hier die größten Abweichungen zwischen der Abfrage- und der Datenmeßzeit, wenn die Abfragezeit kurz vor der Messung des folgenden Meßwertes liegt.

Nächster Wert:
$$\mathbf{S}(t) := \begin{cases} \mathbf{S}(t^{i-1}_{\mathsf{g}}) = S^{i-1} & \text{für} \quad i^-_{\mathsf{g,i-1}} \le t < i^+_{\mathsf{g,i-1}} + \tau^{i-1}_{\mathsf{cg'}}/2 \\ \mathbf{S}(t^i_{\mathsf{g}}) \quad = S^i & \text{für} \quad i^+_{\mathsf{g,i-1}} + \tau^{i-1}_{\mathsf{cg'}}/2 \le t < i^-_{\mathsf{g,i}} \end{cases}$$

Vorteile: Kleinere Abweichung zwischen Abfragezeit und Datenzeit, wenn die Abfragezeit kurz vor der Datenzeit des folgenden Meßwertes liegt.

Nachteile: Unter Umständen unterschiedliche Werte bei Datenzugriffen vor und nach der Aktualisierung einer Datenfolge, die von einer anderen Komponente verursacht wurde.

Interpolierter Wert: $\mathbf{S}(t) := \mathbf{F_{int}}(\mathbf{S}(t_g^i), \mathbf{S}(t_g^{i-1}), \ldots)$ für $i^+_{g,i-1} < t < i^-_{g,i}$

Vorteile: Geringster Fehler bei Abweichungen zwischen Daten- und Anfragezeit.

Nachteile: Unterschiedliche Werte vor und nach einer Aktualisierung der Datenfolge; höherer Rechenaufwand; nicht für alle Datentypen ist eine Interpolation sinnvoll.

5.1.4 Gültigkeit von Sequenzwerten

Eine weitere wichtige Zeitgröße zur Beschreibung der Sequenzwerte — neben Aufnahmezeit und Zykluszeit — ist der Gültigkeitszeitraum. Das ist die Zeitspanne, während der die gemessenen oder berechneten Daten nach ihrer Bereitstellung bzw. nach der Datenaufnahme gültig sind und verwendet werden können, ohne daß auf einen neuen Wert gewartet werden muß. Während die Meßzeit die Problemdomäne, d.h. die Szene charakterisiert, stellt diese Gültigkeitszeit den Bezug zwischen Szene und Datenverarbeitung her.

In Systemen, in denen mehrere Komponenten unabhängig voneinander die gleichen dynamischen Daten verwenden, ist bei jedem Datenzugriff zu entscheiden, ob durch die Datenquelle — egal ob logischer oder physikalischer Sensor — ein neuer Wert bereitgestellt werden soll, oder ob der letzte Wert noch aktuell (genug) ist. Für Daten, die durch andere Systeme, die unabhängig von den zugreifenden Modulen arbeiten, kontinuierlich bereitgestellt werden, ist eine solche Überlegung überflüssig. Dies gilt jedoch nicht für die im gleichen System aufwendig berechneten Daten oder für Sensoren, die ihre Daten erst nach einer gewissen Verzögerung oder nur zyklisch, verbunden mit einer entsprechenden Wartezeit bereitstellen können.

Bei unabhängigen Verbrauchern ist es möglich, daß zwei Datenanfragen unmittelbar aufeinander folgen. In einem solchen Fall würden bei der erneuten Berechnung der Daten zum einen wichtige Systemressourcen unnötig belegt werden. Gibt es für den Sensor noch keine neuen Eingangsdaten, würde der Sensor auf die nächsten Daten warten und damit den Verbraucher blockieren oder es könnten — insbesondere bei logischen Sensoren — die Berechnungen mit den gleichen Sensordaten wiederholt werden, was beides aufgrund der vorhandenen, noch relativ neuen Daten unnötig wäre. Andererseits kann auf diesem Weg erkannt werden, ob die letzten Daten bereits veraltet sind, und der Sensor einen neuen Wert bereitstellen muß.

Gültigkeitsdauer: $\tau_{v,X} = ? \quad \rightarrow \quad$ z.B. $\begin{cases} \tau_{cg,min} : & \text{bekannter Sensorzyklus} \\ \tau_{c,min} : & \text{ermittelte Modulrechenzeit} \end{cases}$

Gültigkeitsintervall: $i^i_{v,X} = i_v(S^i_X) = [i^+_{v,X^{i-1}}, t^i_{g,X} + \tau_{v,X}]$

$i^1_{v,X} = [i^-_{g,X^1}, t^1_{g,X} + \tau_{v,X}]$ (Initialisierung)

Sequenzzugriff: $\mathbf{S}(t) = ? \quad \rightarrow \begin{cases} \mathbf{S}(t) := S^0 & \text{für} \quad t \in i^0_v : \quad \text{letzter Wert} \\ \mathbf{S}(t) := S^{+1} & \text{für} \quad t > i^0_v : \quad \text{Neuberechnung} \end{cases}$

Festlegen läßt sich der Gültigskeitszeitraum für ein Datum zum einen in Abhängigkeit von der maximalen Datenrate der Sensoren: die Daten müssen beispielsweise so lange gültig sein,

wie ein Sensor braucht, um neue Daten bereitzustellen. Sie kann aber auch über die kleinste erreichbare Zyklusdauer der Verarbeitungsmodule definiert werden: Die Daten brauchen nicht häufiger bereitgestellt werden, als andere Komponenten sie auswerten können. Die Gültigkeitsdauer kann dann im laufenden System kontinuierlich über eine bestimmte Anzahl von Zyklen ermittelt und dynamisch angepaßt werden.

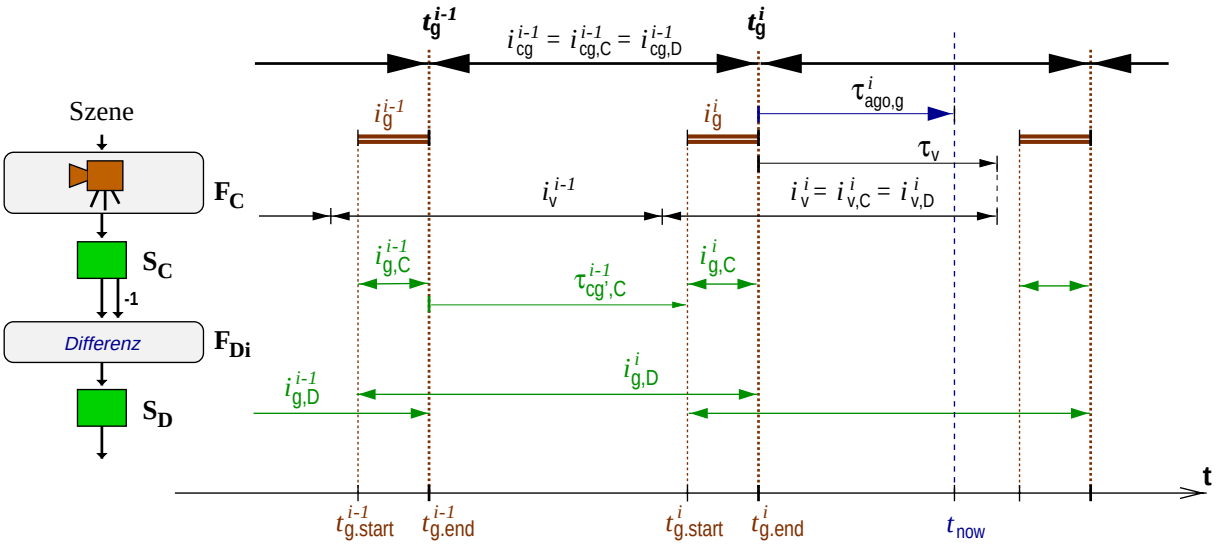


Abbildung 5.1: Charakteristische Zeitwerte dynamischer Datensequenzen.

Abb. 5.1 faßt die wichtigsten Zeiten zur Beschreibung von dynamischen Daten zusammen und stellt sie graphisch dar. Zu beachten ist dabei, daß die dargestellten Daten „Kamerabild“ und „Differenzbild“ den gleichen Datenzyklus besitzen. Die Differenzbilder berechnen sich aus zwei aufeinanderfolgenden Kamerabildern, weswegen ihr Meßintervall den gesamten Aufnahmezeitraum von zwei Kamerabildern umfaßt.

5.2 Zeitverhalten von Funktoren

In diesem Abschnitt wird das Zeitverhalten der Funktoren genauer untersucht und beschrieben. Im Gegensatz zu der im vorhergehenden Abschnitt durchgeführten Charakterisierung der Szene steht nun das Computersystem und der Datenverarbeitungsprozeß im Mittelpunkt der Betrachtungen.

Die Analyse der Laufzeit der einzelnen Datenverarbeitungsprozesse stellt vor allem während der Programmentwicklung, aber auch im laufenden Betrieb eine wichtige Grundlage für die Optimierung des Programmsystems dar. In der Regel gibt es zeitliche Schranken und Restriktionen, die in einem Echtzeitsystem bei der Bearbeitung der Daten eingehalten werden müssen, um die gestellte Aufgabe erfüllen oder auf Zustandsänderungen der Szene schnell genug reagieren zu können. Eine detaillierte Beschreibung des Zeitverhaltens der einzelnen Operatoren hilft dabei, geeignete auszuwählen und diese so zu parametrisieren, daß ein Kompromiß zwischen erzielbarer Genauigkeit und benötigter Rechenzeit gefunden wird. Programmteile mit

sehr hohem Ressourcenbedarf lassen sich durch eine Laufzeitanalyse auffinden und ggf. durch andere Operatoren ersetzen.

Darüber hinaus müssen die von der Software durchzuführenden Planungsaufgaben nicht allein auf Aktionen in der realen Umgebung beschränkt sein. Auch die Auswahl von Algorithmen, Operatoren, Parametern und die Vergabe von Ressourcen stellt ein interessantes Planungsziel dar. Softwareagenten, die einzelne Programmteile planen, organisieren und konfigurieren, können beispielsweise dazu beitragen, das Einhalten von Restriktionen zu überprüfen, die Gesamtsystemperformanz zu verbessern und die vorhandenen Ressourcen optimal auszunutzen. Das Zeitverhalten des Programms mit seinen Operatoren stellt dafür — im Zusammenhang mit einer Wissensbasis, in der die Operatoren und ihr generelles Zeitverhalten beschrieben sind — eine zentrale Planungsgrundlage dar.

5.2.1 Arbeitszyklen in zyklisch arbeitenden Modulen

Die einzelnen Funktoren sind i.d.R. Bestandteil von Programmkomponenten, in denen sie zyklisch aufgerufen werden. Eine solche Programmkomponente stellt ein zyklisch arbeitendes Modul ($\mathcal{M}$) dar. Der i-te Zyklus des Moduls ist $\mathcal{M}^i$, wobei der aktuelle Zyklus alternativ mit $\mathcal{M}^0$ bezeichnet wird. Im wesentlichen besteht das Modul aus Mengen von Eingangs-, Zwischen- und Ausgangsdatensequenzen $\{\mathbf{S_{In}}\}^m$, $\{\mathbf{S_M}\}^l$ und $\{\mathbf{S_{Out}}\}^n$ sowie Funktoren $\{\mathbf{F_M}\}^k$, die die für die Datenverarbeitung notwendigen Operationen kapseln. Ein Agent $\mathbf{A_M}$ kann das Modul steuern, in dem es entsprechend Kapitel 4.4 zyklisch bestimmte Sequenzen aktualisiert und Funktoren aufruft. Nach außen stellt sich ein solches Modul als *ein* logischer Sensor dar.

Die Gesamtzykluszeit derartiger Module wurde bereits in Abschnitt 5.1.3 eingeführt und mit i_{c} bezeichnet. Charakterisiert wird das Zeitverhalten eines Zyklus aber nicht nur durch Beginn- und Endzeitpunkt. Zu den interessierenden Zeiten gehört auch, wann, d.h. mit welchen Verzögerungen τ_{s} gegenüber der Meßzeit, die einzelnen Daten bereitgestellt werden und in welchen Zeiträumen die verschiedenen Operatoren arbeiten. Darüber hinaus können für einen Arbeitszyklus Restriktionen hinsichtlich seiner minimalen, maximalen oder optimalen Zykluslänge definiert werden.

Arbeitszyklus:	
Start:	$t^i_{\mathsf{c.start,M}} = t_{\mathsf{c.start}}(\mathcal{M}^i_{\mathsf{M}})$
Ende:	$t^i_{\mathsf{c.end,M}} = t_{\mathsf{c.end}}(\mathcal{M}^i_{\mathsf{M}}) = t^{i+1}_{\mathsf{c.start,M}}$
Zyklus:	$i^i_{\mathsf{c,M}} = i_{\mathsf{c}}(\mathcal{M}^i_{\mathsf{M}}) = [t^i_{\mathsf{c.start,M}}, t^i_{\mathsf{c.end,M}}] = [i^-_{\mathsf{c,M^i}}, i^+_{\mathsf{c,M^i}}]$
Dauer:	$\tau^i_{\mathsf{c,M}} = \tau_d(i^i_{\mathsf{c,M}}) = t^{i+1}_{\mathsf{c.start,M}} - t^i_{\mathsf{c.start,M}}$
Zyklusrestriktionen:	
Minimallänge:	$\tau^{\mathsf{min!}}_{\mathsf{c,M}}$
Maximallänge:	$\tau^{\mathsf{max!}}_{\mathsf{c,M}}$
Optimale Dauer:	$\tau^{\mathsf{opt!}}_{\mathsf{c,M}}$

5.2.2 Funktorzeiten

Für eine detailliertere Beschreibung der Arbeitsweise von Funktoren wurde deren Aufruf $\mathsf{call}()$ in Abschnitt 4.3.5 in drei Phasen unterteilt: die Aufrufvorbereitung $\mathsf{pre}()$, in der die dynami-

schen Eingangsdaten $\{S_{in}\}^m$ angefordert werden, die eigentliche Berechnungsphase $\mathsf{do}()$ und die Nachbereitung $\mathsf{post}()$, in der die softwaretechnische Aufbereitung und Bereitstellung der berechneten Ausgangsdaten $\{S_{out}\}^n$ erfolgt. Diese Aufteilung spiegelt sich auch bei der Analyse des Zeitverhaltens der Funktoren wider:

Charakteristische Funktorzeiten:

Aufrufzeitpunkt	/ Vorbereitungsbeginn:	$t^i_{\mathsf{call,X}} = t^i_{\mathsf{pre,X}} = t_{\mathsf{pre}}(\mathbf{F}^i_{\mathbf{X}})$
Berechnungsbeginn	/ Vorbereitungsende:	$t^i_{\mathsf{do,X}} = t^i_{\mathsf{pre.end,X}} = t_{\mathsf{do}}(\mathbf{F}^i_{\mathbf{X}})$
Datenaufbereitungsbeginn	/ Berechnungsende:	$t^i_{\mathsf{post,X}} = t^i_{\mathsf{do.end,X}} = t_{\mathsf{post}}(\mathbf{F}^i_{\mathbf{X}})$
Verlassen des Funktors	/ Datenaufbereitungsende:	$t^i_{\mathsf{leave,X}} = t^i_{\mathsf{post.end,X}} = t_{\mathsf{leave}}(\mathbf{F}^i_{\mathbf{X}})$

Funktorphasen:				Dauer:
Phase 1: Vorbereitung:	$i^i_{\mathsf{pre,X}} = i_{\mathsf{pre}}(\mathbf{F}^i_{\mathbf{X}})$	$= [t^i_{\mathsf{pre,X}}, t^i_{\mathsf{do,X}}]$	:	$\tau^i_{\mathsf{pre,X}} = \tau_{\mathsf{d}}(i^i_{\mathsf{pre,X}})$
Phase 2: Berechnung:	$i^i_{\mathsf{do,X}} = i_{\mathsf{do}}(\mathbf{F}^i_{\mathbf{X}})$	$= [t^i_{\mathsf{do,X}}, t^i_{\mathsf{post,X}}]$	:	$\tau^i_{\mathsf{do,X}} = \tau_{\mathsf{d}}(i^i_{\mathsf{do,X}})$
Phase 3: Datenaufbereitung:	$i^i_{\mathsf{post,X}} = i_{\mathsf{post}}(\mathbf{F}^i_{\mathbf{X}})$	$= [t^i_{\mathsf{post,X}}, t^i_{\mathsf{leave,X}}]$	:	$\tau^i_{\mathsf{post,X}} = \tau_{\mathsf{d}}(i^i_{\mathsf{post,X}})$
Gesamter Aufruf:	$i^i_{\mathsf{call,X}} = i_{\mathsf{call}}(\mathbf{F}^i_{\mathbf{X}})$	$= [t^i_{\mathsf{call,X}}, t^i_{\mathsf{leave,X}}]$	:	$\tau^i_{\mathsf{call,X}} = \tau_{\mathsf{d}}(i^i_{\mathsf{call,X}})$

Die Dauer der Vorbereitungsphase τ_{pre} hängt in erster Linie von der Art des Zugriffs auf die Eingangsdaten ab. Darauf wird im Rahmen der Untersuchungen zum Zeitverhalten von Sensoren im Abschnitt 5.3 noch genauer eingegangen. In die zweite, die Berechnungsphase, fällt der eigentliche Operatoraufruf, d.h. die Dauer dieser Phase τ_{do} hängt von den konkreten Datenverarbeitungsalgorithmen ab. Diese Phase bestimmt wesentlich den Ressourcenverbrauch des Funktors und ist daher verantwortlich für das Einhalten eventuell geforderter Rechenzeitrestriktionen. In der Datenaufbereitungsphase schließlich werden die Ausgangsdaten mit einem Zeitstempel versehen und in die entsprechenden Ausgangssequenzen eingefügt, beides Aktionen, die relativ wenig Rechenzeit in Anspruch nehmen. Werden von den Sequenzen allerdings die von ihnen abhängigen Funktoren $\{\mathbf{F}_{\mathbf{dep}}\}$ synchron aufgerufen, fallen diese Aktionen ebenfalls in diese Arbeitsphase und bestimmen maßgeblich deren Dauer τ_{post}.

Für die bessere Vergleichbarkeit von Operationen in Systemen mit präemptiven Multitasking kann neben dem realen Zeitraum, den der Algorithmus für seine Berechnungen braucht ($i_{\mathsf{do}}, \tau_{\mathsf{do}}$), auch die CPU-Zeit ($\tau_{\mathsf{CPU}}$) verwendet werden, insofern das Betriebssystem eine entsprechende Anfrage unterstützt. Hierbei wird nur die vom Operatorprogrammzweig tatsächlich verwendete CPU-Rechenzeit bestimmt. Andere Prozesse, die quasi-parallel auf dem gleichen Prozessor laufen, sowie Prozeßwechselzeiten werden in die CPU-Zeit nicht mit eingerechnet. Abb. 5.2 faßt alle Zeiten, die die zyklische Arbeit der Funktoren beschreiben, zusammen.

5.3 Charakteristische Sensorzeiten

5.3.1 Eigenschaften logischer Sensoren

Zu den allgemeinen Sensoreigenschaften, die vom konkreten Typ der erfaßten Daten weitestgehend unabhängig sind, gehören verschiedene typische Arbeitsmodi, mögliche Zugriffsmechanismen auf den Sensor sowie sein Zeitverhalten, das eng mit den beiden zuvor genannten Größen zusammenhängt. Dabei stellen die *Wartezeit* (τ_{w}) und die *Datenverzögerung*, also das Alter

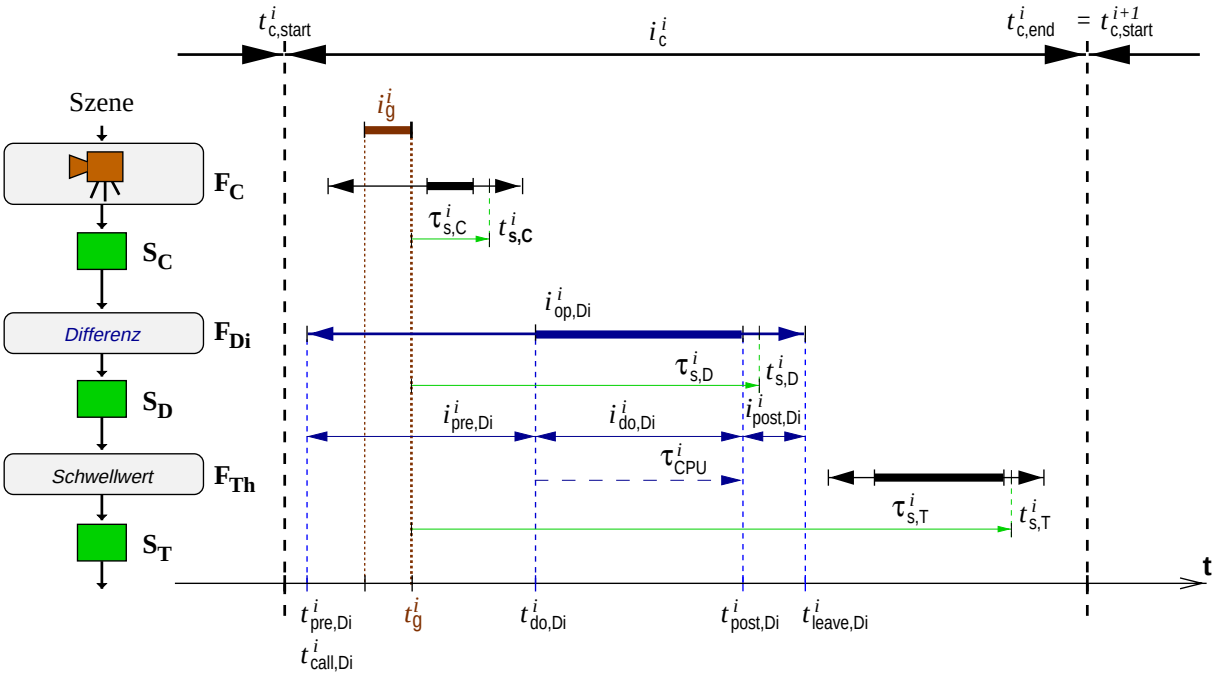


Abbildung 5.2: Charakteristische Bearbeitungszeiten.

der Daten zum Zeitpunkt ihrer Verfügbarkeit (τ_s, vgl. Abschnitt 5.2), die aus Anwendersicht charakteristischen Größen für die Beschreibung des Zeitverhaltens dar. Physikalische Sensoren werden darüber hinaus durch die Meßdauer (τ_g) charakterisiert. Schließlich ist für praktische Belange noch die Meß- oder Framerate ($f_g = \tau_{cg}^{-1}$) von Bedeutung, also der Kehrwert der Zeitspanne, die zwischen zwei Aufrufen vergeht (Wiederholzeit). Diese Wiederholzeit τ_{cg} entspricht bei zyklischer Sensoraktivität ohne Pipeliningbetrieb der Dauer des Arbeitszyklus, in den der Sensor eingebunden wird. Werden die logischen Sensordaten in einem unabhängigen Modul bereitgestellt, hängt τ_{cg} in erster Linie von der Gesamtdauer der Sensoraktivität τ_{call} ab. Dabei kann die Wiederholzeit im Normalfall jedoch nicht kleiner sein, als die des zugrundeliegenden physikalischen Sensors. Eine Ausnahme wäre ein logischer Sensor, der die Daten des Eingangssensors interpoliert weitergibt. Einige praktische und für den RoboCup relevanten Datenraten werden in der folgenden Tabelle zusammengefaßt:

Sensor		Wiederholzeit τ_{cg}	Datenrate f_g
Kamera-Halbbild	PAL	20 ms	50 s^{-1}
	NTSC	16.67 ms	60 s^{-1}
Kamera-Vollbild	PAL	40 ms	25 s^{-1}
	NTSC	33.33 ms	30 s^{-1}
Roboter-Odometrie	Pioneer 1	100 ms	10 s^{-1}
Ultraschallsensoren – beliebiger Sonar	Pioneer 1	100/2 ms	20 s^{-1}
– kompletter Satz von 7 Sonaren		100∗7/2 ms	2.86 s^{-1}

Die Wartezeit beschreibt die Zeitspanne, die zwischen einem Zugriff auf einen Sensor und der Bereitstellung gültiger Daten vergeht: $\tau_w^i = t^i_{s,Sensordatum} - t^i_{call,Sensor}$. Die Datenverzögerung dagegen ergibt sich aus der Zeitspanne zwischen der Datenmessung in der Szene mit

einem physikalischen Meßfühler und der Bereitstellung der Datenwerte durch den jeweiligen logischen Sensor: $\tau_s^i = t^i_{s,\text{Sensordatum}} - t^i_{g,\text{Sensordatum}}$.

Diese Größen stehen in einem engen Zusammenhang mit dem Arbeitsmodus der Sensoren. Hier stehen sich im wesentlichen zwei Arbeitsweisen als auch zwei Zugriffsmethoden gegenüber: Sensoren können die Daten kontinuierlich erfassen (*kontinuierlicher Modus*) oder durch Verbraucheranfragen getriggert werden (*getriggerter Modus*). Im kontinuierlichen Modus ist der Sensor selbst aktiv, d.h. er besitzt seinen eigenen Arbeitszyklus bzw. Programmzweig, der parallel zu den anderen Programmkomponenten arbeitet.

Darüber hinaus kann der Zugriff auf einen Sensor in jedem der beiden Arbeitsmodi *synchron* oder *asynchron* erfolgen. Alle Methoden haben ihre speziellen Vor- und Nachteile, weswegen sich die Auswahl der Verfahren an den Anforderungen der jeweiligen Anwendung und den zur Verfügung stehenden Hardwareressourcen ausrichten muß.

Der Vorteil der kontinuierlichen Datenerfassung gegenüber dem getriggerten Modus liegt in der ständigen Verfügbarkeit der aktuellsten Daten. Dem steht als Nachteil der hohen Ressourcenverbrauch gegenüber. Muß sich der Sensor die Rechenleistung oder andere Hardwareressourcen mit weiteren Programmkomponenten teilen, können diese durch den Sensor ausgebremst oder blockiert werden. Das kann wiederum zur Folge haben, daß nur ein Bruchteil der bereitgestellten Daten auch ausgewertet wird.

Varianten oder Mischformen der verschiedenen Arbeitsweisen können dazu beitragen, die Datenerfassung zu optimieren. So kann z.B. beim kontinuierlichen Modus die Datenrate künstlich reduziert werden, indem etwa nur jedes zweite von einer Kamera gelieferte Halbbild verwendet wird. Eine zweite Möglichkeit besteht darin, erst kurz bevor ein Datenzugriff erwartet wird in den kontinuierlichen Modus zu wechseln, und nach der Datenabfrage diesen sofort wieder abzuschalten, um so die Systemressourcen den anderen Programmkomponenten wieder uneingeschränkt zur Verfügung zu stellen. Im *pseudo-kontinuierlichen Modus* ist schließlich nicht der Sensor selbst aktiv, sondern er wird (zusammen mit anderen logischen Sensoren) von einer aktiven Programmkomponenten (etwa einem Softwareagenten) kontinuierlich getriggert. Das hat den Vorteil, daß der Softwareagent die Datenrate kontrollieren und in Abhängigkeit vom Ressourcenbedarf des Gesamtsystems variieren kann.

Die Entscheidung, ob synchron oder asynchron auf die Sensordaten zugegriffen werden soll, beeinflußt bei vielen Sensoren entscheidend das Alter der Daten und die Wartezeit auf diese. Beim synchronen Sensorzugriff wartet der Anwender auf die nächsten Daten, was als Vorteil die geringstmögliche Datenverzögerung mit sich bringt. Dem steht als Nachteil eine u.U. höhere Wartezeit gegenüber. Im Gegensatz dazu reduziert der asynchrone Sensorzugriff die Wartezeit auf ein Minimum, indem auf zuvor bereitgestellte Daten zugegriffen wird. Dabei wird in Kauf genommen, daß die Daten bei ihrer Verwendung älter sind als beim synchronen Zugriff. Zusätzlich muß, wenn der Sensor nicht kontinuierlich arbeitet, vor dem eigentlichen Datenzugriff die Messung vom Anwender rechtzeitig initialisiert werden. Erfolgt diese Initialisierung nicht zum optimalen Zeitpunkt, sind die Daten beim Zugriff entweder älter als notwendig, oder der Anwenderprozeß muß wie beim synchronen Zugriff auf die Daten warten.

Die Wartezeit beim synchronen Sensorzugriff hängt wesentlich vom Arbeitsmodus des Sensors ab. Arbeitet dieser kontinuierlich, fällt sie geringer aus, da zum Zeitpunkt der Anfrage der aktuelle Datensatz bereits bearbeitet wird. Im getriggerten Modus dagegen wird die komplette Datenerfassung und -auswertung erst durch die Anfrage eingeleitet und bei Kamerazugriffen fällt z.B. der komplette Framegrabberaufruf in diese Wartephase.

Schließlich ist zu beachten, daß logische Sensoren sich aufeinander und letztendlich auch auf physikalische Sensoren abstützen, d.h. jeder logische Sensor greift auf die Daten seiner *Eingangssensoren* zu. Für jeden dieser Sensoren ist nun ebenfalls zu entscheiden, in welchem Modus er arbeiten und wie er auf seine Eingangssensoren zugreifen soll, um das System optimal auszulasten. Dabei kann die Arbeitsweise von physikalischen Meßfühlern und Sensorgeräten systembedingt vorgegeben sein, die Softwareeinbindung aber bereits davon abweichen, und z.B. aus einem kontinuierlichen Videosignal nur bei Bedarf einzelne Bilder in eine Softwarerepräsentation umwandeln. Das Zeitverhalten eines logischen Sensors läßt sich dabei rekursiv aus dem Zeitverhalten der verwendeten Sensoren ableiten.

5.3.2 Phasen bei der Bereitstellung von Sensordaten

Physikalische Sensoren können, wie Operatoren auch, mit Hilfe von Funktoren in ein Softwaresystem eingebunden werden, und beliebige Operatoren lassen sich als logische oder virtuelle Sensoren interpretieren. Bei der Untersuchung des Zeitverhaltens von Sensoren liegt es daher nahe, von den für die Arbeitsweise von Operatoren bzw. Funktoren charakteristischen Bearbeitungsphasen und deren Zeitverhalten auszugehen. An dieser Stelle soll — um die verschiedenen Arbeits- und Zugriffsmodi von Sensoren besser charakterisieren zu können — die Vorbereitungsphase $\mathsf{pre}()$, in der auf die Eingangsdaten oder den zugrundeliegenden physikalischen Sensor zugegriffen wird, nochmals unterteilt werden. Die Datenbearbeitungsphase $\mathsf{do}()$ und die Nachbearbeitung $\mathsf{post}()$ bleiben unverändert auch für die Sensorsichtweise bestehen.

Eine detaillierte Beschreibung der einzelnen Phasen stellt zum einen eine wichtige Planungsgrundlage für den Einsatz von Sensoren sowie die Auswahl optimaler Arbeitsmodi und Zugriffsmethoden dar, und sie ist zum anderen die Voraussetzung für ein Abschätzen des Zeitverhaltens eines konkreten logischen Sensors. Die folgenden Bearbeitungsschritte sollen unterschieden werden, wobei die ersten drei Phasen zusammen die Vorbereitungsphase $\mathsf{pre}()$ der Funktoraufrufe entsprechend Abschnitt 5.2 bilden:

Zugriffsvorbereitung:

$$i^i_{\mathsf{pre}',\mathsf{X}} = i_{\mathsf{pre}'}(\mathbf{F}^i_{\mathbf{X}}) = [t_{\mathsf{call}}(\mathbf{F}^i_{\mathbf{X}}), t_{\mathsf{requ}}(\mathbf{F}^i_{\mathbf{X}})], \quad \tau^i_{\mathsf{pre}',\mathsf{X}} = \tau_{\mathsf{d}}(i^i_{\mathsf{pre}',\mathsf{X}})$$

Die Vorbereitungsphase dient der Initialisierung der verwendeten Eingangssensoren sowie des aktuellen Aufrufs. In diese Phase fallen z.B. Tests, ob die zuletzt ermittelten Daten noch aktuell genug sind oder mit neuen Eingangsdaten der Sensor aktualisiert werden muß. Eine andere Aufgabe kann das Bereitstellen von Speicherbereichen für die Ergebnisdaten oder die Initialisierung statistischer Größen sein.

Bei der Initialisierung der Eingangssensoren muß zwischen der initialen Konfiguration sowie kleineren Voreinstellungen für den aktuell geplanten Zugriff unterschieden werden. Ersteres ist oft deutlich aufwendiger, beispielsweise wenn es umfangreiche Möglichkeiten gibt, einen Sensor zu programmieren. Letzteres umfaßt dagegen nur das Setzen einzelner Parameter, um den Sensorzugriff in geeigneter Weise anzupassen.

Sensordatenanforderung:

$$i^i_{\mathsf{requ},\mathsf{X}} = i_{\mathsf{requ}}(\mathbf{F}^i_{\mathbf{X}}) = [t_{\mathsf{requ}}(\mathbf{F}^i_{\mathbf{X}}), t_{\mathsf{get}}(\mathbf{F}^i_{\mathbf{X}})], \quad \tau^i_{\mathsf{requ},\mathsf{X}} = \tau_{\mathsf{d}}(i^i_{\mathsf{requ},\mathsf{X}})$$

Während dieser Phase wartet der Sensor auf seine Eingangsdaten. Wird auf alle Eingangsdaten asynchron zugegriffen und stehen diese kontinuierlich oder durch einen vorhergehenden Sensoraufruf bereits zur Verfügung, muß nicht gewartet werden und diese Phase entfällt. Anderenfalls synchronisiert sich der Sensor durch diese Phase mit seinen Eingangssensoren.

Sensordatenabruf:

$$i^i_{\mathsf{get,X}} = i_{\mathsf{get}}(\mathbf{F}^i_{\mathbf{X}}) = [t_{\mathsf{get}}(\mathbf{F}^i_{\mathbf{X}}), t_{\mathsf{do}}(\mathbf{F}^i_{\mathbf{X}})], \quad \tau^i_{\mathsf{get,X}} = \tau_{\mathsf{d}}(i^i_{\mathsf{get,X}})$$

In dieser Phase erhält der Sensor von seinen Eingangssensoren die angeforderten Daten oder eine Referenz darauf. Die beiden Phasen Sensordatenanforderung und -abruf müssen für jeden Eingangssensor durchlaufen werden. Sie können nacheinander, Sensor für Sensor, abgearbeitet werden, oder es werden zuerst alle Eingangsdaten asynchron angefordert und, sobald die Daten anliegen, abgefragt.

Datenbearbeitung:

$$i^i_{\mathsf{do,X}} = i_{\mathsf{do}}(\mathbf{F}^i_{\mathbf{X}}) = [t_{\mathsf{do}}(\mathbf{F}^i_{\mathbf{X}}), t_{\mathsf{post}}(\mathbf{F}^i_{\mathbf{X}})], \quad \tau^i_{\mathsf{do,X}} = \tau_{\mathsf{d}}(i^i_{\mathsf{do,X}})$$

Die meisten (logischen) Sensoren führen eine gewisse Aufbereitung, Modifizierung, Filterung oder Auswertung der Eingangsdaten durch, was in dieser Phase erfolgt.

Nachbearbeitung:

$$i^i_{\mathsf{post,X}} = i_{\mathsf{post}}(\mathbf{F}^i_{\mathbf{X}}) = [t_{\mathsf{post}}(\mathbf{F}^i_{\mathbf{X}}), t_{\mathsf{leave}}(\mathbf{F}^i_{\mathbf{X}})], \quad \tau^i_{\mathsf{post,X}} = \tau_{\mathsf{d}}(i^i_{\mathsf{post,X}})$$

Schließlich müssen die ermittelten Daten softwaretechnisch so aufbereitet werden, daß andere Komponenten darauf zugreifen können. In der Nachbearbeitungsphase können daher entsprechende Sequenzwertobjekte erzeugt und in Datensequenzen bereitgestellt werden. Außerdem kann z.B. eine statistische Auswertung des Sensorzugriffs erfolgen. Schließlich fallen in diese Phase auch Aktionen, die mit allen neu ermittelten Sensordaten auszuführen sind. Das kann beispielsweise das Protokolieren der Daten, deren Darstellung auf dem Bildschirm oder deren Verschicken in einem Netzwerk beinhalten.

Zu beachten ist, daß die Phasen, die in erster Linie dem Daten- oder Aufrufmanagement dienen ($i_{\mathsf{pre'}}$, i_{post} und u.U. auch i_{get}), i.d.R. deutlich weniger Rechenzeit in Anspruch nehmen als die eigentliche Datenverarbeitung (i_{do}) oder auch die Synchronisation mit den Eingangsdaten (i_{requ}). Für die Gesamtrechenzeit der Sensoroperatoren sind daher in erster Linie die zuletzt genannten Phasen von Bedeutung.

Abb. 5.3 illustriert anhand des Einlesens von Bildern mit Hilfe einer konventionellen Videokamera und eines Framegrabbers die einzelnen Sensorphasen und das Zusammenspiel der verschiedenen Sensorebenen bei einem synchronen Datenzugriff. Dargestellt werden die Sensorebenen: CCD-Chip (Ladungen entsprechend des optischen Abbilds), DA-Wandler der Kamera (Videosignal), Framegrabber (digitale Bildmatrix) und Kamerabildfolge (Softwarerepräsentation). Dabei kann jede dieser Ebenen wiederum selbst als ein Sensor betrachtet werden.

Auf der untersten Ebene befindet sich der CCD-Chip, der als physikalischer Meßfühler ein optisches Abbild der Szene zuerst in elektrische Ladungen und dann in Spannungswerte umwandelt ④. Ein Shutter sorgt für eine definierte Belichtungszeit. Nach der Belichtung eines Halbbildes werden die Ladungswerte ausgelesen und zwischengespeichert, bevor das Bild vom DA-Wandler der Kamera zeilen- und pixelweise in ein analoges Videosignal umgewandelt wird ⑤. Währenddessen wird bereits das nächste Halbbild belichtet.

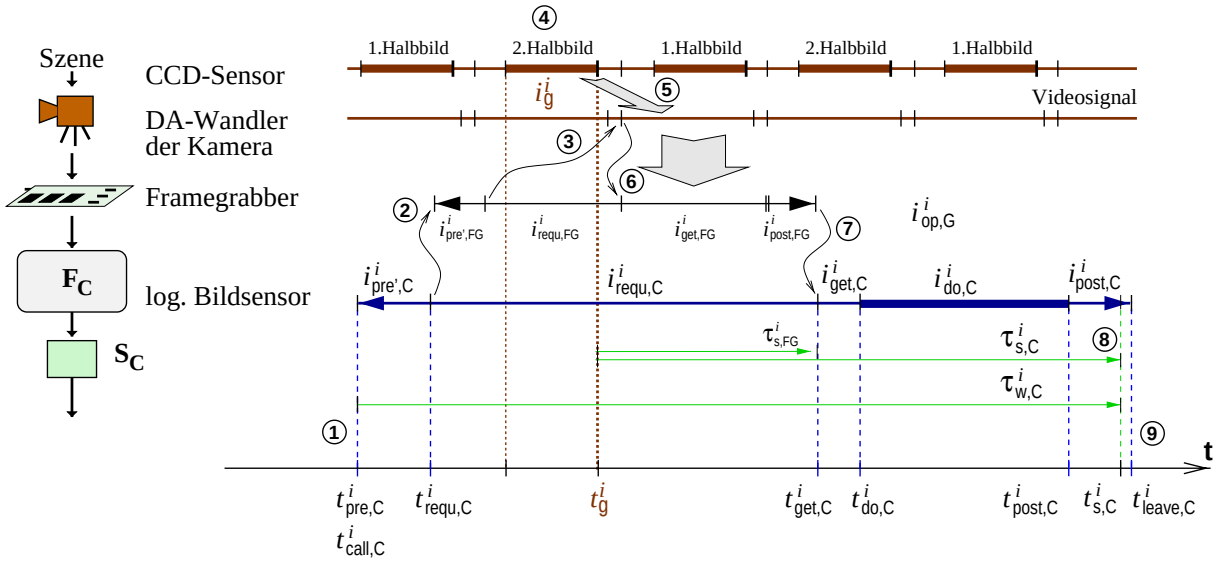


Abbildung 5.3: Arbeitsphasen und Zeitverhalten logischer und physikalischer Sensoren am Beispiel des synchronen Einlesens eines einzelnen Videohalbbildes.

Das Videosignal wiederum wird im Framegrabber diskretisiert und als digitale Bildmatrix in den Hauptspeicher geschrieben. Dazu wartet der Framegrabber auf das nächste Bildsynchronisationssignal ③, das ihm anzeigt, wann im Videosignal ein neues Halbbild beginnt. Daraufhin wird mit Hilfe eines Analog-Digital-Wandlers das Videosignal zeilenweise digitalisiert und die Bildmatrix der aufrufenden Methode übergeben ⑥. Eine weitere Bearbeitung der Daten findet im Framegrabber nicht statt, weswegen die Bearbeitungsphase entfällt: $\tau_{\mathsf{do,FG}} = 0$.

Die oberste Sensorebene wird in diesem Beispiel schließlich durch den Kamerabild-Funktor bzw. logischen Bildsensor gebildet. Auf diesen greifen die Module in einem Bildfolgenprogramm zu, um digitalisierte Kamerabilder zu erhalten. Fordern sie ein neues Bild an ①, werden zuerst der Funktor und der von ihm gekapselte Operator, der den Framegrabberzugiff durchführt, initialisiert ($i_{\mathsf{pre',C}}$). Danach greift der Operator auf den Framegrabbertreiber zu und fordert von ihm das nächste Bild an ②. Aufgrund des synchronen Zugriffs auf den Framegrabber bleibt der Operator solange blockiert, bis der Framegrabber ein neues Bild hat $i_{\mathsf{requ,C}}$ ⑦. Da der Framegrabber in diesem Beispiel nicht kontinuierlich läuft, muß er zuerst auf das nächste Halbbild warten ③, bevor mit dessen Digitalisierung begonnen werden kann ⑥.

Nachdem der Framegrabber seine Aufgabe erfüllt und das Kamerabild dem Rechner zur Verfügung gestellt hat, kehrt die Steuerung zum Funktor zurück ⑦. Er kann jetzt die digitalisierten Sensordaten, d.h. die Bildmatrix abfragen ($i_{\mathsf{get,C}}$) und entsprechend des geforderten Bildformats aufbereiten ($i_{\mathsf{do,C}}$). Beim Sensordatenabruf wird entweder die komplette Bildmatrix kopiert oder eine Referenz auf den Speicherbereich, in dem das Bild liegt, übergeben. Außerdem kann z.B. die Meßzeit, wenn sie vom Framegrabbertreiber mitprotokolliert wurde, abgefragt werden. Als Datenverarbeitung kann ein Umkopieren der Daten erfolgen — etwa um gepackte Pixeldaten in planare Farbkanäle umzuwandeln. Eine Farbraumtransformation oder grundlegende Filteroperationen sind ebenfalls möglich. Diese Phase kann aber auch entfallen, wenn die Daten schon beim Sensordatenabruf kopiert wurden und bereits in der gewünschten Form vorliegen.

Zum Abschluß des Operatoraufrufs wird die Bildmatrix in ein geeignetes Softwareobjekt eingebunden oder mit ihm assoziiert und zum Zeitpunkt $t_{s,C}$ den anderen Modulen zur Verfügung gestellt ⑧. Daran anschließend wird der Operator verlassen ⑨. Für praktische Belange kann mit $t_{leave,C}$ als oberer Schranke für $t_{s,C}$ gerechnet werden. Die für den Anwender interessanten Zeitwerte lassen sich in diesem Beispiel dann wie folgt abschätzen:

Datenverzögerung:

$$\tau_{s,C} = \tau_{s,FG} + \tau_{get,C} + \tau_{do,C} + \tau_{post,C} \quad \text{mit}$$

$\tau_{s,FG} = const. \approx \begin{cases} 20 \text{ ms}, & \text{pro Halbbild im PAL-Format} \\ 16.7 \text{ ms}, & \text{pro Halbbild im NTSC-Format} \end{cases}$
Durch Sequentialisierung bedingte Verzögerung zwischen der Belichtung des CCD-Sensors und dem Bildende im Videosignal

$\tau_{get,C} \to 0,$ Kopieren einer Referenz auf den Speicherbereich, in dem die Bilddaten stehen

$\tau_{do,C} \to$ Nachbearbeitung des Bildes, z.B.:

- 0 Grauwertbild: nur umkopieren der Matrix als Ganzes
- $\vdots$ Farbbild: gepackte Pixel in planare Farbkanäle umwandeln
- τ_{max} weitere Bearbeitung: Farbraumtransformation, Klassifikation

$\tau_{post,C} \to 0,$ Neues Bild der Ausgangsdatensequenz hinzufügen (kein Aufruf abhängiger Funktoren $\mathbf{F_{dep}}$ im gleichen Programmzweig)

Alle angegebenen Teilintervalle sind unabhängig vom Arbeitsmodus des Framegrabbers und der Zugriffsmethode. $\tau_{get,C}$ und $\tau_{post,C}$ sind i.d.R. vernachlässigbar klein, so daß die Datenverzögerung sich nur durch einen Austausch der Datenverarbeitungsalgorithmen reduzieren läßt.

Wartezeit:

$$\tau_{w,C} = \tau_{pre',C} + \tau_{requ,C} + \tau_{get,C} + \tau_{do,C} + \tau_{post,C} \quad \text{mit}$$

$\tau_{pre',C} \to 0,$ Framegrabberparameter setzen

$\tau_{requ,C} = \tau_{call,FG},$ bei synchronem Zugriff auf getriggerten Framegrabber

$= 0,$ bei asynchronem Zugriff auf kontinuierlichen Framegrabber

$\tau_{get,C} \to 0,$ s.o.

$\tau_{do,C}$ s.o.

$\tau_{post,C} \to 0,$ s.o.

Die Wartezeit hängt in diesem Beispiel neben der eigentlichen Datenverarbeitung im Operator auch vom Framegrabberaufruf ab. Dieser kann stark variieren, da sich durch den getriggerten Arbeitsmodus des Framegrabbers dieser erst mit dem Videosignal synchronisieren muß. Zusammen mit der Digitalisierung kann dies bei einem PAL-Signal — je nachdem ob ein beliebiges Halbbild, ein bestimmtes Halbbild oder ein Vollbild gegrabbt werden soll — bis zu 40, 60 oder 80 ms dauern. Dies kann durch einen Wechsel der Zugriffsart und des Arbeitsmodus des Framegrabbers reduziert werden. Beim asynchronen Framegrabberzugriff entfällt die Anforderungsphase ganz: $\tau_{requ,C} = 0$. Statt dessen steigt die Datenverzögerungszeit. Arbeitet der Framegrabber im kontinuierlichen Modus, wird

sich die Anforderungsphase auch beim synchronen Zugriff reduzieren, da zum Zeitpunkt der Datenanforderung der Framegrabber bereits das gewünschte Bild bearbeitet.

Abb. 5.4 faßt die Zeitcharakteristiken für die unterschiedlichen Zugriffsmethoden und Arbeitsmodi für Sensoren mit einem Eingangssensor zusammen und stellt die sich ergebenden Zeiten (Wartezeit und Datenverzögerung) einander gegenüber. Für eine bessere Vergleichbarkeit der charakteristischen Daten wurden die Aufrufzeiten des Kamerafunktors $\mathbf{F_C}$ so plaziert, daß bei allen Sensorzugriffen immer das gleiche Bild geliefert wird, wobei jeweils der günstigste und der ungünstigste Fall (kleinster und größter Zeitwert) angegeben wurde. Bei der

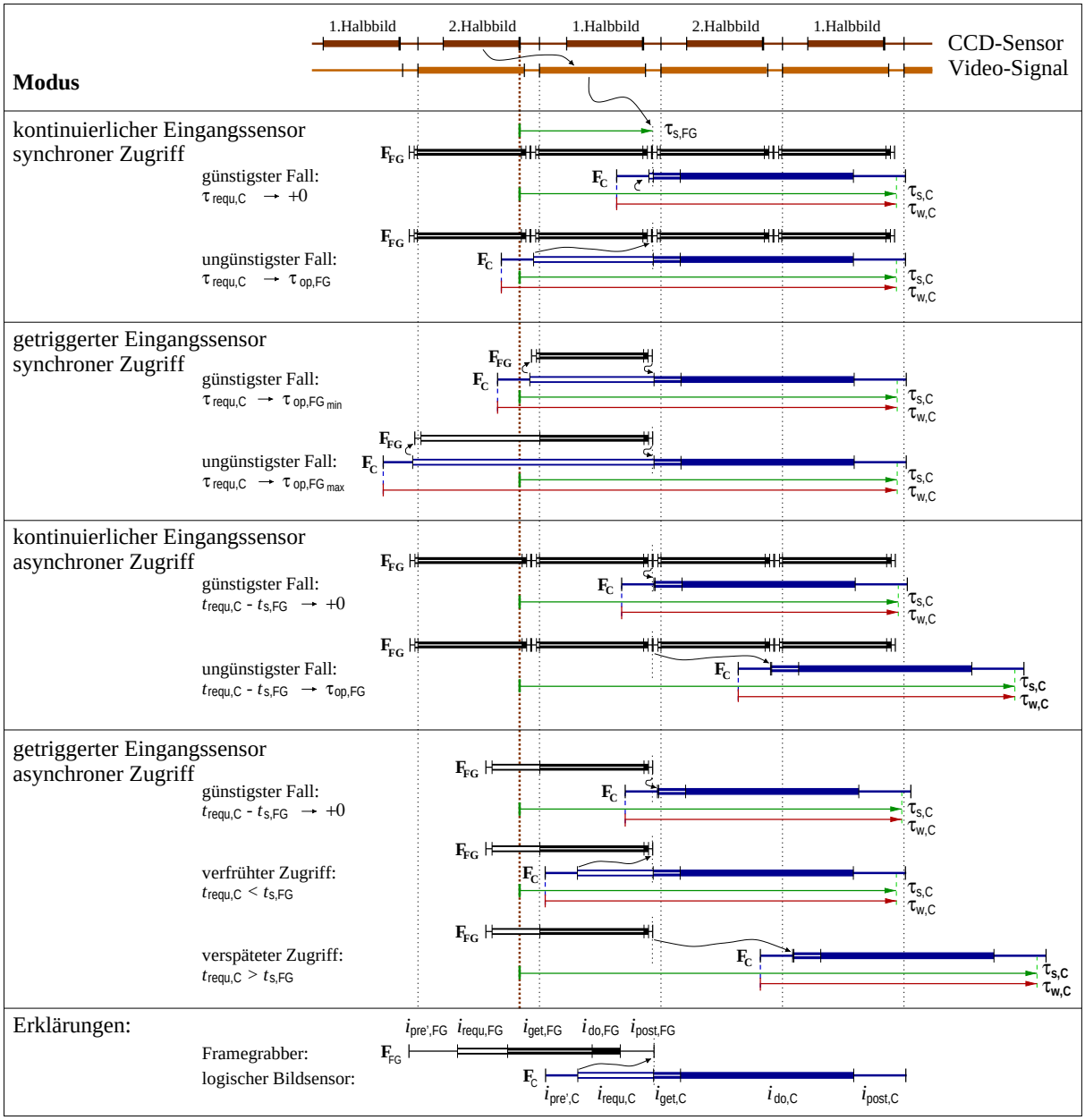


Abbildung 5.4: Gegenüberstellung unterschiedlicher Zeitverhalten eines Sensors ($\mathbf{F_C}$) in Abhängigkeit vom Arbeitsmodus des Eingangssensors ($\mathbf{F_{FG}}$) und der Art des Zugriffs auf ihn.

Untersuchung der ungünstigen Fälle wurde davon ausgegangen, daß immer das jeweils nächste *Halbbild* verwendet werden kann. Anderenfalls (Vollbild oder ein bestimmtes Halbbild) würde sich der Framegrabberaufruf $\tau_{\mathsf{call,FG}}$ entsprechend verlängern.

Hier soll davon ausgegangen werden, daß der Framegrabber schnell genug ist, um im kontinuierlichen Modus jedes Halbbild digitalisieren zu können, daß also keine Wartezeiten entstehen, weil der Framegrabber die Daten nicht schnell genug in den Hauptspeicher transportieren konnte. Das bedeutet, daß die Vorbereitungsphase, die Datenverarbeitungsphase und die Nachbearbeitung im Framegrabber zusammen kürzer sein müssen als die Austastphase des Videosignals τ_{out} zwischen zwei Halbbildern:

$$\tau_{\mathsf{pre',FG}} + \tau_{\mathsf{do,FG}} + \tau_{\mathsf{post,FG}} < \tau_{\mathsf{out}}, \quad \text{mit Austastphase } \tau_{\mathsf{out}}$$

$$\tau_{\mathsf{out}} = \begin{cases} 1.568\,\mathsf{ms} \;= 20.00\,\mathsf{ms} - 288\,\mathsf{Zeilen} * 64.0\,\mu\mathsf{s/Zeile}, & \text{PAL} \\ 1.427\,\mathsf{ms} \;= 16.67\,\mathsf{ms} - 240\,\mathsf{Zeilen} * 63.5\,\mu\mathsf{s/Zeile}, & \text{NTSC} \end{cases}$$

Wenn diese Bedingung erfüllt ist, sind die Datenraten f_{g} bzw. die Wiederholzeiten τ_{cg} für die Kamera ($\mathbf{F_{CCD}}$) und die Framegrabbereinbindung ($\mathbf{F_{FG}}$) gleich: $f_{\mathsf{g,CCD}} = f_{\mathsf{g,FG}}$ bzw. $\tau_{\mathsf{cg,CCD}} = \tau_{\mathsf{cg,FG}}$. Damit auch der logische Kamerafunktor $\mathbf{F_C}$ diese Datenrate erreicht ($f_{\mathsf{g,C}} \stackrel{!}{=} f_{\mathsf{g,FG}}$), muß in Abhängigkeit vom jeweiligen Arbeitsmodus des Framegrabbersensors $\mathbf{F_{FG}}$ die folgende Bedingung hinsichtlich der Rechenzeit, die in dem logischen Funktor verbraucht wird ($\tau'_{\mathsf{call,C}} = \tau_{\mathsf{call,C}} - \tau_{\mathsf{requ,C}}$), erfüllt sein:

kontinuierlicher Modus: $\tau'_{\mathsf{call,C}} < \tau_{\mathsf{cg,FG}}$

getriggerter Modus: $\tau'_{\mathsf{call,C}} < \tau_{\mathsf{cg,FG}} - t_{\mathsf{call,FG}}$

mit $\tau'_{\mathsf{call,C}} = \tau_{\mathsf{call,C}} - \tau_{\mathsf{requ,C}}$
$= \tau_{\mathsf{get,C}} + \tau_{\mathsf{do,C}} + \tau_{\mathsf{post,C}} + \tau_{\mathsf{pre',C}}$

Dies verdeutlicht, daß für Kameras, aber auch für viele andere Sensoren deren kontinuierliche Arbeitsweise eine notwendige Voraussetzung ist, um die durch einen logischen Sensor aufbereiteten Daten mit einer vergleichbaren Datenrate bereitstellen zu können.

5.4 Zykluszeit als Index für den Datenzugriff

In einem Bildfolgenprogramm arbeiten viele Programmkomponenten zyklisch, wobei verschiedene Komponenten $\mathcal{M}_{\mathsf{X}}$ mit unterschiedlichen Zyklen $\mathcal{M}^i_{\mathsf{X}}$ arbeiten können. In einem solchen Zyklus berechnet ein Modul alle Daten, für die es verantwortlich ist, neu und stellt diese Daten anderen Programmkomponenten zur Verfügung. Dafür können die Daten anderer Module als Eingangsdaten verwendet und verschiedene Zwischenergebnisse, und daraus dann die neuen Ausgangsdaten, bestimmt werden. Wenn das geschehen ist, beginnt der nächste Zyklus und damit die Berechnungen — mit neuen Eingangsdaten — von vorn.

Dadurch, daß sich die neuen Ausgangsdaten eines Moduls aus verschiedenen Eingangsdaten und Zwischenergebnissen berechnen, die ihrerseits wiederum von anderen zyklischen Daten abhängen, entsteht ein Graph von Abhängigkeiten zwischen den Daten. Abb. 5.5 illustriert das anhand eines Beispiels. Es zeigt, daß innerhalb eines Zyklus mehrfach auf die gleichen Datensequenzen zugegriffen werden kann, wenn sie als Eingangsdaten verschiedener Operatoren verwendet werden. So fließt beispielsweise $\mathbf{S_B}$ in die Berechnungen von $\mathbf{S_D}$ und $\mathbf{S_E}$ ein.

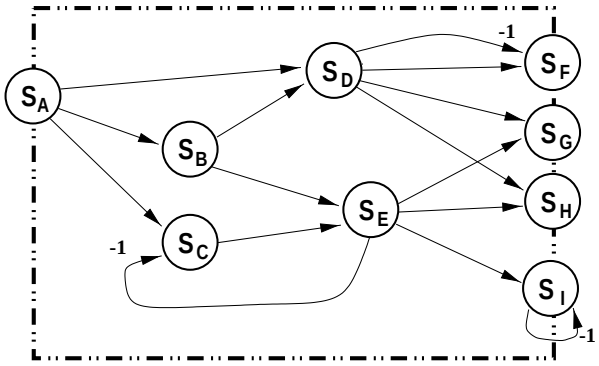


Abbildung 5.5: Beispiel für einen Graphen voneinander abhängiger Sequenzen.

Hierbei muß sichergestellt werden, daß in einem Zyklus immer mit den gleichen Daten gerechnet wird. Hat beispielsweise ein Operator die Aufgabe, Suchbereiche im Bild zu definieren, die ein anderer Operator in einem zweiten Schritt näher untersuchen soll, schlägt dieses Verfahren fehl, wenn der zweite Operator ein neueres Eingangsbild erhält als der erste.

Um derartige Inkonsistenzen bei der Interpretation der Daten zu vermeiden, muß nun verhindert werden, daß bei wiederholten Zugriffen auf eine Sequenz innerhalb des gleichen Zyklus die Sequenz einen anderen, neueren Wert zurückliefert als beim ersten Zugriff. Dies könnte beispielsweise dadurch geschehen, daß eine Eingangssequenz von einer anderen, vom betrachteten Modul unabhängigen Komponente aktualisiert wurde, oder aber dadurch, daß zwischen zwei Anfragen soviel Zeit vergangen ist, daß das Eingangsdatum als zu alt eingestuft wird.

Ein möglicher Ausweg könnte darin bestehen, alle Anfragen an die Meßzeit der zu verarbeitenden Daten zu koppeln. Das Problem dabei ist allerdings, daß die Messung der Sensordaten oft erst durch die Operatoraufrufe eines Zyklus eingeleitet wird, und die Meßzeit damit beim Aufruf der Funktoren noch nicht bekannt ist. Daher soll hier folgendermaßen vorgegangen werden: Zu Beginn eines jeden Modulzyklus wird eine spezielle Zykluszeit festgelegt. Dabei handelt es sich i.d.R. um die Startzeit des Zyklus. Um ein Modul mit einer anderen Komponente zu synchronisieren, kann aber auch deren aktuelle Zykluszeit übernommen werden.

Zykluszeit im aktuellen Zyklus:

$$t^0_{\mathsf{c,X}} = t_{\mathsf{c}}(\mathcal{M}^0_{\mathsf{X}}) = \begin{cases} t_{\mathsf{c.start}}(\mathcal{M}^0_{\mathsf{X}}) & \text{eigene Modulstartzeit} \\ t_{\mathsf{c}}(\mathcal{M}^0_{\mathsf{Y}}) & \text{Zykluszeit eines anderen Moduls} \end{cases}$$

Für alle Datenzugriffe innerhalb eines Moduls ist nun nicht mehr die aktuelle Zeit t_{now} ausschlaggebend, sondern die zuvor festgelegte Zykluszeit t_{c}. Diese fungiert damit als ein Index für den Sequenzzugriff. Statt $\mathbf{S}(t_{\mathsf{now}})$ wird von allen Operatoren $\mathbf{S}(t_{\mathsf{c}})$ verwendet. Durch das Anfordern eines Sequenzwertes für eine bestimmte, konstante Zeit wird sichergestellt, daß immer der gleiche Sequenzwert zurückgeliefert wird. Es ergibt sich damit innerhalb eines Moduls eine eindeutige Abbildung von Zykluszeiten t_{c} auf die Datenmeßzeiten t_{g}.

$$S^i = S(t^i_{\mathsf{c}}) = S(t^i_{\mathsf{g}})$$

Unabhängig davon können andere Module mit abweichenden Zykluszeiten auf die Sequenzen zugreifen und dabei andere Werte erhalten. Für die praktische Realisierung muß dabei sichergestellt werden, daß Sequenzen gleichzeitig mehrere Werte verwalten können.

Schließlich steht mit der Zykluszeit ein Mechanismus zur Verfügung, mit dem der Status von Funktionen und Datensequenzen in einem Zyklus eindeutig bestimmt werden kann. Insbesondere läßt sich so lokal — allein anhand der für den Zugriff verwendeten Zykluszeit — ermitteln, ob ein Funktor innerhalb eines Zyklus bereits aufgerufen und ob eine Datensequenz bereits aktualisiert wurde. Dabei sind Datensequenzen immer so aktuell, wie die Funktoren, die sie bereitstellen:

Zykluszeit des letzten Aufrufs von $\mathbf{F_X}$: $t_c(\mathbf{F_X}) = t_c(\mathbf{F_X^0})$

Zykluszeit der letzten Aktualisierung von $\mathbf{S_X}$: $t_c(\mathbf{S_X}) = t_c(S_X^0)$

$\mathbf{S_{Out}} = \mathbf{F_X}(\mathbf{S_X})$

$\rightarrow t_c(\mathbf{S_{Out}}) = t_c(\mathbf{F_X})$	Aufruf von $\mathbf{F_X}$ bedingt Aktualisierung von $\mathbf{S_{Out}}$.
$\not\rightarrow t_c(\mathbf{S_X}) = t_c(\mathbf{F_X})$	$\mathbf{S_X}$ kann anderweitig aktualisiert worden sein und muß nicht unbedingt den Aufruf von $\mathbf{F_X}$ nach sich ziehen.

Kapitel 6

Funktionale Beschreibung von Bildfolgenprogrammen

Im Mittelpunkt dieses Kapitels sollen verschiedene Aspekte der funktionalen Beschreibung von den Programmkomponenten stehen, die für die kontinuierliche Verarbeitung von Bildfolgen und anderen Sensordaten zuständig sind. Von besonderer Bedeutung für den funktionalen Entwurf ist die Wahl einer geeigneten Abstraktionsebene, in der der Entwurf anzusiedeln ist. In diesem Rahmen sollen Ausdrucksmittel für eine von den zugrundeliegenden Softwarebibliotheken unabhängige Beschreibung der Bereitstellung und Verarbeitung dynamischer Daten untersucht werden.

6.1 Einordnung

6.1.1 Abstraktionsebenen und Beschreibungsmittel

Die Beschreibung einer Applikation und deren Komponenten kann in verschiedenen Ebenen mit unterschiedlichem Abstraktionsgrad erfolgen. Das Spektrum möglicher Abstraktionsebenen reicht dabei, wie Abb. 6.1 zeigt, von einer allgemeinen, abstrakten Programmspezifikation über formalisierte, auf Operatoren und komplexen Datenobjekten basierenden Beschreibungen, bis hin zur Implementierung in einer Programmiersprache und dem ausführbaren Maschinenprogramm.

Für jede Ebene existieren bestimmte, typische Beschreibungsmittel. Zu ihnen zählen beispielsweise imperative Programmiersprachen, objektorientierte Modellierungssprachen, funktionale Sprachen sowie Graphen und Diagramme; aber auch der Maschinencode oder, als anderes Extrem, verbale, formlose Spezifikationsmittel, z.B. in Form von Pflichtenheften, stellen mögliche Beschreibungsformen dar.

6.1.2 Bedeutung der Wiederverwendbarkeit von Systementwürfen und Softwareentwicklungen

Bei der Entwicklung von Softwaresystemen wird i.d.R. von einer abstrakten Programmspezifikation ausgegangen, die in den verschiedenen Entwurfsebenen schrittweise verfeinert und dabei

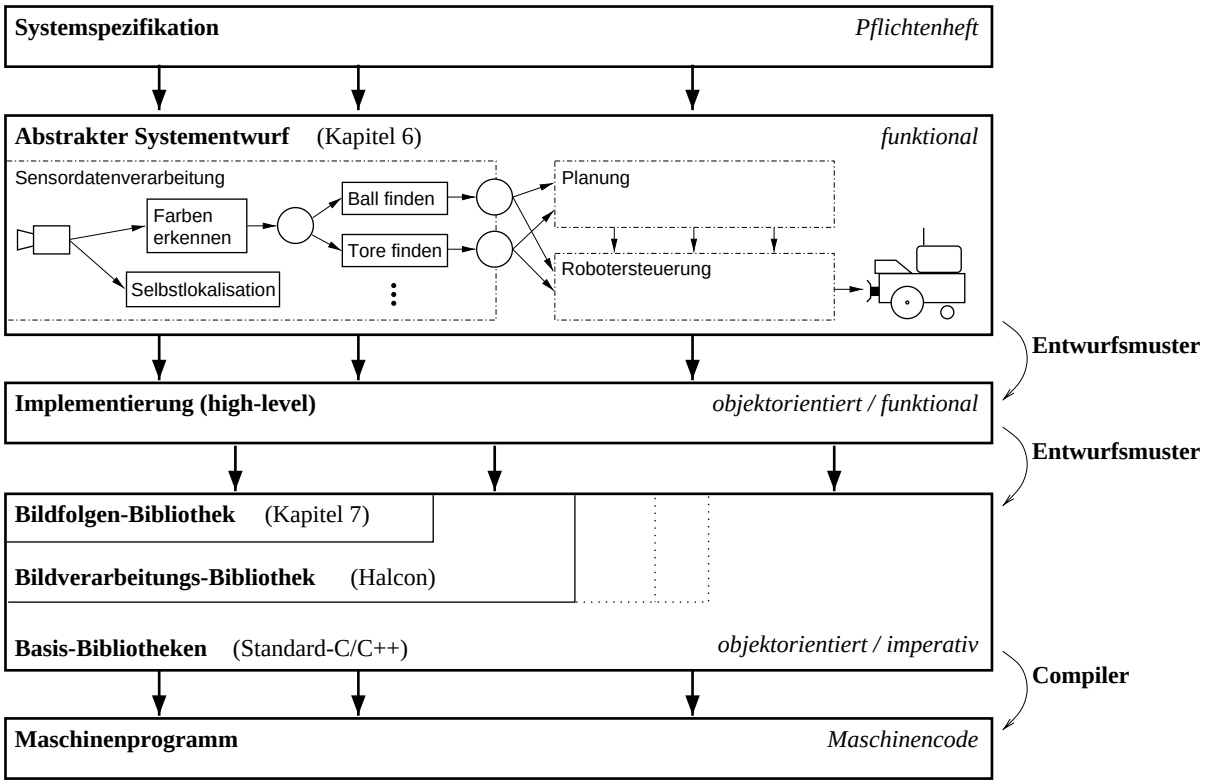


Abbildung 6.1: Hierarchische Gliederung des Programmentwurfs in verschiedene Abstraktionsebenen.

immer stärker konkretisiert wird. Beim Übergang von einer Abstraktionsebene in eine darunterliegende kann die Beschreibungsform wechseln. Dies bedeutet, daß die Elemente einer Ebene auf die korrespondierenden Elemente der nächsten Ebene abgebildet werden müssen. Wie diese Abbildung zu erfolgen hat, hängt von den Entwurfsmethoden und Beschreibungsformen der jeweiligen Ebenen ab.

Für die Effizienz der Entwicklungsarbeit spielt sowohl die Wiederverwendbarkeit der bereits getätigten Entwicklungen als auch die leichte Umsetzung der Entwürfe durch entsprechende Mittel der darunterliegenden Realisierungsebenen eine entscheidende Rolle. Da dies eng mit der Wahl geeigneter Entwurfsmethoden und Beschreibungsformen in den jeweiligen Abstraktionsebenen zusammenhängt, sollen die wichtigsten Vorgehensweisen hier kurz beleuchtet werden. Abb. 6.1 zeigt, wie die verschiedenen Abstraktionsebenen, Beschreibungsformen und Vorgehensweisen zusammenhängen und hierarchisch aufeinander aufbauen.

Hard- und Softwareunabhängigkeit des Systementwurfs

Die Beschreibung von Programmen, Komponenten oder Algorithmen unabhängig von einem bestimmten Hardwaresystem und einer konkreten Implementierung in Software, also ein sauberer Top-Down-Entwurf, ist wesentlich für die Wiederverwendbarkeit des Systementwurfs unter einer veränderten Hardwarekonfiguration oder -architektur. Imperative Beschreibungsformen sind dabei für komplexere Entwürfe zu unflexibel, da die von ihnen geforderte explizite Auf-

teilung eines Verfahrens in sequentielle und parallele Vorgänge den Entwurf an eine spezielle Hardware bindet sowie eine manuelle Planung der einzelnen Abläufe fordert.

Eine Alternative stellen funktionale Beschreibungsformen dar, da sie potentielle Parallelitäten von Algorithmen implizit enthalten. Abstrakte Datenobjekte und Funktionen ermöglichen eine Systembeschreibung, die unabhängig von einer konkreten Programmiersprache und Softwarebibliothek ist, was einen Wechsel des zugrundeliegenden Systems erleichtert. Dies ist das zentrale Thema dieses Kapitels. In den folgenden Abschnitten werden entsprechende Beschreibungsformen für dynamische Datenfolgen und die sie verarbeitenden Operatoren entwickelt, und es wird untersucht, in welcher Abstraktionsebene eine solche Beschreibung anzusiedeln ist.

Applikationsunabhängige Softwarebibliotheken

Mit der Implementierung von allgemeinen, weitestgehend anwendungsunabhängigen Verfahren und Datentypen bzw. -klassen und deren Zusammenstellen zu Softwarebibliotheken ist es möglich, bestimmte Basisfunktionalitäten, wie z.B. mathematische Algorithmen, Bildverarbeitungsoperatoren oder Operationen für Benutzerinteraktionen einem breiten Spektrum von Applikationen zur Verfügung zu stellen. Ziel eines solchen Bottom-Up-Entwurfs ist, die konkrete Implementierung der Verfahren zu kapseln und nur das Funktionsinterface nach außen zu geben. Dabei ist es möglich, plattformübergreifend einheitliche Schnittstellen zu den Verfahren bereitzustellen. Dies ermöglicht eine Implementierung von Programmkomponenten und Applikationen bereits auf einem relativ hohen Abstraktionsniveau sowie mit einer gewissen Hardwareunabhängigkeit.

In Kapitel 7 werden mit objektorientierten Methoden — unabhängig von einer konkreten Bildverarbeitungs- oder sonstigen anwendungsspezifischen Softwarebibliothek — Softwareobjekte für die Verarbeitung von dynamischen Datensequenzen wie z.B. Bildfolgen spezifiziert. Diese haben zum Ziel, eine einfache, direkte Umsetzung der funktionalen Repräsentationen dynamischer Zusammenhänge, wie sie in diesem Kapitel beschrieben werden, in eine objektorientierte Programmiersprache zu ermöglichen. Sie stellen damit eine Zwischenschicht zwischen dem funktionalen Entwurf und den Bibliotheken, die die konkreten Datentypen und Algorithmen implementieren, dar, wobei durch diese Objekte die Vorteile des funktionalen Systementwurfs in der Implementierung weitestgehend erhalten bleiben.

Standardisierte Entwurfsmethoden

Bei der Umsetzung der in der abstrakten Entwurfsebene beschriebenen Komponenten und Algorithmen in ein Programm innerhalb einer konkreten Entwicklungsumgebung aus Hard- und Software ergeben sich weitere Ansatzpunkte für die Wiederverwendung von Entwicklungsleistungen. Standardisierte Implementierungsschritte, sogenannte Entwurfsmuster, sowie Anwendungsrahmen repräsentieren typische Designentscheidungen, die unabhängig von der konkreten Anwendung und dem eingesetzten Softwaresystem angewendet werden können. Sie sollen helfen, einen Entwurf nach formalen Gesichtspunkten oder automatisch in ein Softwareprogramm zu übertragen.

6.2 Beschreibungskonzepte

6.2.1 Funktionale und imperative Beschreibungen

Imperative Sprachen

Imperative Sprachen beschreiben Algorithmen durch die Art ihrer Abarbeitung. Der Algorithmus wird dazu in sequentiell, also nacheinander abzuarbeitende Schritte zerlegt. Die Abarbeitung einzelner konkreter, vom Gesamtsystem weitestgehend unabhängiger Aufgaben läßt sich so auf eine relativ natürliche Weise beschreiben. Darüber hinaus sind imperative Programmiersprachen gut für die Programmierung von Einprozessorrechnern geeignet, da sie eine zur Arbeitsweise der Rechner adäquate Beschreibungsform darstellen.

An ihre Grenzen stoßen imperative Sprachen sowohl bei der Beschreibung potentieller Parallelitäten in Algorithmen als auch bei deren nebenläufigen Ausführung auf parallelen Hardwareplattformen. Mit Prozessen und Threads stehen zwar Mittel zur Verfügung, um den Algorithmen mehrere Zeitachsen für die parallele Abarbeitung einzelner Funktionssequenzen bereitzustellen, die möglichen Nebenläufigkeiten müssen jedoch zuvor aufgelöst und die Abfolge für die gegebene Hardware geplant und explizit spezifiziert werden. Darüber hinaus stellt die Synchronisation der Teilprozesse, die für jede Anwendung konkret bestimmt werden muß, keine triviale Aufgabe dar.

Da bereits in der Entwicklungsphase alle möglichen Funktionsabfolgen, Aufrufbedingungen und Nebenläufigkeiten spezifiziert werden müssen, sind imperative Sprachen relativ unflexibel gegenüber veränderten Systemanforderungen, dem Austausch von Komponenten, Erweiterungen des Systems oder wechselnden Ressourcenverfügbarkeiten. Um eine optimale Systemperformanz zu erreichen, erfordert jede der zuvor genannten Veränderungen die komplette Neuplanung der Funktionsabläufe und Ressourcenverteilungen, was in einem laufenden System i.d.R. nicht möglich ist.

Funktionale Sprachen

Eine funktionale Beschreibung besteht aus einer Menge von Datenobjekten und einer Menge von Funktionen, die Datenobjekte in Datenobjekte abbilden. Eine Funktion besitzt für jedes Eingangsdatum genau ein Eingangstor und für jedes Ausgangsdatum genau ein Ausgangstor. Weiterhin zeichnen sich funktionale Sprachen durch bestimmte Bildungsgesetze für die Verknüpfung von Funktionen und Datenobjekten aus. Repräsentieren läßt sich eine funktionale Beschreibung sowohl durch Gleichungen, also in Textform, als auch graphisch mit Hilfe von Datenflußnetzwerken. Beide Darstellungsformen sind äquivalent und können ineinander überführt werden.

Im Gegensatz zu imperativen Sprachen beschreiben funktionale Sprachen in erster Linie die Struktur von Algorithmen. Mögliche Parallelitäten sind hier implizit enthalten. Die Reihenfolge der Funktionsaufrufe muß beim Systementwurf nicht explizit festgelegt werden. Ob und wann eine Funktion aufgerufen wird, ergibt sich „lokal“ anhand der Zustände der direkt beteiligten Eingangs- und Ausgangsdaten der Funktionen.

Hinsichtlich der Datenlaufzeiten bzw. der Rechenzeiten von Funktionen können synchrone und asynchrone Datenflußmodelle unterschieden werden. Erstere gehen von der idealisierten Vorstellung aus, daß keine Verzögerungen im Datennetzwerk auftreten, sich die Daten also

mit unendlicher Geschwindigkeit ausbreiten. Der Zustand des Netzes ändert sich dabei nur synchron zu diskreten Zeittakten.

Asynchrone Datenflußmodelle beruhen dagegen auf der Vorstellung, daß Datenübertragung und -auswertung langsam sind, da sie u.a. Rechenzeit kosten. Das führt dazu, daß sich die Daten unabhängig voneinander, also asynchron, im Netz ausbreiten. Dies stellt das allgemeinere Modell dar, da es reale, parallel arbeitende Computersysteme besser modelliert als ein synchrones Datenflußmodell. Da sich in verschiedenen Teilnetzen die Zustände unabhängig voneinander ändern können, ist jedoch ein zusätzlicher Aufwand für die Synchronisation der Daten erforderlich. Diese muß jedoch nicht wie bei imperativen Programmzweigen für jede Anwendung neu bestimmt werden, sondern kann automatisch durch Anwendung allgemeiner Regeln erfolgen.

6.2.2 Grenzen rein funktionaler Beschreibungen

Für die vollständige Repräsentation komplexer Applikationen sind streng funktionale Beschreibungen nur bedingt geeignet. Das liegt zum einen daran, daß Programmsteueranweisungen wie Iterationen oder Verzweigungen, die in anderen Beschreibungsformen sehr einfach ausgedrückt werden können, mit rein funktionalen Mitteln nur recht umständlich zu formulieren sind.

Da in einer funktionalen Beschreibung jedes der m Eingangs- und n Ausgangstore einer Funktion mit genau einem Datenobjekt belegt sein muß, erfordern Funktionen mit einer variablen Anzahl an Ein- und Ausgangsdaten ebenfalls einen zusätzlichen Aufwand. In praktischen Anwendungen kommen solche Funktionen recht häufig vor, so kann es z.B. vorkommen, daß die Anzahl der aus einem Bild zu extrahierenden Merkmale schwankt oder Null wird. Eine Lösung ohne ein Verlassen der funktionalen Ebene wäre nur durch das Einfügen von Iteratorfunktionen in Verbindung mit Datenlisten möglich. Dabei haben die Iteratorfunktionen die Aufgabe, die Listenelemente nacheinander den sich anschließenden Operationen zur Verfügung zu stellen, was den Entwurf durch die Vermischung von Datenfluß- und Kontrollstrukturen jedoch deutlich verkompliziert.

Im Zusammenhang mit diesem Problem steht der Umgang mit vergangenen oder undefinierten Werten. So kommt es in der Praxis häufig vor, daß einzelne Objektmerkmale im aktuellen Bild nicht extrahiert und somit keine neuen Werte bestimmt werden können. Wie hier weiter zu verfahren ist, läßt sich nur schwer verallgemeinern und hängt von den Anforderungen der jeweiligen Anwendung ab. Eine Möglichkeit besteht beispielsweise darin, mit einer Ausnahmebehandlung zu reagieren, eine andere, daß nachfolgende Funktionen ihrerseits undefinierte Ausgabewerte erzeugen, oder daß sie in diesem Iterationsschritt ignoriert werden. Weiterhin ist es möglich, noch eine gewisse Zeit mit den alten Werten zu rechnen, wofür diese allerdings zur Verfügung stehen müssen. Rein funktionale Beschreibungen stellen für eine derart differenzierte Vorgehensweise keine Ausdrucksmittel zur Verfügung.

Ein weiteres Problem, das zusätzliche Anweisungen für die Programmsteuerung erfordert, ergibt sich, wenn bestimmte Funktionen unabhängig von der Datenrate der Eingangsdaten des Datennetzwerks ausgeführt werden sollen, beispielsweise um alle τ Sekunden einen Farbabgleich durchzuführen.

Programmsteuerkonstrukte bedeuten dabei immer einen gewissen Bruch mit der funktionalen Beschreibung. In den genannten Situationen werden einzelne Datenobjekte nach Änderung der Eingangsdaten des Datennetzwerks mehrfach oder gar nicht berechnet, das aber heißt, die Abbildung der Eingangsdaten auf die betroffenen Zwischenergebnisse ist nicht mehr eindeutig.

Schließlich besitzen rein funktionale Beschreibungsformen keine Ausdrucksmittel, um auf Zeitpunkt und Randbedingungen von Funktionsaufrufen direkt Einfluß zu nehmen. Damit fehlt ihnen eine wichtige Möglichkeit, bei der Umwandlung eines funktionalen Entwurfs in ein lauffähiges Programm eine Optimierung des Programmablaufs vorzunehmen.

6.2.3 Wahl einer geeigneten funktionalen Abstraktionsebene

Eine wichtige Voraussetzung für einen sauberen und effizienten Systementwurf ist eine klare Trennung der verschiedenen Entwurfsebenen sowie die Wahl der geeigneten Beschreibungsform für die jeweilige Ebene. In dieser Arbeit soll daher, wie es in Abb. 6.1 bereits gezeigt wurde, strikt zwischen einer abstrakten, systemunabhängigen, funktionalen Sicht auf eine Komponente und deren konkreten Implementierung in einem bestimmten Hard- und Softwaresystem unterschieden werden.

Repräsentation von Bildverarbeitungsalgorithmen

Im Gegensatz zu anderen Arbeiten, etwa zur Programmplanung für Echtzeit-Bildauswertegeräten [Mel95] oder zu graphischen Experimental- oder Prototypingsystemen wie KBVision [Wil90, Ame92], Khoros/Cantata [KR94], HCanvas [MG95] oder HI-VISUAL [YMH+86, HTI90], soll das Ziel hier nicht die detaillierte Beschreibung oder gar Programmierung ganzer Bildverarbeitungsalgorithmen mit funktionalen Mitteln auf der Grundlage vordefinierter Operatoren und Datentypen, also mit einer von der zugrundeliegenden Softwarebibliothek vorgegebenen Granularität sein.

Eine solche funktionale Programmierung einzelner mehr oder weniger komplexer Algorithmen ist, wie die zuvor genannten Ansätze zeigen, zwar möglich, wird jedoch von vielen Programmentwicklern als umständlich und wenig effizient empfunden. So wirkt z.B. die graphische Repräsentation von Programmsteueranweisungen wie Schleifen und Verzweigungen unnatürlich. Aufwendigere Algorithmen werden dabei schnell unübersichtlich und schwer handhabbar. Ein Wechsel der Beschreibungsform weg von den imperativen bzw. objektorientierten Sprachmitteln, mit denen etwa mathematische oder Bildverarbeitungsbibliotheken zur Verfügung stehen, hin zu einer funktionalen Sprache ist für solche relativ abgeschlossenen, eng umgrenzten Aufgaben i.d.R. nicht sinnvoll, da dem mit einem solchen Wechsel verbundenen Aufwand in einem realen Softwareprojekt nicht der entsprechende Nutzen gegenüber steht.

Aus diesen Gründen lassen sich einzelne konkrete Bildverarbeitungsaufgaben, wie z.B. im RoboCup die Selbstlokalisation anhand bestimmter Bildmerkmale oder die Extraktion des Balles aus farbigen Bildregionen, i.d.R. leichter und effizienter als in einem aufgesetzten graphischen System oder einer funktionalen Sprache mit den imperativen Mechanismen der Programmiersprache, in der auch die Bildverarbeitungsoperatoren bereitstehen, erstellen.

Abstraktionsgrad und Granularität der funktionalen Beschreibung

Unabhängig von der Umsetzung konkreter, weitestgehend abgeschlossener Teilaufgaben bietet sich die Modellierung dynamischer Programmaspekte und potentieller Aufgabenparallelitäten in einer eigenen funktionalen Beschreibungsebene an. Aufgabe dieser Ebene ist es, die für andere Module relevanten dynamischen Daten mit den zwischen ihnen bestehenden Abhängigkeiten

und möglichen Nebenläufigkeiten bei der Bereitstellung darzustellen sowie das Zusammenspiel der mit den verschiedenen Teilaufgaben beauftragten Funktionen und Programmkomponenten zu koordinieren. Die Granularität des Entwurfs wird nicht wie in anderen Systemen durch die verwendeten Softwarebibliotheken, also von außen vorgegeben, sondern folgt, wie Abb. 6.2 illustriert, in erster Linie den für diesen Abstraktionsgrad relevanten Entwurfskriterien der Anwendungsdomäne, wie z.B. Aufgabenüberschneidungen bei den verschiedenen Arbeitsschritten oder einer gewünschten Untergliederung der Modulaufgaben in kleinere, leichter handhabbare und unabhängige Teilaufgaben. Die Anzahl der in dieser Abstraktionsebene interessanten Daten und Funktionen bleibt dabei relativ klein und unabhängig von den verwendeten Bibliotheken. Die Funktionen stellen voneinander unabhängige Einheiten dar, die mit den Mitteln der darunterliegenden Ebenen definiert werden.

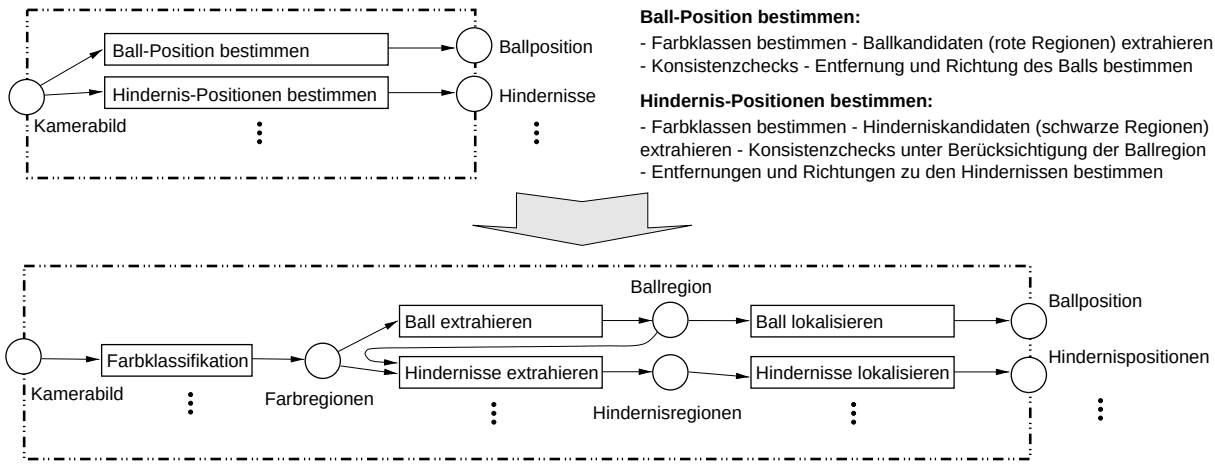


Abbildung 6.2: Gliederung eines Beispiel-Bildverarbeitungs-Moduls aus dem RoboCup in kleinere Teilaufgaben.

Diese strikte Trennung von Funktionalität und Programmstruktur ermöglicht einen sehr modularen Entwurf, was z.B. eine verteilte Projektentwicklung im Team unterstützt und ein hohes Maß an Flexibilität bei der Ausführung des Programmsystems mit sich bringt.

Ausdruck dieser Flexibilität ist beispielsweise der einfache Austausch von Funktionen durch alternative Verfahren. Da die einzelnen Funktionen vollkommen unabhängig voneinander arbeiten und über definierte Schnittstellen in das Gesamtsystem eingebunden werden, können sie, ohne daß die Programmsteuerung davon betroffen ist, leicht durch alternative Funktionen ersetzt werden. Auch eine Erweiterung des Systems um zusätzliche Aufgabenbereiche wird durch die Trennung von Funktionalität und Programmsteuerung sowie dem Fehlen einer explizit festgeschriebenen Ablaufsteuerung deutlich vereinfacht. Des weiteren ermöglicht dieser Ansatz die Bereitstellung von Daten in Abhängigkeit von der aktuellen Nachfrage durch andere Komponenten. Die Ausnutzung der vorhandenen Ressourcen wird optimiert, da nur die in einem Zyklus benötigten Daten berechnet werden müssen. Schließlich lassen sich in einem parallelen bzw. verteilten System die Aufgaben flexibel auf verschiedene Prozessoren oder Rechner aufteilen.

6.2.4 Automatische Ablaufsteuerung von Funktionsaufrufen

Auch wenn es ein wichtiges Ziel funktionaler Systembeschreibungen ist, keinen detaillierten Ablaufplan eines Programms erstellen zu müssen, stellt sich spätestens bei der Umsetzung eines Entwurfs auf einem realen Computersystem die Frage nach der Steuerung und Sequentialisierung der Funktionsaufrufe, also einer Festlegung, wie bestimmte funktional beschriebene Programmkomponenten ausgeführt werden sollen. Dabei müssen die folgenden Fragen beantwortet werden:

- Unter welchen Voraussetzungen *kann* bzw. *muß* eine Funktion aufgerufen werden?
- Wie sollen Funktionsaufrufe, die gleichzeitig erfolgen könnten, bei begrenzten Ressourcen sequentialisiert werden?
- Kann dieses Problem automatisch gelöst werden oder sind dafür spezielle Audrucksmittel notwendig?

In anderen Ansätzen sind i.d.R. keine speziellen Ausdrucksmittel vorgesehen, um die Reihenfolge der Abarbeitung zu beeinflussen. Folglich wird diese durch eine Planungskomponente im voraus, also im Rahmen der Implementierung festgelegt (z.B. [Mel95]), oder es wird versucht, automatisch zur Laufzeit durch Anwendung bestimmter Regeln das Problem zu lösen (z.B. [MG95]). Die zuerst genannten Planungsmodule wandeln die funktionale in eine imperative Beschreibung um und legen so die Aufrufreihenfolge fest. Sie sollen an dieser Stelle nicht weiter interessieren, da bei ihnen die funktionale Struktur verloren geht, und sie nicht flexibel auf Änderungen des laufenden Systems reagieren können.

Ausgangspunkt für Funktionsaufrufe in bestehenden funktionalen Softwarerepräsentationen sind i.d.R. die sich ändernden Eingangsdaten verbunden mit der Forderung nach einem konsistenten Datennetz. Das bedeutet, daß immer wenn die Eingangsdaten des Netzes sich ändern, alle nachfolgenden Daten ebenfalls aktualisiert werden müssen. Eine Inkonsistenz zwischen den Ein- und Ausgangsdaten einer Funktion stellt damit ein hinreichendes Kriterium für einen Funktionsaufruf dar, und alle Datenänderungen werden vorwärts durch das Netz propagiert.

Dabei wird implizit davon ausgegangen, daß die auf die Bereitstellung eines neuen Datums folgende Datenverarbeitung schnell genug ist und mit der Datenrate mithalten kann. Bei Funktionen mit mehr als einem Eingangsdatum ist darüber hinaus zu beachten, daß das automatische Bestimmen eines Aufrufzeitpunktes in Abhängigkeit von den sich ändernden Eingangsdaten nicht mehr trivial ist. So muß entschieden werden, ob ein Funktionsaufruf erfolgen soll,

- erst wenn *alle* Eingangsdaten einen neuen Wert haben,
- jedesmal wenn sich *ein* Eingangsdatums ändert oder
- nur wenn sich *bestimmte* Eingangsdaten ändern.

Zusätzliche Probleme bereiten in diesem Zusammenhang nichtsynchronisierte Sensoren, da sie an einer Funktion vollkommen unabhängig voneinander neue Eingangsdaten erzeugen können.

Weiterhin können bedingt durch die Netzarchitektur unterschiedliche Aufrufkonflikte auftreten, die lokal nicht oder nur unter Berücksichtigung zusätzlicher Statusinformationen der beteiligten Daten und Funktionen erkannt werden können. Daß es ohne eine Erweiterung des Ansatzes dafür keine Pauschallösung gibt, kann anhand der Abb. 6.3 leicht demonstriert werden.

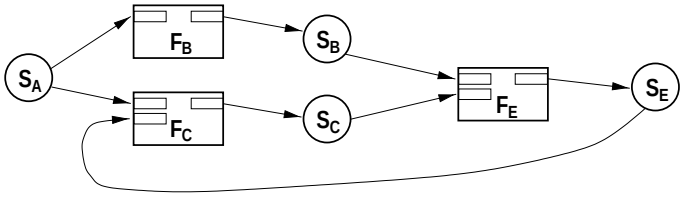


Abbildung 6.3: Datenflußgraph mit möglichen Aufrufkonflikten.

Erfolgt ein Funktionsaufruf nur, wenn alle Eingangsdaten neu sind, kommt es bei diesem Netzwerk zu Verklemmungen, die lokal — also nur anhand von Betrachtungen der Funktion und ihrer Ein- und Ausgangsdaten und ohne eine übergeordnete Planungskomponente, nicht erkannt werden können ($\mathbf{F_C}$ wartet auf $\mathbf{S_E}$, $\mathbf{S_E}$ wird aber nicht berechnet, solange $\mathbf{F_C}$ nicht $\mathbf{S_C}$ bereitstellt, damit $\mathbf{F_E}$ aufgerufen werden kann). Im zweiten Fall, also wenn bereits ein neues Eingangsdatum für einen Funktionsaufruf genügt, können Mehrfachberechnungen oder Endlosschleifen das System blockieren bzw. weitere Inkonsistenzen produzieren.

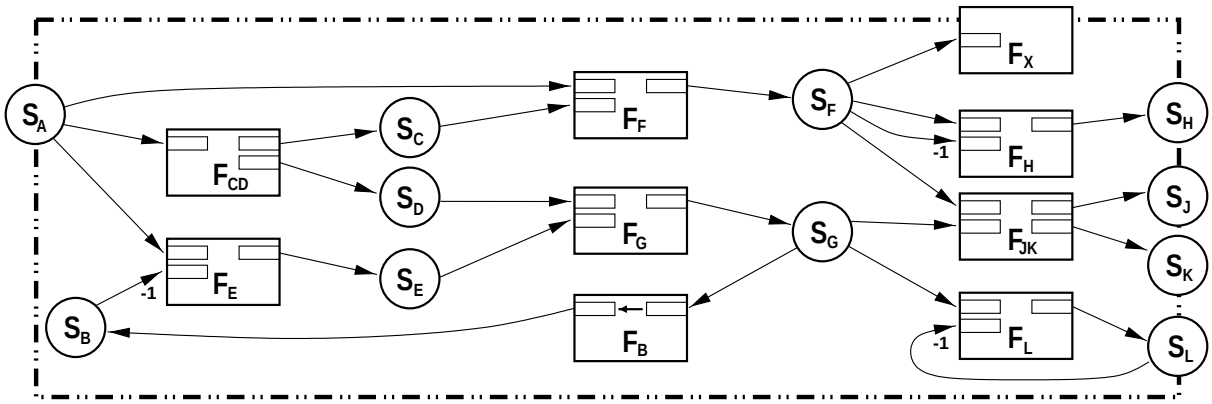
Zusammenfassend kann festgestellt werden, daß eine vollständige automatische Auflösung der Funktionsaufrufe nach einfachen, auf dem Datenfluß basierenden Regeln nur in relativ einfachen Systemen möglich bzw. praktikabel ist. Wann eine Funktion aufgerufen werden muß, kann nicht in jedem Fall anhand formaler Kriterien sondern häufig nur in Abhängigkeit von der konkreten Anwendung entschieden werden, wofür dem Entwickler entsprechende Einflußmöglichkeiten zur Verfügung stehen müssen.

6.3 Die funktionale Programmbeschreibung

6.3.1 Daten und Funktionen

Eine funktionale Beschreibung besteht aus Daten und Funktionen sowie den zwischen diesen Elementen bestehenden Beziehungen. Die konkreten Zusammenhänge lassen sich, wie Abb. 6.4 zeigt, durch ein Gleichungssystem oder mit Hilfe von Datenflußgraphen repräsentieren. In dieser Arbeit werden im Datenflußgraphen Datenobjekte als Kreise und Funktionsobjekte als Rechtecke mit den entsprechenden Eingangs- und Ausgangsdaten dargestellt. Dieses System zeigt zunächst einmal nur an, welche Eingangs- und Ausgangsdaten über Funktionen miteinander in Beziehung stehen, d.h. wie die Szeneninformationen über verschiedene Repräsentationsformen und Verarbeitungsstufen durch den Graphen „fließen“. An und für sich ist damit noch keine Vorschrift verbunden, wann in einem realen, sequentiell arbeitenden Computersystem Funktionen aufgerufen und Daten aktualisiert werden müssen. Der Graph beschreibt auch nicht, in welcher Weise die Daten modifiziert werden, d.h. wie die Funktionen intern arbeiten.

Als Datenobjekte sollen hier nur dynamische Datenfolgen interessieren, die kontinuierlich mit den Arbeitszyklen aktualisiert werden. Sie werden entsprechend Kapitel 4.2 explizit als dynamische Datensequenzen $\mathbf{S}$, also Zeitfolgen beliebiger Datenwerte oder -objekte modelliert. Das bedeutet, daß sie nicht nur den aktuellen Wert eines Datums sondern auch dessen Vergangenheit repräsentieren. Darüber hinaus werden Zeitinformationen für die Daten explizit modelliert (vgl. Kapitel 5.1).



$$\{\mathbf{S_C}, \mathbf{S_D}\} = \mathbf{F_{CD}}(\mathbf{S_A}) \qquad \{\mathbf{S_F}\} = \mathbf{F_F}(\mathbf{S_A}, \mathbf{S_C}) \qquad \{\ \} = \mathbf{F_X}(\mathbf{S_F})$$
$$\{\mathbf{S_E}\} = \mathbf{F_E}(\mathbf{S_A}, \mathbf{S_B^{(-1)}}) \qquad \{\mathbf{S_G}\} = \mathbf{F_G}(\mathbf{S_D}, \mathbf{S_E}) \qquad \{\mathbf{S_H}\} = \mathbf{F_H}(\mathbf{S_F^{(0,-1)}})$$
$$\{\mathbf{S_B}\} = \mathbf{F_B}(\mathbf{S_G}) \qquad \{\mathbf{S_J}, \mathbf{S_K}\} = \mathbf{F_{JK}}(\mathbf{S_F}, \mathbf{S_G})$$
$$\{\mathbf{S_L}\} = \mathbf{F_L}(\mathbf{S_G}, \mathbf{S_L^{(-1)}})$$

Abbildung 6.4: Datenflußgraph aus Funktoren $\mathbf{F_B} \ldots \mathbf{F_L}$ und Datensequenzen $\mathbf{S_A} \ldots \mathbf{S_L}$ — graphisch und als Gleichungssystem. Zugriffe auf alte Werte werden über einen Relativindex (z.B. -1) angezeigt. Voraussetzung dafür ist, daß die Datenobjekte alte Werte zwischenspeichern können.

Einfache Variablen oder Konstanten, die als steuernde Parameter in die Funktionen einfließen, spielen in diesem Rahmen nur eine untergeordnete Rolle und werden daher nicht gesondert untersucht. Variablen, die in jedem Zyklus neu berechnet werden, können wie Szenendaten als dynamische Datensequenzen repräsentiert werden, ändern sie sich dagegen nur gelegentlich, sind also relativ konstant im Vergleich zur Zyklusdauer, werden sie als integraler Bestandteil der Funktionen, die sie verwenden, interpretiert. Eine Änderung dieser Parameter entspricht dann, wie in Abschnitt 4.3.4 gezeigt wurde, einem Austausch der durch das Funktionsobjekt gekapselten Funktion.

Die zentrale Aufgabe der Funktionsobjekte ist es, Datenobjekte in Datenobjekte abzubilden. Sie werden entsprechend Kapitel 4.3 als Funktoren $\mathbf{F}$ modelliert und können beliebig komplexe Funktionen, Operatorfolgen oder ganze Programmkomponenten kapseln. Die so eingeschlossene Funktionalität kann sich von Aufruf zu Aufruf ändern, dabei bleiben jedoch Anzahl, Semantik und Typen der zu verknüpfenden Datenobjekte, die die Einordnung der Funktoren in den Datenflußgraphen, und damit den Datenfluß durch die Funktoren definieren, gleich.

Ein Funktionsaufruf stellt so eine Instanziierung der in der funktionalen Ebene dargestellten abstrakten Funktion durch einen bestimmten Algorithmus mit konkreten Parametern dar. Die Zeit ist einer dieser Parameter und fließt daher in jeden Funktionsaufruf mit ein. Legt man eine streng funktionale Betrachtungsweise zugrunde, erlaubt dies überhaupt erst eine Änderung der Funktionalität über die Zeit, ohne die für funktionale Beschreibungen geforderte Seiteneffektsfreiheit aufzugeben. Diese ist hier allerdings nur von marginalem Interesse, da nicht die Funktionen selbst Gegenstand der Untersuchungen sind, sondern das Zusammenspiel der Daten- und Funktionsobjekte und die sich daraus ergebenden Relationen. Diese bleiben aber bis zu einer Neukonfiguration des Programms konstant, auch wenn einzelne Parameter, Algorithmen oder Funktionen ausgetauscht werden.

6.3.2 Konsistenz des Datenflußgraphen

In rein funktionalen Beschreibungen wird davon ausgegangen, daß Datenobjekte immer genau einen aktuellen Wert haben, und sich die Werte zusammen mit den Eingangsdaten der Funktionen, an deren Ausgang die Daten anliegen, ändern. Das Datennetzwerk befindet sich dabei stets in einem konsistenten Gleichgewicht, welches durch das Gleichungssystem beschrieben wird. Dies stellt jedoch eine Idealisierung dar, die in realen Computersystemen aufgrund von Rechenzeiten und begrenzten Ressoucen so nicht aufrechterhalten werden kann. Während das Gleichungssystem im Idealfall stets die aus den bestehenden Abhängigkeiten unmittelbar folgenden Zustände aller Daten im System beschreibt, kann es hier nur als eine Anleitung verstanden werden, wie die Systemkonsistenz wieder hergestellt werden kann, nachdem sich einzelne Eingangsdaten verändert haben. In dem Zeitraum, in dem Funktionen noch damit beschäftigt sind, diesen Zustand herbeizuführen, befindet sich das Datennetzwerk in einem inkonsistenten Zustand, und Zugriffe auf die Datenobjekte liefern unter Umständen nicht zusammengehörige Daten.

Eine mögliche Lösung des Problems stellt ein synchrones Datenflußmodell dar. Dieses erlaubt den Zugriff auf die Daten nur synchron im Takt der Datenbereitstellung, also erst nach Erreichen eines konsistenten Zustands. Dieses Modell bringt jedoch einige Nachteile mit sich. So verbietet sich bei diesem Datenflußmodell von vornherein eine Parallelisierung nach dem Pipelining-Prinzip, welches ja gerade darauf beruht, daß in verschiedenen Netzkomponenten unterschiedlich alte Daten bearbeitet werden (vgl. Kapitel 1.4.3). Des weiteren ist hierbei die gemeinsame Verarbeitung nicht synchronisierter Eingangsdaten nur sehr eingeschränkt möglich. Damit in einem solchen System auf ältere Werte zugegriffen werden kann, müssen diese durch Verzögerungsfunktionen explizit bereitgestellt werden. Und schließlich müssen Module, die synchron auf berechnete Merkmale zugreifen wollen, immer solange auf die Daten warten, bis ein Berechnungszyklus vollständig abgeschlossen wurde, selbst wenn nur Daten verwendet werden, die im Zyklus bereits sehr früh bestimmt wurden.

In dem hier vorgestellten Ansatz soll daher die Asynchronität bei der Datenbereitstellung explizit berücksichtigt werden. Die Langsamkeit der Datenverbindungen stellt dabei jedoch keine Eigenschaft der funktionalen Beschreibung dar, sondern charakterisiert deren Umsetzung auf einem realen System. Das bedeutet, daß das (ideale) funktionale Gleichungssystem für die Beschreibung der Datenrelationen nach wie vor gültig ist, woraus folgt, daß bei sich ändernden Eingangsdaten des Datennetzes theoretisch auch für alle anderen Daten dieses Netzes neue Werte fällig werden. Diese sich nun offensichtlich ergebende Diskrepanz zwischen der Beschreibung des Systems durch das Gleichungssystem bzw. dem Datenflußgraphen und dessen Software-Realisierung, die sich in Form von Inkonsistenzen im Datennetz — neue Eingangsdaten produzieren per Definition sofort auch neue Ausgangsdaten, diese sind aber nicht sofort verfügbar — darstellt, wird dafür explizit modelliert. Sie stellt ein wichtiges Kriterium für die Steuerung von Funktionsaufrufen dar, braucht allerdings nur für die Daten aufgelöst zu werden, die in diesem Zyklus auch tatsächlich bestimmt werden sollen.

Eine zentrale Aufgabe dieses Ansatzes ist es daher, Dateninkonsistenzen erkennen und bei Bedarf auflösen zu können. Das wichtigste Hilfsmittel dafür sind die Zeitwerte, die die Daten beschreiben, insbesondere die Meß- oder Aufnahmezeit t_{g}. Daten mit der gleichen Datenzeit ($t_{\mathrm{g}}(S_X) = t_{\mathrm{g}}(S_Y)$) stehen zueinander entsprechend der aufgestellten Gleichungen in Beziehung, Zugriffe darauf sind also konsistent. Unterschiedliche Datenzeiten deuten dagegen auf

eine Inkonsistenz, und damit auf eine nicht vollständig besetzte Realisierung des Gleichungssystems hin. Wurde diese erkannt, kann die Konsistenz bei Bedarf durch entsprechende Funktionsaufrufe wieder hergestellt werden.

Eine weitere zentrale Zeitgröße stellt die Zykluszeit t_c dar. Mit diesem Zeitwert lassen sich Arbeitszyklen und die in einem Zyklus ermittelten Daten indizieren (vgl. Abschnitt 5.4), auch wenn die eigentliche Datenmeßzeit (noch) nicht bekannt ist. Dies ist insbesondere dann von Bedeutung, wenn es gilt, mögliche Aufrufkonflikte und Zugriffe auf ältere Werte zu erkennen.

Weichen die Daten- oder Zykluszeiten der von einem Funktor gemeinsam zu verarbeitenden Daten voneinander ab, ist dies i.d.R. auf einen inkonsistenten Zustand des Datennetzes zurückzuführen, der vor einem Aufruf der Funktoroperationen durch entsprechende Aktionen behoben werden muß. Dies ist jedoch nicht in jedem Fall notwendig, so z.B. wenn Daten unterschiedlicher, nicht synchronisierter Sensoren verknüpft oder wenn bewußt ältere Werte verwendet werden sollen. Auch gibt es verschiedene Möglichkeiten, auf Inkonsistenzen zu reagieren. So kann z.B. ein Funktor aktiv durch eigene Aktionen die Inkonsistenz zu beheben versuchen, er kann die Eingangsdatensequenz zu einer entsprechenden Aktion veranlassen, oder er kann mit dem letzten konsistenten Zustand arbeiten. Bei nichtsynchronen Eingangsdaten kann er die Daten, so wie sie eintreffen, verarbeiten oder mit Hilfe von Interpolations- oder Prädiktionsverfahren einen pseudokonsistenten Zustand herstellen, und schließlich kann der Funktor einfach warten, bis andere Instanzen im System die Konsistenz der Daten wieder hergestellt haben.

Welche Verfahren nun konkret zum Einsatz kommen sollen, ist u.U. eine Designentscheidung des Entwicklers und hängt von der Semantik der Daten und von den Beziehungen zwischen den Datenobjekten und Funktoren ab. Darüber hinaus spielt die Art und Weise, mit der das Datennetz gesteuert werden soll, eine wichtige Rolle. Mit Hilfe der im folgenden Abschnitt vorgestellten abstrakten Relationen zwischen Daten- und Funktionsobjekten lassen sich diese Beziehungen genauer spezifizieren. Durch bestimmte, vom zugrundeliegenden Steuerungskonzept abhängige Regeln werden diese Relationen dann automatisch in konkrete Zugriffsmethoden und Funktionsaufrufe umgesetzt.

6.3.3 Programmsteuerung auf der Basis von Relationen zwischen Daten- und Funktionsobjekten

Die aus einer funktionalen Sicht zentralen Beziehungen zwischen Daten und Funktionen ergeben sich aus dem Datenfluß, also der Verarbeitung von Daten mit Hilfe von Funktionen. Die Daten „fließen“ von einer Repräsentationsform zur nächsten, wobei sie in den Funktionen modifiziert werden. Der Datenfluß beschreibt die Struktur der Algorithmen, stellt jedoch keine eindeutige Anleitung für deren Abarbeitung dar. Auf mögliche Probleme bei der automatischen Umsetzung der strukturellen Beschreibung in konkrete Aufrufmethoden wurde bereits in den Abschnitten 6.2.4 und 6.3.2 eingegangen. Diese kann daher nur als eine unter gewissen Umständen anwendbare Standardvorgehensweise in Frage kommen. Daneben sind jedoch konkrete Ausdrucksmittel für eine Einflußnahme auf die Programmsteuerung notwendig.

Eine erste Möglichkeit — die direkte Einbeziehung von Funktionsaufrufen oder speziellen Zugriffsmethoden auf die Daten in die funktionale Beschreibung — kommt hier nicht in Frage, da dadurch die Systemunabhängigkeit und Flexibilität der funktionalen Beschreibung verloren gehen würde. Mit dem Festschreiben bestimmter Aufrufmethoden in der funktiona-

len Beschreibung wird bereits im Entwurf die Programmsteuerung entscheidend festgelegt, was mögliche Anpassungen an veränderte Randbedingungen zur Laufzeit verhindern würde.

Als Alternative soll hier der folgende Weg vorgeschlagen werden. Verschiedene Möglichkeiten der Einflußnahme auf die Programmsteuerung werden in einer eigenen Beschreibungsschicht explizit modelliert. Dazu werden die Datensequenzen $\mathbf{S}$ und Funktoren $\mathbf{F}$ als Objekte betrachtet, die definiert zueinander in Beziehungen gesetzt werden können. Diese Beziehungen, die im folgenden als Relationen bezeichnet werden, haben vor allem die Aufgabe, zwischen den Objekten eine mögliche Abgabe der Programmsteuerung zu regeln. Sie basieren auf dem Datenfluß, d.h. sie verbinden in erster Linie die im Datenflußgraphen benachbarten Daten und Funktoren. Relationen sind — unabhängig vom Datenfluß — immer in einem der beteiligten Objekte verankert, wodurch sich $\mathbf{S}\times\mathbf{F}$- und $\mathbf{F}\times\mathbf{S}$-Relationen ergeben. Mit ihrer Hilfe lassen sich bestimmte Zugriffs- und Aufrufmechanismen spezifizieren, die zur Laufzeit des Programms in konkrete Funktionsaufrufe umgesetzt werden, so daß die Konsistenz der funktionalen Beschreibung entsprechend den Erfordernissen aufrechterhalten und das Programm unter Ausnutzung der vorhandenen Ressourcen optimal gesteuert wird. Sie ermöglichen es somit, beim Programmentwurf von den konkreten Funktionsaufrufen zu abstrahieren.

Durch die Modellierung dieser Beziehungen in einer eigenen Ebene kann im Entwurf der Aspekt der Programmsteuerung vollkommen unabhängig von der Struktur der Algorithmen behandelt werden. Bei der Realisierung der Algorithmen sind die Objektrelationen, die auch in der Softwareebene explizit repräsentiert werden, mit bestimmten Aufruf- oder Methodenoptionen verbunden. Die Relationen beschreiben dabei den relativ statischen Teil einer Implementierung, wobei die verschiedenen Optionen immer noch einen differenzierten, kontextabhängigen Zugriff auf Datenobjekte und Funktoren erlauben. Ändern sich die Randbedingungen des Systems, z.B. durch neue parallel nutzbare Ressourcen oder Veränderungen bei bestimmten Teilaufgaben, gestattet die explizite Repräsentation der Relationen deren Neuplanung und damit eine Anpassung des Systems auch zur Laufzeit.

Welche konkreten Methoden letztendlich in der Applikation aufgerufen werden, legen die zwischen den Objekten bestehenden Relationen, der aktuelle Systemzustand sowie der Kontext der Objektzugiffe fest. Dabei kann durch die Wahl geeigneter Relationen das Spektrum möglicher Aufrufoptionen mehr oder weniger eingeschränkt werden. Durch die Definition von Defaultrelationen wird ein Standardverhalten vorgegeben, wodurch der Entwickler nur noch Ausnahmen explizit angeben muß.

Aufgrund der Lage der beteiligten Datensequenzen und Funktoren zueinander im Datenflußgraphen und der Verankerung der Relation können vier Klassen von Relationen unterschieden werden, der Pfeil gibt dabei die Datenflußrichtung an:

1. $\mathbf{F} \overset{\leftarrow}{\times} \mathbf{S}$: Zugriffe eines Funktors auf seine Eingangsdaten,
2. $\mathbf{F} \overset{\rightarrow}{\times} \mathbf{S}$: Setzen der Ausgangsdaten durch einen Funktor,
3. $\mathbf{S} \overset{\leftarrow}{\times} \mathbf{F}$: Funktoraufruf durch eine Datensequenz, um diese zu aktualisieren,
4. $\mathbf{S} \overset{\rightarrow}{\times} \mathbf{F}$: Funktoraufrufe von Datenobjekten, um neue Sequenzwerte zu propagieren.

Die ersten beiden Relationen spiegeln aus Sicht der Funktoren deren Beziehungen zu den Daten, auf die sie zugreifen, wider. Sie stellen damit *Funktor-Sequenz-Relationen* dar. Im Gegensatz dazu gehen die letzten beiden Relationen von einem Datenobjekt aus und beschreiben

dessen Verbindung zu Funktoren, die entweder einen neuen Datenwert erzeugen können oder einen neuen Datenwert weiterverarbeiten sollen. Hierbei handelt es sich somit um *Sequenz-Funktor-Relationen*.

Darüber hinaus lassen sich die Relationen anhand der Datenflußrichtung klassifizieren. So können sie einerseits *vorwärts*, d.h. mit dem Datenfluß gerichtet sein. Das betrifft das Setzen neuer Datenwerte durch einen Funktor (2.) sowie den Aufruf von Funktoren, die neue Datenwerte weiter verarbeiten (4.). Beide Relationen beschreiben Wege, um Datenänderungen im Graphen zu propagieren. Im Gegensatz dazu sind die beiden anderen Relationen — der Zugriff eines Funktors auf seine Eingangsdaten (1.) sowie der Aufruf eines Funktors für die Aktualisierung einer Datensequenz (3.) — *rückwärts* oder dem Datenfluß entgegengerichtet. Diese Relationen beschreiben den Weg zur Befriedigung von Datenanforderungen.

Agenten können — unabhängig vom Datenfluß — ebenfalls in Beziehung zu den Sequenzobjekten und Funktoren des Datenflußgraphen gesetzt werden. Um diese Beziehungen mit Hilfe der Relationen beschreiben zu können, treten die Agenten gegenüber Sequenzen wie ein Funktor und gegenüber Funktoren als Sequenz auf. Im folgenden sollen die Relationen näher spezifiziert werden.

6.3.4 Die Funktor-Eingangsdaten-Relation $\mathbf{F} \overset{\leftarrow}{\times} \mathbf{S} : (\mathtt{get})$

Das dieser Arbeit zugrundeliegende Datenflußmodell geht davon aus, daß jede Datenbewegung explizit auszuführen ist. Nach der Aktualisierung einer Sequenz liegt demzufolge der neue Sequenzwert nicht automatisch an den Eingangstoren der verbundenen Funktoren an, sondern der Funktor muß sich diese Daten zu Beginn seiner Ausführung explizit holen.

Die dafür notwendigen Zugriffe der Funktoren auf ihre Eingangsdaten können mit verschiedenen Anforderungen verbunden sein, wofür die Datensequenzen entsprechende Zugriffsmethoden zur Verfügung stellen (vgl. Abschnitt 4.2.2). In der funktionalen Ebene spiegelt sich das in Form unterschiedlicher Funktor-Sequenz-Relationen wider. So können Datenzugriffe mit oder ohne einen zeitlichen Bezug, d.h. der Forderung nach einer bestimmten Datenzeit, erfolgen, wobei als Zeitbezug eine Datenmeßzeit $t_{\mathtt{g}}$ oder die aktuelle Zykluszeit $t_{\mathtt{c}}$ in Frage kommt. Innerhalb einer solchen Relation kann auf den aktuellen oder auf ältere Werte (Relativindex $i < 0$) zugegriffen werden.

Spielt der Zeitwert bei Datenzugriffen eine Rolle, muß festgelegt werden, wie reagiert werden soll, falls für die angegebene Zeit keine Daten verfügbar sind. Möglich ist, auf die Daten zu warten, einen Fehler zu melden oder aus vorhandenen Datenwerten einen neuen Wert zu interpolieren bzw. zu prädizieren. Besitzt ein Funktor mehrere Eingangsdaten, auf deren Bereitstellung er warten muß, ist ein Zugriffsmodus zu bestimmen, mit dem festgelegt wird, ob die Datenwerte nacheinander bereitgestellt werden sollen — erst nachdem der Wert S_{X_i} vorliegt, wird der Wert $S_{X_{i+1}}$ angefordert — oder parallel — in einem ersten Schritt werden alle Werte angefordert, anschließend wird auf die Bereitstellung aller Daten, die parallel erfolgen kann, gemeinsam gewartet. Die Entscheidung darüber hängt von den zur Verfügung stehenden Ressourcen und möglichen Abhängigkeiten zwischen den Daten ab.

Ausgangspunkt bildet die Basisrelation $(\mathtt{get}) : \mathbf{F}\ (\mathtt{get})\ \mathbf{S}$. Diese Relation kennzeichnet lediglich, daß aufgrund des Datenflusses von den Datensequenzen $\mathbf{S_{X_j}}, 1 \leq j \leq m$ zum Funktor $\mathbf{F_{XY}} = \mathbf{F}(\mathbf{S_{X_1}}, \ldots, \mathbf{S_{X_m}})$ der Funktor $\mathbf{F_{XY}}$ auf ein Eingangsdatum $\mathbf{S_{X_j}}$ zugreift, ohne dabei die Zugriffsart genauer zu spezifizieren. Diese Relation wird verwendet, wenn in einer

Basisrelation:

$$\begin{array}{l}\mathbf{F}\ (\mathsf{get})\ \mathbf{S_A}\\ \mathbf{F}\ (\mathsf{get})\ \mathbf{S_B^{(-1)}}\end{array} \quad \Longleftrightarrow \quad \{\ldots\} = \mathbf{F}(\mathbf{S_A}, \mathbf{S_B^{(-1)}}) \quad \Longleftrightarrow$$

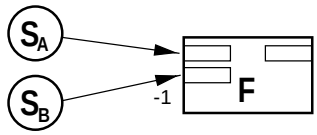


Abbildung 6.5: Basisrelation für den Datenzugriff (get).

bestimmten Entwurfsphase die Zugriffsart (noch) nicht genauer spezifiziert werden soll, oder falls die Wahl einer geeigneten Zugriffsmethode automatisch durch das System erfolgen soll. Entsprechend Abb. 6.5 wird sie im Datenflußdiagram durch den Datenflußpfeil vom Eingangsdatum zum Funktor und in der Gleichungsschreibweise durch $\mathbf{F}(\ldots, \mathbf{S}, \ldots)$ angezeigt. Alle drei Darstellungsformen (Relationenbeschreibung, Gleichung und Datenflußdiagramm) sind äquivalent. Mit $i : i < 0$ neben dem Pfeil bzw. $\mathbf{S}^{(i)}$ wird angegeben, daß der i-te Wert der Sequenz (bezogen auf den aktuellen Wert) als Eingangsdatum verwendet werden soll.

Im folgenden sollen weitere Zugriffsarten von (get) abgeleitet und modelliert werden. Dabei können verschiedene Aspekte des Datenzugriffs einzeln konkretisiert werden, wobei dies durch spezielle Symbole am Bezeichner der Zugriffsrelationen $(\mathsf{get}_{\mathsf{x}})$ gekennzeichnet wird. Im einzelnen sind die folgenden Zugriffsaspekte relevant:

- Zeitbezug: Mit der Zykluszeit t_{c} bzw. einer Datenmeßzeit t_{g} können Vorgaben an die zeitliche Einordnung der Daten gemacht werden. Fehlen diese, wird von der aktuellen Situation (t_{now}) ausgegangen;
- Verhalten bei zeitgebundenen Zugriffen, wenn die geforderten Daten (noch) fehlen: Aktualisierung einleiten und warten, Fehlermeldung oder Interpolation;
- Zugriffsmodus bei mehreren Eingangsdaten: sequentiell (synchron) oder parallel (asynchron).

Zeitbezug: Durch die Angabe von Zeitbezügen lassen sich verschiedene Aspekte des Datenzugriffs steuern. Die wichtigste Aufgabe ist die Synchronisation gemeinsam zu verarbeitender Eingangsdaten. Darüber hinaus dienen Zeitangaben dazu, Anforderungen hinsichtlich das Datenalters zu formulieren und ggf. aktiv umzusetzen, und schließlich lassen sich mit ihrer Hilfe einzelne Berechnungs- und Datenzyklen eindeutig indizieren.

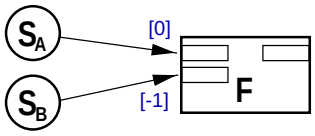
Zugriffe ohne Zeitbezug gehen vom jeweils aktuellen Zustand der Datensequenz aus und liefern ohne zu warten den letzten bzw. einen weiter zurück liegenden Wert. Die entsprechende Relation wird mit $(\mathsf{get}_{[]})$ bezeichnet und bietet sich für den Zugriff auf Daten, die kontinuierlich und unabhängig von ihrer Verarbeitung durch ein anderes Modul bereitgestellt werden, an. Wird innerhalb eines Verarbeitungszyklus allerdings mehrfach auf ein solches Datum zugegriffen, kann diese Relation nicht garantieren, daß immer die gleichen Datenwerte geliefert werden, da das Datum zwischen zwei Zugriffen von außen aktualisiert worden sein kann.

Sollen solche Mehrfachzugriffe auf ein unabhängig bereitgestelltes Datum möglich sein und innerhalb eines Zyklus immer den gleichen Wert liefern, kann dies durch die Zugriffsrelation $(\mathsf{get}_{[\mathsf{t}]})$, die die aktuelle Zykluszeit t_{c} als Referenzzeit t verwendet, erreicht werden. Wie $(\mathsf{get}_{[]})$ liefert diese Relation sofort einen Wert zurück, wobei allerdings nicht vom aktuellen Zustand der Sequenz sondern von dem zum Zeitpunkt t ausgegangen wird. Der Wert der als letztes vor t erzeugt wurde ($t_{\mathsf{s}} < t$) trägt somit den Index 0 beim Zugriff.

Für die Steuerung der Datenaktualisierung und die Synchronisation unterschiedlicher Daten innerhalb eines Zyklus ist ein stärkerer Zeitbezug als der der $(\mathsf{get}_{[\mathsf{t}]})$-Relation erforderlich. Dem wird eine Klasse von weiteren Zugriffsrelationen gerecht, an deren Spitze $(\mathsf{get}_{\mathsf{t}})$ steht. Diese fordern für die zu verarbeitenden Daten explizit eine bestimmte Daten- oder Zykluszeit (t_{g} bzw. t_{c}), wobei sie gegebenfalls auch auf die Bereitstellung der Daten warten oder sie aktiv herbeiführen. Ausgehend von diesen Daten sind mit negativen Indizes auch Zugriffe auf ältere Werte möglich. Die Darstellungsformen für die genannten Relationen faßt Abb. 6.6 zusammen.

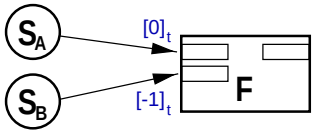
explizit ohne Zeitbezug:

$$\begin{array}{l}\mathbf{F}\ (\mathsf{get}_{[\,]})\ \mathbf{S_A}\\ \mathbf{F}\ (\mathsf{get}_{[\,]})\ \mathbf{S_B^{(-1)}}\end{array} \iff \{\ldots\} = \mathbf{F}(\mathbf{S_A^{[0]}}, \mathbf{S_B^{[-1]}}) \iff$$



mit Angabe der Zykluszeit als Referenzzeit:

$$\begin{array}{l}\mathbf{F}\ (\mathsf{get}_{[\mathsf{t}]})\ \mathbf{S_A}\\ \mathbf{F}\ (\mathsf{get}_{[\mathsf{t}]})\ \mathbf{S_B^{(-1)}}\end{array} \iff \{\ldots\} = \mathbf{F}(\mathbf{S_A^{[0]}}(t), \mathbf{S_B^{[-1]}}(t)) \iff$$



mit Zykluszeit oder Datenmeßzeit als Zeitbezug:

$$\begin{array}{l}\mathbf{F}\ (\mathsf{get}_{\mathsf{t}})\ \mathbf{S_A}\\ \mathbf{F}\ (\mathsf{get}_{\mathsf{t}})\ \mathbf{S_B^{(-1)}}\end{array} \iff \{\ldots\} = \mathbf{F}(\mathbf{S_A}(t), \mathbf{S_B^{(-1)}}(t)) \iff$$

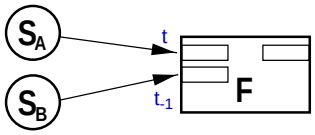


Abbildung 6.6: Darstellung des Zeitbezugs von Datenzugriffen.

Zeitliche Bezüge besitzen die folgende Semantik: t_{g} gibt den Zeitpunkt an, zu dem die Datenaufnahme bzw. -messung erfolgen soll bzw. erfolgt sein sollte. Damit wird sichergestellt, daß die Daten den Zustand des Systems zu *einem* konkreten Zeitpunkt beschreiben. Etwas schwächer ist die Angabe von t_{c}. Dies erfordert, daß die Daten aus dem gleichen Verarbeitungszyklus, also dem gleichen Iterationsschritt stammen. Dadurch wird i.d.R. auch sichergestellt, daß nur Daten eines Aufnahmezeitpunktes verarbeitet werden. Darüber hinaus ermöglicht es den konsistenten Datenzugriff auch dann, wenn die genaue Datenaufnahmezeit (noch) nicht bekannt ist. Als Zykluszeit wird typischerweise die Zykluszeit, mit der der Funktoraufruf erfolgte, verwendet. Die Datenmeßzeit kann dagegen beispielsweise von einen bereits ermittelten Eingangswert stammen:

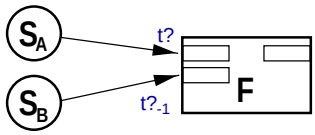
$$\begin{array}{lll}\text{Zykluszeit:} & \mathbf{F}\ (\ \mathbf{S_A}(t),\ \ldots\ ), & t = t_{\mathsf{c}}(\mathbf{F})\\ \text{Datenmeßzeit:} & \mathbf{F}\ (\ \mathbf{S_A^{[0]}}(),\ \mathbf{S_B}(t),\ \ldots\ ), & t = t_{\mathsf{g}}(S_A^0)\end{array}$$

Verhalten bei Zugriffen auf fehlende Daten: Als einen zweiten Aspekt des Datenzugriffs sollen spezielle Relationen es ermöglichen, Richtlinien für das Verhalten der Funktoren vorzugeben, falls bei einem zeitgebundenen Datenzugriff (noch) kein Datenwert für den angegebenen Zeitpunkt verfügbar ist. Die folgenden, in Abb. 6.7 zusammengefaßten Relationen stellen damit Spezialisierungen der $(\mathsf{get}_{\mathsf{t}})$-Relation dar.

Eine erste Möglichkeit besteht darin, Zugriffe, die durch die Datensequenz nicht unmittelbar befriedigt werden können, mit einer entsprechenden Statusmeldung abzubrechen. Der Funktor

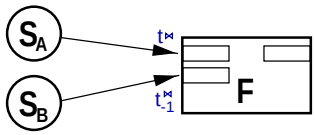
Test und ggf. Abbruch mit Statusinformation:

$$\begin{array}{l}\mathbf{F}\ (\mathrm{get}_t^{?})\ \mathbf{S_A}\\ \mathbf{F}\ (\mathrm{get}_t^{?})\ \mathbf{S_B^{(-1)}}\end{array} \iff \{\ldots\} = \mathbf{F}(\mathbf{S_A^{?}}(t), \mathbf{S_B^{(-1)\,?}}(t)) \iff$$



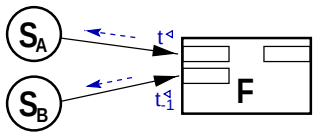
Warten auf Bereitstellung (passiv):

$$\begin{array}{l}\mathbf{F}\ (\mathrm{get}_t^{\bowtie})\ \mathbf{S_A}\\ \mathbf{F}\ (\mathrm{get}_t^{\bowtie})\ \mathbf{S_B^{(-1)}}\end{array} \iff \{\ldots\} = \mathbf{F}(\mathbf{S_A^{\bowtie}}(t), \mathbf{S_B^{(-1)\bowtie}}(t)) \iff$$



Wert insistent anfordern und auf Ergebnis warten:

$$\begin{array}{l}\mathbf{F}\ (\mathrm{get}_t^{\triangleleft})\ \mathbf{S_A}\\ \mathbf{F}\ (\mathrm{get}_t^{\triangleleft})\ \mathbf{S_B^{(-1)}}\end{array} \iff \{\ldots\} = \mathbf{F}(\mathbf{S_A^{\triangleleft}}(t), \mathbf{S_B^{(-1)\triangleleft}}(t)) \iff$$



Prädizieren oder Interpolieren, ggf. wartend oder insistent:

$$\begin{array}{l}\mathbf{F}\ (\mathrm{get}_t^{\sim})\ \mathbf{S_A}\\ \mathbf{F}\ (\mathrm{get}_t^{\widetilde{\bowtie}})\ \mathbf{S_B}\\ \mathbf{F}\ (\mathrm{get}_t^{\widetilde{\triangleleft}})\ \mathbf{S_C}\end{array} \iff \{\ldots\} = \mathbf{F}(\mathbf{S_A^{\sim}}(t), \mathbf{S_B^{\widetilde{\bowtie}}}(t), \mathbf{S_C^{\widetilde{\triangleleft}}}(t)) \iff$$

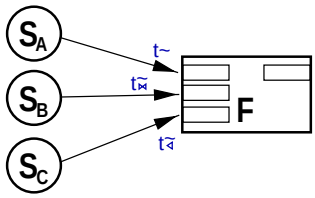


Abbildung 6.7: Darstellungsmittel zur Bestimmung des Verhaltens bei Zugriffen auf fehlende Daten.

kann diese auswerten und daraufhin z.B. seine Ausführung ebenfalls ohne Ergebnis beenden. Gekennzeichnet wird dieses Verhalten durch die Relation $(\mathrm{get}_t^{?})$. Diese Zugriffsart impliziert somit einen Test, ob das geforderte Datum (bereits) verfügbar ist. Schlägt der Test für mindestens ein Eingangsdatum fehl, muß der Funktor zu einem späteren Zeitpunkt — z.B. wenn ein weiteres Eingangsdatum bereitsteht — erneut angestoßen werden, damit er in diesem Zyklus den Aufrufversuch wiederholen kann.

Alternativ zu dieser Möglichkeit wird mit der Relation $(\mathrm{get}_t^{\triangleleft})$ ausgedrückt, daß beim Zugriff auf den angeforderten Wert bestanden wird, d.h. der Zugriff *insistent* ist. Dafür wird die Programmkontrolle an die Sequenz übertragen und auf ein Ergebnis gewartet. Damit steht ein Mittel zur Verfügung, Datenanfragen durch das Netz bis zu den Eingangsdaten des Moduls durchzureichen und sukzessive abzuarbeiten. Wurde über einen beliebigen Anfrageweg im Graphen ein Datum einmal berechnet, steht es auch allen anderen Anfragen an die Sequenz zur Verfügung. Es kann über die Daten- oder Zykluszeit eindeutig identifiziert und dann ohne Verzögerung an den Funktor übergeben werden. Damit wird sichergestellt, daß jedes Datum pro Zyklus nur einmal berechnet und jeder Funktor nur einmal ausgeführt wird.

Werden bestimmte Eingangsdaten durch eine vom betrachteten Funktor unabhängige Komponente bereitgestellt, ist es unter Umständen erforderlich, beim Zugriff auf diese Daten einfach nur zu warten, bis die Sequenz aktualisiert wurde. Dies wird mit Hilfe der $(\mathrm{get}_t^{\bowtie})$-Relation spezifiziert.

Und schließlich kann eine Interpolation oder Prädiktion der fehlenden Daten aus den in der Sequenz gespeicherten Werten erforderlich sein, was mit Hilfe der Relation $(\mathrm{get}_t^{\sim})$ ausgedrückt wird. Soll dabei auf den ersten Meßwert *nach* dem angegebene Zeitpunkt gewartet werden —

wird also eine echte Interpolation und keine Prädiktion gefordert, ist die Relation $(\mathsf{get}_{\mathsf{t}}^{\widetilde{\lhd}})$ zu verwenden, und soll der neue Wert aktiv durch die Sequenz bereitgestellt werden, kann dies mit der insistenten Interpolationsrelation $(\mathsf{get}_{\mathsf{t}}^{\widetilde{\lhd}})$ ausgedrückt werden.

Die Interpolation von Datenwerten ist insbesondere im Zusammenhang mit unabhängigen, nicht synchronisierten Sensoren von Bedeutung. Mit ihrer Hilfe ist es trotz unterschiedlicher Meßzeiten möglich, mit weitestgehend konsistenten Daten zu rechnen. Dabei kann es sich, wie im RoboCup, z.B. um Kamera- und Odometriedaten als Daten unabhängiger Sensoren handeln. Durch die speziellen Zugriffsmethoden läßt sich einfach beschreiben, welche Daten wie anzupassen sind:

$$\mathbf{F}(\mathbf{S_{Cam}}, \mathbf{S_{Odo}}) = \mathbf{F}\ \Big(\ \underbrace{\mathbf{S}^{\lhd}_{\mathbf{Cam}}(t = t_{\mathsf{c}}(\mathbf{F}))}_{\substack{\text{Warten auf}\\ \text{Bilddaten}}},\ \underbrace{\mathbf{S}^{\widetilde{}}_{\mathbf{Odo}}(t = t_{\mathsf{g}}(S_A(t_{\mathsf{c}})))}_{\substack{\text{Interpolation der}\\ \text{Odometriedaten}}},\ \ldots\ \Big)$$

Bereitstellungsmodus bei mehreren Eingangsdaten: Als letztes sollen hier verschiedene Zugriffsverfahren für Funktoren, die mehr als ein Eingangsdatum besitzen, untersucht werden. Muß auf die Berechnung oder Messung mehrerer Daten gewartet werden, können prinzipiell zwei Wege, die Daten anzufordern, unterschieden werden. So ist es zum einen möglich, die Daten sequentiell, d.h. nacheinander anzufordern. Die Programmsteuerung wird dafür nacheinander an die entsprechenden Eingangsdatensequenzen übergeben. Sie kehrt nach Bereitstellung eines angeforderten Datenwertes zurück zum Funktor, von wo aus sie an das nächste Sequenzobjekt übergeben wird.

Alternativ dazu können Daten parallel bzw. asynchron bereitgestellt werden. In diesem Fall läuft die Anfrage in zwei Schritten ab. Zuerst wird von allen Datensequenzen der entsprechende Wert angefordert, wobei jedes Sequenzobjekt einen eigenen *Thread*, der parallel zu den anderen arbeitet, erhält. Der Funktor wartet anschließend darauf, daß die Daten nacheinander eintreffen. Liegen alle Daten vor, kann mit der eigentlichen Funktoroperation begonnen werden.

Diese Unterscheidung ist nur für aktive Datenanforderungen sinnvoll, also für $(\mathsf{get}_{\mathsf{t}}^{\lhd})$, $(\mathsf{get}_{\mathsf{t}}^{\widetilde{}})$ und $(\mathsf{get}_{\mathsf{t}}^{\widetilde{\lhd}})$. Symbolisiert wird der sequentielle Anfragemodus durch das Symbol '$\circ$', die Kennzeichnung des parallelen oder asynchronen Anfragemodus erfolgt mit dem Symbol '$\|$'. Abb. 6.8 faßt die Darstellungsformen zusammen.

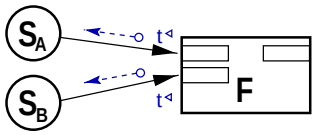
Der Vorteil der asynchronen Datenbereitstellung liegt in der möglichen Ausnutzung paralleler Ressourcen, d.h. die Datenbereitstellungsphase des Funktoraufrufs kann sich deutlich verkürzen. Als Nachteil stehen dem, wie in Abschnitt 6.4.6 genauer untersucht wird, mögliche Probleme beim Erkennen von Fehlern im Design des Datennetzes und ein erhöhter Synchronisationsaufwand gegenüber. Für den Entwickler ist dies allerdings nicht von Bedeutung, da die Synchronisation von den Sequenz- und Funktorobjekten vorgenommen wird.

6.3.5 Die Funktor-Ausgangsdaten-Relation $\mathbf{F} \overrightarrow{\times} \mathbf{S} : (\mathsf{set})$

Die Beziehungen zwischen einem Funktor $\mathbf{F}$ und seinen Ausgangsdaten $\{\mathbf{S_{Out}}\}$ ergeben sich direkt aus dem Datenfluß. Mit jeder Ausführung des Funktors, die in der Vorbereitungsphase $\mathsf{pre}()$ nicht abgebrochen wurde, werden dessen Ausgangsdaten neu bestimmt und unmittelbar nach Beendigung der Funktoroperationen in den jeweiligen Sequenzen explizit gesetzt. Das bedeutet, daß jeder Ausgangsdatensequenz ein neues Element S_{out_i} zugewiesen wird. Konnte

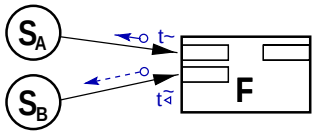
Datenbereitstellung nacheinander:

$$\begin{array}{l}\mathbf{F}\ \ (\mathrm{get}^{\triangleleft}_{\mathrm{to}})\ \mathbf{S_A}\\ \mathbf{F}\ \ (\mathrm{get}^{\triangleleft}_{\mathrm{to}})\ \mathbf{S_B}\end{array} \iff \{\ldots\} = \mathbf{F}(\mathbf{S}^{\triangleleft}_{\mathbf{A}}(t) \circ \mathbf{S}^{\triangleleft}_{\mathbf{B}}(t)\circ) \iff$$



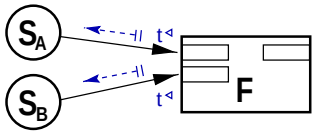
Interpolation nacheinander:

$$\begin{array}{l}\mathbf{F}\ \ (\mathrm{get}^{\sim}_{\mathrm{to}})\ \mathbf{S_A}\\ \mathbf{F}\ \ (\mathrm{get}^{\widetilde{\triangleleft}}_{\mathrm{to}})\ \mathbf{S_B}\end{array} \iff \{\ldots\} = \mathbf{F}(\mathbf{S}^{\sim}_{\mathbf{A}}(t) \circ \mathbf{S}^{\widetilde{\triangleleft}}_{\mathbf{B}}(t)\circ) \iff$$



Datenbereitstellung parallel:

$$\begin{array}{l}\mathbf{F}\ \ (\mathrm{get}^{\triangleleft}_{\mathrm{t}\|})\ \mathbf{S_A}\\ \mathbf{F}\ \ (\mathrm{get}^{\triangleleft}_{\mathrm{t}\|})\ \mathbf{S_B}\end{array} \iff \{\ldots\} = \mathbf{F}(\mathbf{S}^{\triangleleft}_{\mathbf{A}}(t) \parallel \mathbf{S}^{\triangleleft}_{\mathbf{B}}(t) \|) \iff$$



Interpolation parallel:

$$\begin{array}{l}\mathbf{F}\ \ (\mathrm{get}^{\sim}_{\mathrm{t}\|})\ \mathbf{S_A}\\ \mathbf{F}\ \ (\mathrm{get}^{\widetilde{\triangleleft}}_{\mathrm{t}\|})\ \mathbf{S_B}\end{array} \iff \{\ldots\} = \mathbf{F}(\mathbf{S}^{\sim}_{\mathbf{A}}(t) \parallel \mathbf{S}^{\widetilde{\triangleleft}}_{\mathbf{B}}(t) \|) \iff$$

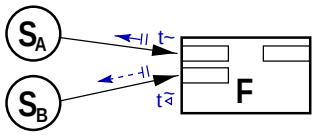


Abbildung 6.8: Darstellungsmittel zur Beschreibung von sequentiellen und parallelen Zugriffsmodi.

für eine Sequenz $\mathbf{S_{Out_i}}$ aufgrund der gemessenen Eingangsdaten kein neuer Wert bestimmt werden, wird durch ein entsprechendes Element $S_{out_i} = (V = \emptyset, \ldots)$ die Sequenz für den aktuellen Aufrufzyklus explizit als '*nicht definiert*' markiert.

Die entsprechende Basisrelation wird mit $(\mathsf{set})\colon\ \mathbf{F}\ (\mathsf{set})\ \mathbf{S}$ bezeichnet. Sie ist mit dem Datenfluß gleichgerichtet und fest an ihn gekoppelt, so daß sie nicht explizit angegeben werden muß. Abb. 6.9 zeigt die entsprechende Darstellung.

Basisrelation:

$$\begin{array}{l}\mathbf{F}\ \ (\mathsf{set})\ \mathbf{S_A}\\ \mathbf{F}\ \ (\mathsf{set})\ \mathbf{S_B}\end{array} \iff \{\mathbf{S_A}, \mathbf{S_B}\} = \mathbf{F}(\ldots) \iff$$

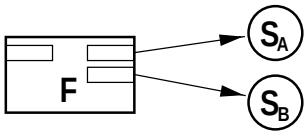


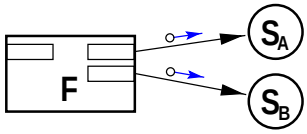
Abbildung 6.9: Implizit gegebene Basisrelationen (set) zwischen Funktoren und ihren Ausgangsdaten.

Da die Aktualisierung einer Datensequenz eine Reihe von Aktionen nach sich ziehen kann, wie z.B. das Triggern und Abarbeiten weiterer, von der Sequenz abhängigen und mit der entsprechenden Sequenz-Funktor-Relation (vgl. Abschnitt 6.3.7) gekennzeichneten Folgefunktoren oder das Propagieren der neuen Daten in einem Rechnernetz, ist es auch hier sinnvoll, zwischen synchronem (d.h. sequentiellem) und asynchronem (bzw. parallelem) Setzen der Datenwerte zu unterscheiden. Die entsprechenden Relationen zeigt Abb. 6.10. Im synchronen Fall werden die Datensequenzen nacheinander aktualisiert, d.h. erst nachdem eine zu aktualisierende Sequenz die Kontrolle an den Funktor zurückgegeben hat, wird die Programmsteuerung an

die nächste Sequenz weiter gereicht. Das bedeutet auch, daß erst nach Abschluß der Aktualisierung der letzten Sequenz der Funktoraufruf beendet wird. Die entsprechende Relation wird mit $(\mathsf{set}_\circ) : \mathbf{F}\ (\mathsf{set}_\circ)\ \mathbf{S}$ bezeichnet.

Sequentielle (synchrone) Aktualisierung:

$$\begin{array}{l}\mathbf{F}\ \ (\mathsf{set}_\circ)\ \ \mathbf{S_A}\\ \mathbf{F}\ \ (\mathsf{set}_\circ)\ \ \mathbf{S_B}\end{array} \quad \Longleftrightarrow \quad \{\mathbf{S_A} \circ \mathbf{S_B} \circ\} = \mathbf{F}(\ldots) \quad \Longleftrightarrow$$



Parallele (asynchrone) Aktualisierung:

$$\begin{array}{l}\mathbf{F}\ \ (\mathsf{set}_{\|})\ \ \mathbf{S_A}\\ \mathbf{F}\ \ (\mathsf{set}_{\|})\ \ \mathbf{S_B}\end{array} \quad \Longleftrightarrow \quad \{\mathbf{S_A} \parallel \mathbf{S_B} \parallel\} = \mathbf{F}(\ldots) \quad \Longleftrightarrow$$

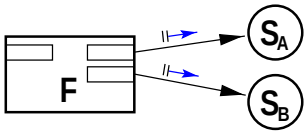


Abbildung 6.10: Relationen zur Spezifizierung des synchronen oder asynchronen Setzens der Ausgangsdaten eines Funktors.

Im Gegensatz dazu erfolgt die asynchrone Aktualisierung einer Ausgangsdatensequenz in einem eigenen *Thread*, so daß der Funktor die Kontrolle behält. Damit werden alle Sequenzen parallel aktualisiert und noch bevor die Aktualisierung vollständig abgeschlossen ist, wird die Aufrufmethode des Funktors verlassen. Wird dieses Verhalten gefordert, kann dies durch die Relation $(\mathsf{set}_{\|}) : \mathbf{F}\ (\mathsf{set}_{\|})\ \mathbf{S}$ ausgedrückt werden. Auch hier gilt, daß sich damit u.U. parallele Ressourcen besser ausnutzen lassen, andererseits aber ein erhöhter Synchronisationsaufwand erforderlich ist, da sich die Daten asynchron im Graphen ausbreiten.

6.3.6 Die Sequenz-Aktualisierungsfunktor-Relation $\mathbf{S} \overset{\leftarrow}{\times} \mathbf{F} : (\mathsf{upd})$

Wie die zuvor beschriebene Funktor-Ausgangsdatum-Relation basiert die folgende Relation zwischen einem Datenobjekt und dem dazugehörigen Aktualisierungsfunktor auf dem Datenfluß von einem Funktor $\mathbf{F}$ zu der von ihm aktualisierten Datensequenz $\mathbf{S}$. Anders als (set) geht diese Relation jedoch von der Sequenz $\mathbf{S}$ aus und ist damit dem Datenfluß entgegengerichtet. Sie wird dem Umstand gerecht, daß ein Datenobjekt von genau einem Funktor $\mathbf{F_{upd}}$ aktualisiert wird. Durch das Setzen dieser Relation, die mit $(\mathsf{upd}) : \mathbf{S}\ (\mathsf{upd})\ \mathbf{F}$ bezeichnet und in Abb. 6.11 dargestellt wird, erfolgt die Zuordnung zwischen einem konkreten Funktor und der sequenzinternen Größe $\mathbf{F_{upd}}$.

$$\begin{array}{l}\mathbf{S_A}\ \ (\mathsf{upd})\ \ \mathbf{F}\\ \mathbf{S_C}\ \ (\mathsf{upd})\ \ \mathbf{F}\end{array} \quad \Longleftrightarrow \quad \{\vec{\mathbf{S_A}}, \mathbf{S_B}, \vec{\mathbf{S_C}}\} = \mathbf{F}(\ldots) \Longleftrightarrow$$

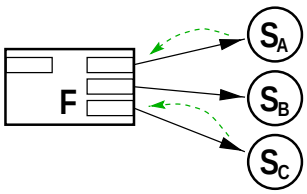


Abbildung 6.11: Relation (upd) zur Kennzeichnung der Beziehung eines Datenobjekts zu dem dazugehörigen Aktualisierungsfunktor.

Durch das Setzen dieser Relation wird ein Sequenzobjekt in die Lage versetzt, insistente zeitgebundene Datenzugriffe, wie sie die Funktor-Eingangsdaten-Relationen $(\mathsf{get}_t^{\triangleleft})$, $(\mathsf{get}_t^{\tilde{\triangleleft}})$ fordern, aktiv durch Aufruf des Aktualisierungsfunktors $\mathbf{F_{upd}}$ zu befriedigen. Wird diese Relation, wie das im Beispiel für die Sequenz $\mathbf{S_B}$ der Fall ist, nicht gesetzt: $\mathbf{S_B}\ \cancel{(\mathsf{upd})}\ \mathbf{F}$, kennt die Sequenz den entsprechenden Funktor nicht und ist somit darauf angewiesen, daß der Funktor durch eine andere Instanz aufgerufen wird.

Obwohl sich für diese Relation nicht direkt die Frage nach einer Parallelausführung stellt — eine Sequenz hat nur einen Aktualisierungsfunktor und muß auf dessen Ergebnis warten — soll auch hier zwischen einer synchronen $(\mathsf{upd}_\circ)$ und einer asynchronen Variante $(\mathsf{upd}_{||})$ unterschieden werden. Beide Varianten werden in Abb. 6.12 dargestellt. Sie unterscheiden sich vor allem hinsichtlich des Synchronisationspunktes für den Datenzugriff, von dem die Aktualisierung der Sequenz ausging. Während der durch $(\mathsf{upd}_\circ)$ vorgegebene synchrone Funktoraufruf darauf wartet, von dem Funktor $\mathbf{F}$ die Kontrolle zurück zu bekommen, was in jedem Fall nach dem Setzen dessen Ausgangsdaten geschieht, teilt sich beim asynchronen Funktoraufruf (gesetzte $(\mathsf{upd}_{||})$-Relation) der Kontrollfluß auf, und die Steuerung kehrt sofort zur Sequenz zurück. Die Verantwortung für die Synchronisation mit den zu erwartenden Daten liegt nun in der Zugriffsmethode, die auf das Setzen der Daten wartet — analog zum wartenden Zugriff mit $\mathsf{get}_t^{\bowtie}$. Da der Funktor seine Ausgangsdaten setzt, bevor er die Steuerung zurückgibt, bedeutet dies für die Daten in der Abb. 6.12, daß — bei in etwa gleichzeitigen Datenzugriffen auf $\mathbf{S_A}$ und $\mathbf{S_C}$ — letztere aufgrund der asynchronen $(\mathsf{upd}_{||})$-Relation *zuerst* ihren Wert weitergeben kann; und zwar nach dem Setzen von S_C und nicht erst nach der Beendigung des Funktoraufrufs.

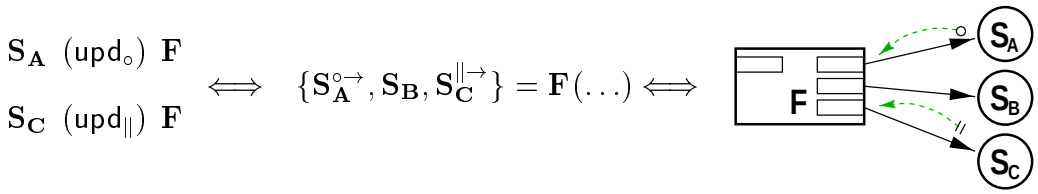


Abbildung 6.12: Synchrone und asynchrone Relationen zwischen einem Datenobjekt und dem entsprechenden Aktualisierungsfunktor.

6.3.7 Die Sequenz-Folgefunktor-Relation $\mathbf{S} \vec{\times} \mathbf{F} : (\mathsf{trig})$

Die letzte Relationsklasse für die Charakterisierung der Beziehungen zwischen Daten- und Funktionsobjekten bietet die Möglichkeit, Funktoren durch ihre Eingangsdaten triggern zu lassen und so ihre Ausführung direkt an die Aktualisierung der Datensequenzen zu koppeln. Dies dient — unmittelbar aus dem Datenfluß folgend — dazu, nach jeder Sequenzaktualisierung die im Datenflußgraphen folgenden Funktoren abzuarbeiten.

Das Bestehen einer derartigen Kopplung wird durch die Basisrelation $(\mathsf{trig}) : \mathbf{S}\ (\mathsf{trig})\ \mathbf{F}$ (s. Abb. 6.13) ausgedrückt. Im Gegensatz zur (get)- oder zur (set)-Relation folgt eine solche Kopplung nicht zwingend aus dem Datenfluß. Durch das Setzen der (trig)-Relation wird der Funktor $\mathbf{F}$ in die Liste der Folgefunktoren der Sequenz $\mathbf{S}$ eingetragen: $\mathbf{F} \in \{\mathbf{F_{dep}}\}$. Auf das Setzen der Relation zwischen Daten und den sie verarbeitenden Funktoren kann verzichtet

werden, wenn der Funktoraufruf von einer anderen Seite, beispielsweise bei Anfragen an dessen Ausgangsdaten erfolgt.

$$\begin{array}{l}\mathbf{S_A}\ (\mathsf{trig})\ \mathbf{F_{A1}}\\ \mathbf{S_A}\ (\mathsf{trig})\ \mathbf{F_{A2}}\\ \mathbf{S_B}\ (\mathsf{trig})\ \mathbf{F_{A2}}\\ \mathbf{S_B}\ (\mathsf{trig})\ \mathbf{F_B}\end{array} \iff \begin{array}{l}\{\ldots\} = \mathbf{F_{A1}}(\ \ldots, \mathbf{S_A}\ )_{\leftarrow \mathbf{S_A}}\\ \{\ldots\} = \mathbf{F_{A2}}(\ \mathbf{S_B}, \mathbf{S_A}\ )_{\leftarrow \mathbf{S_A}, \mathbf{S_B}}\\ \{\ldots\} = \mathbf{F_B}(\ \mathbf{S_B}\ )_{\leftarrow \mathbf{S_A}}\end{array} \iff$$

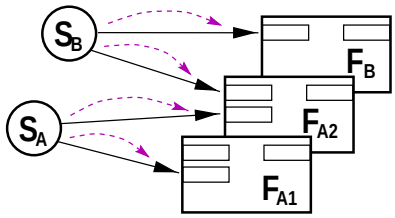
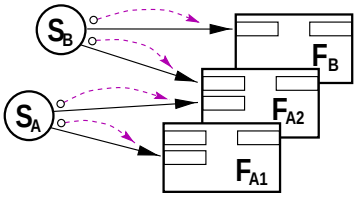


Abbildung 6.13: Kopplung von Funktoren an Datenobjekte mit der Relation (trig).

Ausgehend von dieser Basisrelation läßt sich der Aufruf- bzw. Ausführungsmodus der so getriggerten Funktoren konkretisieren. Zum einen können die Funktoraufrufe explizit nacheinander erfolgen, wobei auf die Beendigung des einen Funktors gewartet wird, bevor der nächste aufgerufen wird. Die Aktualisierung des Datenobjektes $\mathbf{S_A}$ ist damit auch erst nach Ausführung der letzten Folgefunktors vollständig abgeschlossen. Ausgedrückt wird dieses Verhalten durch die Relation $(\mathsf{trig}_\circ)$. Die Relation $(\mathsf{trig}_{||})$ bewirkt dagegen, daß, wenn einer Sequenz ein neuer Wert hinzugefügt wurde, alle Folgefunktoren angestoßen, dann aber in einem eigenen Programmzweig ausgeführt werden. Unmittelbar nach dem Triggern der Funktoren wird die Datenaktualisierung beendet. Besitzt ein Datenobjekt mehrere zu triggernde Folgefunktoren, werden diese im ersten Modus sequentiell und im zweiten parallel abgearbeitet. Abb. 6.14 stellt diese beiden Relation einander gegenüber.

Sequentielle Abarbeitung:

$$\begin{array}{l}\mathbf{S_A}\ (\mathsf{trig}_\circ)\ \mathbf{F_{A1}}\\ \mathbf{S_A}\ (\mathsf{trig}_\circ)\ \mathbf{F_{A2}}\\ \mathbf{S_B}\ (\mathsf{trig}_\circ)\ \mathbf{F_{A2}}\\ \mathbf{S_B}\ (\mathsf{trig}_\circ)\ \mathbf{F_B}\end{array} \iff \begin{array}{l}\{\ldots\} = \mathbf{F_{A1}}(\ \ldots, \mathbf{S_A}\ )_{\leftarrow \mathbf{S_{A\circ}}}\\ \{\ldots\} = \mathbf{F_{A2}}(\ \mathbf{S_B}, \mathbf{S_A}\ )_{\leftarrow \mathbf{S_{A\circ}}, \mathbf{S_{B\circ}}}\\ \{\ldots\} = \mathbf{F_B}(\ \mathbf{S_B}\ )_{\leftarrow \mathbf{S_{A\circ}}}\end{array} \iff$$



Parallele Abarbeitung:

$$\begin{array}{l}\mathbf{S_A}\ (\mathsf{trig}_{||})\ \mathbf{F_{A1}}\\ \mathbf{S_A}\ (\mathsf{trig}_{||})\ \mathbf{F_{A2}}\\ \mathbf{S_B}\ (\mathsf{trig}_{||})\ \mathbf{F_{A2}}\\ \mathbf{S_B}\ (\mathsf{trig}_{||})\ \mathbf{F_B}\end{array} \iff \begin{array}{l}\{\ldots\} = \mathbf{F_{A1}}(\ \ldots, \mathbf{S_A}\ )_{\leftarrow \mathbf{S_{A||}}}\\ \{\ldots\} = \mathbf{F_{A2}}(\ \mathbf{S_B}, \mathbf{S_A}\ )_{\leftarrow \mathbf{S_{A||}}, \mathbf{S_{B||}}}\\ \{\ldots\} = \mathbf{F_B}(\ \mathbf{S_B}\ )_{\leftarrow \mathbf{S_{A||}}}\end{array} \iff$$

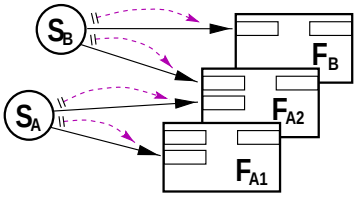


Abbildung 6.14: Darstellungsmittel für sequentielle und parallele Arbeitsmodi bei der Kopplung von Datenobjekten an von ihnen abhängige Funktoren.

6.4 Kontrollfluß in funktionalen Datennetzen

6.4.1 Grundkonzepte und Bewertungskriterien

In diesem Abschnitt sollen basierend auf den im vorherigen Abschnitt vorgestellten Relationen zwischen Daten- und Funktionsobjekten Konzepte für die Ablaufsteuerung der funktional beschriebenen Programmkomponenten untersucht werden. Das Ziel ist dabei auf der einen Seite die direkte und weitestgehend automatische Übertragung der funktionalen Datenfluß-Beschreibung in eine Softwarerepräsentation, und auf der anderen Seite Möglichkeiten für eine gezielte Einflußnahme auf den Kontrollfluß anzubieten, wobei auch in diesem Fall eine konsistente Programmsteuerung ohne Aufrufkonflikte oder Verklemmungen gewährleistet bleiben soll. Die Reihenfolge der Funktionsaufrufe, die sich durch das Setzen geeigneter Relationen zwischen benachbarten Netzobjekten weitestgehend automatisch ergibt, soll vom Entwickler nicht bis ins Detail aufgelöst werden müssen, sondern bestimmten Standardvorgaben folgen.

Die Grundlagen für das Setzen der Relationen sind zum einen der Datenfluß, aber auch ein vom Entwickler auszuwählender Arbeitsmodus für die interne Steuerung des Sensordatenmoduls. Dieser sollte sich an den zur Verfügung stehenden Ressourcen, dem Bedarf an bestimmten Daten sowie an Forderungen einzelner Programmkomponenten hinsichtlich der Wartezeit beim Datenzugriff und des Alters der Daten orientieren.

Prinzipiell können zwei Basis-Steuerungskonzepte unterschieden werden. Das erste basiert auf der Propagierung neuer Eingangsdaten *vorwärts* — Daten- und Kontrollfluß sind gleichgerichtet — durch das Netz (s. Abb. 6.15 a). In diesem Ansatz wird die Ausführung eines Funktors durch die Daten, die von diesem verarbeitet werden, gesteuert — der Funktoraufruf wird durch neue Eingangsdaten getriggert. Alternativ dazu können Ausgabedaten in Abhängigkeit von Anfragen an die entsprechenden Datenobjekte angefordert und somit nur im Bedarfsfall bereitgestellt werden (s. Abb. 6.15 b). Hierbei pflanzen sich Anfragen *rückwärts* durch das Netz fort und werden in der durch den Datenfluß vorgegebenen Reihenfolge befriedigt, so daß alle für eine Anfrage erforderlichen Eingangs- und Zwischendaten — aber auch nur diese — bestimmt werden. Darüber hinaus sind, um einzelne Einschränkungen der Basiskonzepte zu überwinden, Kombinationen beider Mechanismen möglich.

Durch die Auswahl eines dieser Konzepte als Grundlage für den Kontrollfluß ist es möglich, Standardrelationen zwischen benachbarten Datensequenzen und Funktoren direkt aus dem Datenfluß abzuleiten und somit weitestgehend automatisch zu setzen, wodurch ohne zusätzlichen Aufwand für den Entwickler bereits eine konsistente Programmsteuerung installiert wird. Dies schafft zudem die Grundlage dafür, daß Softwareagenten, die das arbeitende System hinsichtlich der verfügbaren Ressourcen überwachen, auf Veränderungen in der gewünschten Weise reagieren können. Werden die Zusammenhänge zwischen den Relationen und der Ablaufsteuerung etwa in Form einer Wissensbasis im Rechner modelliert, können die Agenten durch Neusetzen von Relationen den Datenflußgraphen und dessen Ablaufsteuerung auch zur Laufzeit rekonfigurieren.

Neben der automatischen Vergabe von Standardrelationen bleiben für den Entwickler weiterhin alle Möglichkeiten bestehen, um durch das Anpassen ausgewählter Relationen an die vorgegebenen Randbedingungen gezielt Einfluß auf den Kontrollfluß zu nehmen. Für die Realisierung und Bewertung der verschiedenen Steuerungsmechanismen sind die folgenden Kriterien von besonderer Bedeutung:

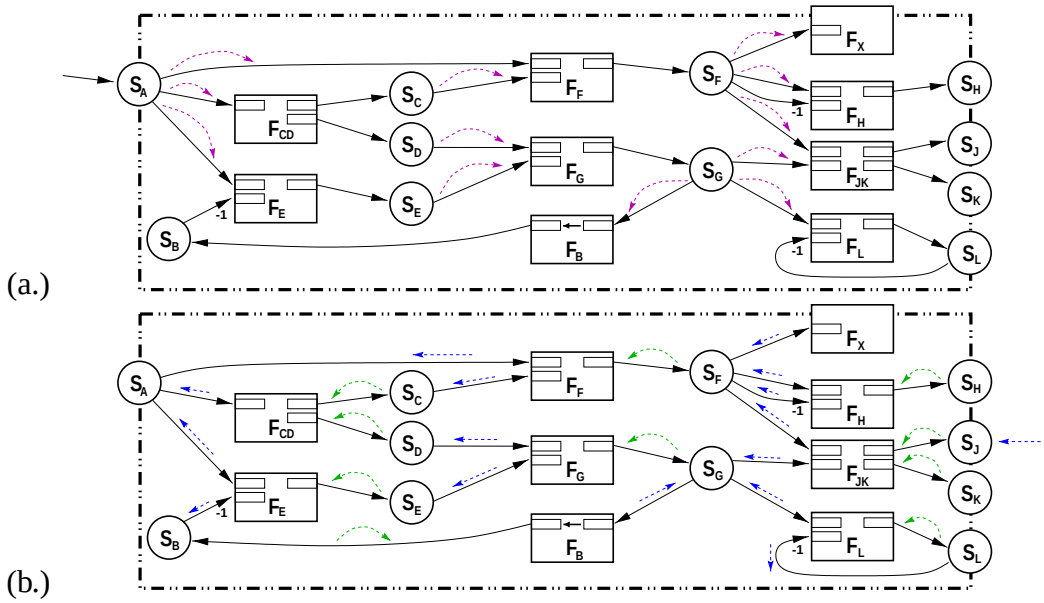


Abbildung 6.15: Basiskonzepte für die Steuerung eines Datennetzes: (a.) Kontrollfluß vorwärts, durch Propagieren der Daten bzw. (b.) Kontrollfluß rückwärts, d.h. anfragegetrieben.

Standardrelationen: Um dem Entwickler eine detaillierte und über den Datenfluß hinausgehende Spezifizierung sämtlicher Relationen zwischen Daten und Funktoren zu ersparen, soll nach Auswahl eines Grundkonzepts für die Modulsteuerung das Ableiten der passenden Relationen weitestgehend automatisch aus dem Datenfluß möglich sein, so daß eine Einflußnahme des Entwicklers auf die Programmsteuerung nur noch in Ausnahmefällen nötig wird.

Erreichbarkeit: Es muß sichergestellt sein, daß alle interessierenden Daten berechnet und eventuell zusätzliche Aktionen (Funktoren ohne Ausgangsdatensequenzen) ausgeführt werden, d.h. alle im Datenflußgraphen enthaltenen Objekte müssen erreichbar sein. Darüber hinaus ist es bei knappen Ressourcen oft wünschenswert, die Menge der in einem Zyklus zu bestimmenden Daten einschränken zu können.

Mehrfachberechnungen bzw. Endlosschleifen: Aufgrund der Forderung nach einer eindeutigen Zuordnung zwischen Zeitpunkten und Datenwerten darf pro Arbeitszyklus jedes Datum nur einmal aktualisiert werden, d.h. ein wichtiges Grundprinzip des vorgestellten Konzepts ist, daß innerhalb eines Zyklus Daten nicht mehrfach berechnet werden. Die Konsequenz dieser Forderung ist, daß die funktionale Ebene, die den Datenfluß *eines* Bearbeitungszyklus beschreibt, keine Schleifen enthalten darf.

Verklemmungen: In einem ungünstig gestalteten Datennetz kann es aufgrund falsch definierter Abhängigkeiten zu Verklemmungen kommen, wenn verschiedene Komponenten wechselseitig aufeinander warten oder sich gegenseitig immer wieder aufrufen. I.d.R. wird dies durch implizite Zugriffe auf Werte des vorangegangenen Zyklus verursacht. Diese gilt es zu erkennen — lokal oder, falls dies nicht möglich sein sollte, durch den Prozeß überwachende Agenten bzw. mit Hilfe eines speziellen Konfigurationszyklus. Ein weiteres wichtiges Grundprinzip ist es daher, Zugriffe in die Vergangenheit in jedem Fall explizit als solche zu kennzeichnen.

Verklemmungen, die durch Endlosschleifen provoziert werden — eine Gruppe von Objekten ruft sich wechselseitig auf und erreicht nie eine Abbruchbedingung, ist leicht zu lokalisieren, da Iterationen innerhalb eines Zyklus nicht erlaubt sind, und jeder Aufruf über die Zykluszeit eindeutig indiziert ist.

Überlastungen: Werden innerhalb eines bestimmten Zeitraums mehr Zyklen angestoßen, als mit der zur Verfügung stehenden Rechenleistung tatsächlich bearbeitet werden können, kommt es zu einer Überlastung des Moduls: Um dies zu verhindern, muß — z.B. durch synchrone Aufrufmechanismen oder Agenten — sichergestellt werden, daß ab einer bestimmten Prozessorauslastung ein neuer Zyklus erst dann angestoßen werden kann, nachdem ein anderer zuvor beendet wurde.

Parallelisierbarkeit: Für die parallele Abarbeitung von Teilaufgaben innerhalb des Datenflußgraphen ergeben sich verschiedene Ansatzpunkte. Sie läßt sich erreichen durch

- eine nebenläufige Bearbeitung verschiedener Teilaufgaben eines Zyklus durch asynchrone Datenzugriffe und Funktionsaufrufe (||-Relationen),
- das Aufteilen des Datennetzes in mehrere unabhängig gesteuerte, parallel arbeitende Teilnetze, die so zu eigenständigen Modulen mit eigenen Zyklen werden,
- eine parallele, zeitversetzte Abarbeitung aufeinanderfolgender Zyklen.

Darüber hinaus kann die durch das betrachtete Datennetz erfolgende Bereitstellung der dynamischen Daten parallel zu anderen Aufgaben der Applikation erfolgen, und es kann parallel und unabhängig voneinander in verschiedenen Teilsystemen eine Sensordatenaufbereitung erfolgen.

Aktiv steuernde und überwachende Komponenten: Neben der Möglichkeit, ein Modul allein von außen durch Anfragen oder neue Eingangsdaten zu steuern, kann ein Modul auch eigene aktive Komponenten besitzen. Dafür eignen sich besonders Softwareagenten, da diese aufgrund ihrer Flexibilität, guten Skalierbarkeit und Lernfähigkeit neben der zyklischen Steuerung des Moduls auch verschiedene andere Aufgaben übernehmen können. Dazu zählt beispielsweise, aktiv die Arbeit der Eingangssensoren, externe Datenanfragen und die eigenen Verarbeitungszyklen zu koordinieren sowie den Graphen hinsichtlich möglicher Konfliktpotentiale oder der Ressourcenauslastung zu überwachen.

Schnittstelle zu anderen Komponenten und externen Prozessen: Die Schnittstelle zu einem externen Prozeß bzw. anderen aktiven Systemkomponenten, wie Sensoren oder unabhängig arbeitenden Modulen erfordert unter Umständen besondere Aufmerksamkeit, damit sich die Kontrollflüsse der Teilsysteme nicht gegenseitig behindern oder blockieren. Darüber hinaus kann für bestimmte Aufgaben auch eine Synchronisation mit dem externen Prozeß notwendig sein.

6.4.2 Charakteristische Netzstrukturen

Inwiefern und mit welchem Aufwand sich die im vorigen Abschnitt beschriebenen Anforderungen in den verschiedenen Steuerungskonzepten erfüllen lassen, und wie die Kontrollflüsse im einzelnen aussehen, soll in den folgenden Abschnitten u.a. anhand der hier aufgeführten charakteristischen Netzstrukturen gezeigt werden. Dabei handelt es sich um typische Teilnetze, in

die sich reale Datenflußgraphen üblicherweise zerlegen lassen, so z.B. auch der Beispielgraph in der Abb. 6.4. Die Abb. 6.16 bis 6.22 zeigen die grundlegenden Netzstrukturen zusammen mit einer entsprechenden Beschreibung in Gleichungsform.

$$\{\mathbf{S_J}, \ldots, \mathbf{S_K}\} = \mathbf{F_{JK}}(\mathbf{S_F}, \ldots, \mathbf{S_G});$$

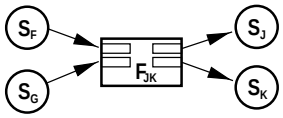


Abbildung 6.16: Einzelner Funktor mit mehreren Ein- und Ausgangsdaten.

Abb. 6.16 zeigt die Standardzelle eines Datenflußgraphen, bestehend aus einem einzelnen Funktor und den anliegenden Datensequenzen. Anhand dieser Struktur lassen sich die grundlegenden Mechanismen der Ablaufsteuerung mit dem daraus resultierenden Kontrollfluß zeigen.

$$\begin{aligned}
\{\mathbf{S_G}\} &= \mathbf{F_G}(\ldots);\\
\{\ldots\} &= \mathbf{F_{JK}}(\ldots, \mathbf{S_G});\\
\{\ldots\} &= \mathbf{F_L}(\mathbf{S_G}, \ldots);\\
\{\ldots\} &= \mathbf{F_B}(\mathbf{S_G});
\end{aligned}$$

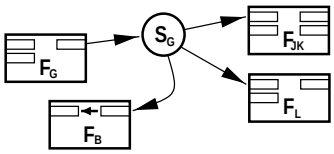


Abbildung 6.17: Sequenz mit genau einem Funktor, der für sie die Daten bereitstellt, und mehreren die Daten 'konsumierenden' Folgefunktoren.

Während Abb. 6.16 vom Funktor ausgeht, steht in Abb. 6.17 die Datensequenz im Mittelpunkt. Charakteristisch für alle Datensequenzen ist, daß sie ihre Daten von genau einem Funktor erhalten, jedoch beliebig viele Funktoren auf sie zugreifen können.

$$\{\,\} = \mathbf{F_X}(\mathbf{S_F});$$

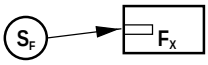


Abbildung 6.18: Funktor ohne Ausgangsdaten.

Da der in Abb. 6.18 dargestellte Funktor keine Ausgangsdaten besitzt, scheidet hier die Datenanfrage als Steuerungsmechanismus aus. Ein solcher Funktor kann nur von seinen Eingangsdaten getriggert oder direkt von einem Agenten aufgerufen werden. Es kann aber auch jeder „Verbraucher“, also jede Komponente, die Sensordaten verarbeitet, ohne selbst wieder zyklische Datensequenzen bereitzustellen, durch einen solchen Funktor repräsentiert werden. Deren Aufgabenspektrum reicht von der Darstellung der Sensordaten auf dem Bildschirm über Steuerungs- und Planungsaufgaben bis hin zu Modelladaptionen oder Ansammlungen von Szenen- oder Objektinformationen in einer Wissensbasis.

$$\{\mathbf{S_H}\} = \mathbf{F_H}(\mathbf{S_F}^{(0,-1)});$$

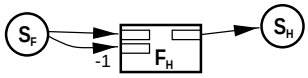


Abbildung 6.19: Funktor, der unterschiedlich alte Datenwerte einer Sequenz verarbeitet.

Funktoren, die wie in Abb. 6.19 gleichzeitig auf mehrere, unterschiedlich alte Werte einer Sequenz zugreifen, werden prinzipiell genauso angesteuert, wie andere Funktoren auch. Dabei kann allerdings auf das mehrfache Setzen von Steuerrelationen zwischen dem Funktor und dem Eingangsdatum verzichtet werden. Beide Zugriffe können aus Sicht der Programmsteuerung zusammengefaßt und wie ein einziger behandelt werden.

$$
\begin{aligned}
\{\mathbf{S_C}, \mathbf{S_D}\} &= \mathbf{F_{CD}}(\mathbf{S_A});\\
\{\mathbf{S_F}\} &= \mathbf{F_F}(\mathbf{S_A}, \mathbf{S_B});\\
\{\mathbf{S_G}\} &= \mathbf{F_G}(\mathbf{S_D}, \mathbf{S_E});\\
\{\mathbf{S_J}, \mathbf{S_K}\} &= \mathbf{F_{JK}}(\mathbf{S_F}, \mathbf{S_G});
\end{aligned}
$$

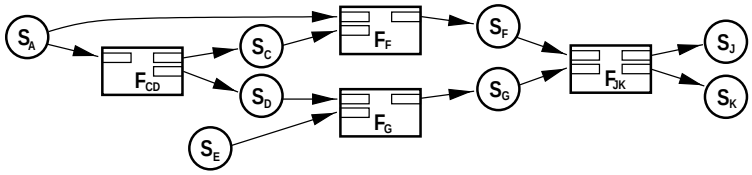


Abbildung 6.20: Komplexes Teilnetz ohne Rückkopplung.

Das etwas komplexere Teilnetz in Abb. 6.20 enthält zwar keine Rückkopplungszweige, jedoch verschiedene Vorwärtszweige, die teilweise voneinander abhängen und in denen sich bei einer asynchronen Ablaufsteuerung Daten unterschiedlich schnell ausbreiten können. Anhand dieses Teilnetzes lassen sich gut asynchrone Steuerungsmechanismen zeigen.

$$\{\mathbf{S_L}\} = \mathbf{F_L}(\mathbf{S_G}, \mathbf{S_L^{(-1)}});$$

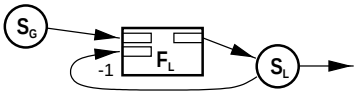


Abbildung 6.21: Einfache Rückkopplung.

$$
\begin{aligned}
\{\mathbf{S_E}\} &= \mathbf{F_E}(\mathbf{S_A}, \mathbf{S_B^{(-1)}});\\
\{\mathbf{S_G}\} &= \mathbf{F_G}(\mathbf{S_A}, \mathbf{S_E});\\
\{\mathbf{S_B}\} &= \mathbf{F_B}(\mathbf{S_G});
\end{aligned}
$$

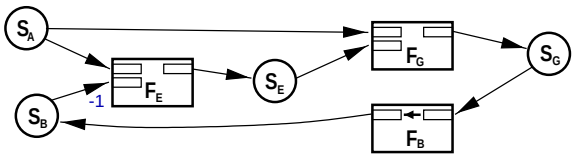


Abbildung 6.22: Mehrstufige Rückkopplung.

Die Netzstrukturen in den Abb. 6.21 und 6.22 zeichnen sich dadurch aus, daß sie einen Rückkopplungszweig enthalten, wodurch der Datenflußgraph nicht mehr zyklenfrei ist. Dabei wird im ersten Fall die Rückkopplung durch einen einzigen Funktor gebildet, der seine eigenen Ausgangsdaten als Eingangsdaten verwendet. Der zweite Fall ist komplexer, da hier sowohl im Vorwärts- als auch im Rückwärtszweig zusätzliche Funktoren liegen. Dadurch ist es nicht mehr ohne weiteres möglich, die Rückkopplung lokal zu erkennen.

Eine zentrale Forderung im hier vorgestellten Konzept ist, daß es stets eine eindeutige Zuordnung zwischen Zeitpunkten und den einzelnen Werten der Datensequenzen gibt, weswegen Mehrfachberechnungen eines Datenwertes innerhalb eines Zyklus verboten sind. Daraus folgt, daß der Datenflußgraph selber keine Schleifen enthalten darf, in denen innerhalb eines Modulzyklus Iterationen über einzelne Datenwerte ausgeführt werden (vgl. dazu Abschnitt 6.2.3).

Auflösen läßt sich dieser Widerspruch zwischen dem notwendigen und auch erlaubten Rückgriff auf alte Werte auf der einen Seite und der Forderung nach einem zyklenfreien Graphen

auf der anderen Seite dadurch, daß zwischen zwei Arten von Rückkopplungen unterschieden wird. Rückkopplungen, die lediglich den Zugriff auf vergangene Datenwerte ermöglichen und pro Modulzyklus nur einmal durchlaufen werden, sind erlaubt, müssen jedoch explizit als solche gekennzeichnet werden, Rückkopplungen, die ausschließlich auf Datenwerte des aktuellen Modulzyklus zugreifen und damit eine echte Schleife darstellen, sind dagegen verboten.

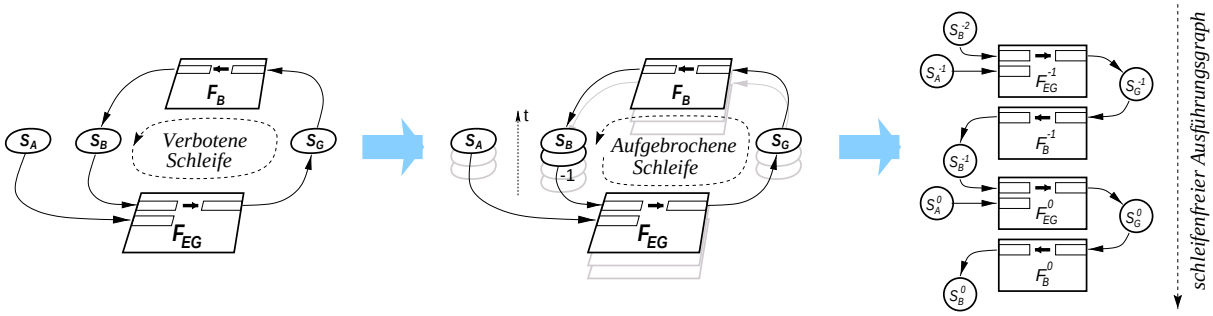


Abbildung 6.23: Erzeugen eines schleifenfreien Datenflußgraphen aus einem Graphen mit Rückkopplungen durch Aufbrechen der Schleifen und explizite Datenzugriffe auf vergangene Werte.

Wie Abb. 6.23 illustriert, ist bei einfachen Rückkopplungen ein Aufbrechen der Schleifen eines Graphen und deren Sequentialisierung ohne weiteres möglich, wobei eine solche Rückkopplung direkt an den Modulzyklus gekoppelt wird. Der explizite Zugriff auf einen alten Wert (z.B. $\mathbf{S_B^{(-1)}}$) kennzeichnet die Stelle, an der der Datenflußgraph aufgebrochen wird. Damit liegt trotz der Rückkopplungen im Graphen de facto ein schleifenfreier Graph[1] vor.

Im direkten Zusammenhang mit dem Zugriff auf ältere Sequenzwerte, sowohl in Rückkopplungen als auch im Vorwärtszweig wie in Abb. 6.19, steht die Bereitstellung von Startwerten: Damit im allerersten Modulzyklus ein solcher Zugriff nicht scheitert, muß für diese Daten ein Startwert zur Verfügung stehen. Dieser kann als elementarer Bestandteil der Datensequenzen unmittelbar beim Erzeugen der Sequenzobjekte initialisiert werden, womit ein allgemeiner Mechanismus für alle Initialisierungsprobleme im Zusammenhang mit Datensequenzen bereitgestellt wird, was den Entwickler von einer expliziten Verwaltung derartiger Daten befreit.

6.4.3 Modulgrenzen, Eingangssensoren und aktive Objekte

Neben der Möglichkeit, die Verarbeitung sämtlicher periodischer Datensequenzen einer Applikation in einem einzigen Datenflußgraphen darzustellen, ist es häufig sinnvoll, das Gesamtsystem in mehrere, relativ unabhängige Module, die über definierte Schnittstellen Daten austauschen und miteinander kommunizieren, zu unterteilen. Diese Aufteilung ermöglicht einerseits einen weitestgehend separaten Entwurf verschiedener Programmteile und kann andererseits die unterschiedlichen Aufgabenbereiche sowie unabhängigen Hardwarekomponenten einer Anwendung widerspiegeln.

[1] In diesem Zusammenhang ist häufig auch von „zyklenfreien Graphen“ die Rede. Dieser Begriff wurde hier wegen der Doppeldeutigkeit des Begriffs “Zyklus“ vermieden. In dieser Arbeit wird Zyklus in erster Linie mit Arbeitsschritten verbunden, also im Sinne von Iterationen verwendet. Für Zyklen in einer graphischen Darstellung sollen dagegen die Begriffe „Schleife“ und „Rückkopplung“ verwendet werden.

Module sind somit durch eine gewisse Eigenständigkeit — basierend auf ihrer Semantik und / oder ihres Ausführungsrahmens — gekennzeichnet. Repräsentiert wird ein Modul $\mathcal{M}$ in erster Linie durch die Menge der in ihm zusammengefaßten Datensequenzen $\{\mathbf{S}\}$ und Funktoren $\{\mathbf{F}\}$, wobei zusätzlich ein Agent $\mathbf{A}$ das Modul steuern oder überwachen kann. Jedes Modul kann dabei seinen eigenen Arbeitszyklus, einen eigenen Kontrollfluß sowie eine eigenständige Steuerungskomponente besitzen, die Ausführung mehrerer Module kann aber auch zusammengefaßt werden, wodurch die Module aus Sicht des Kontrollflusses miteinander verschmelzen.

Neben seiner inneren Struktur wird ein Datenflußgraph somit durch seine Abgrenzung wie auch durch seine Schnittstellen zu anderen Modulen — den Eingangssensoren sowie den die Ausgabedaten verarbeitenden Komponenten — gekennzeichnet. Eine zentrale Frage ist in diesem Zusammenhang, welche Objekte aktiv sind und die Kontrolle über die Modulsteuerung haben. So kann ein Modul entsprechend Abb. 6.24 durch verschiedene Instanzen gesteuert werden:

(a) durch ein oder mehrere Eingangsdatensequenzen, in denen regelmäßig Werte bereitgestellt werden; dabei wird die Modulkontrolle direkt an einen externen Prozeß gekoppelt;

(b) über die Ausgabedatensequenzen, die fortlaufend abgefragt werden; die Modulkontrolle liegt damit beim Verbraucher der Daten, was dem getriggerten Sensormodus entspricht;

(c) durch bestimmte Funktoren, die z.B. zyklisch auf einen *Lowlevel*-Sensor zugreifen und die Daten anschließend im Datennetz verteilen;

(d) durch einen Agenten, der kontinuierlich eine Auswahl von Datensequenzen und Funktoren aufruft; hierbei arbeitet das Modul autonom und ist am flexibelsten konfigurierbar.

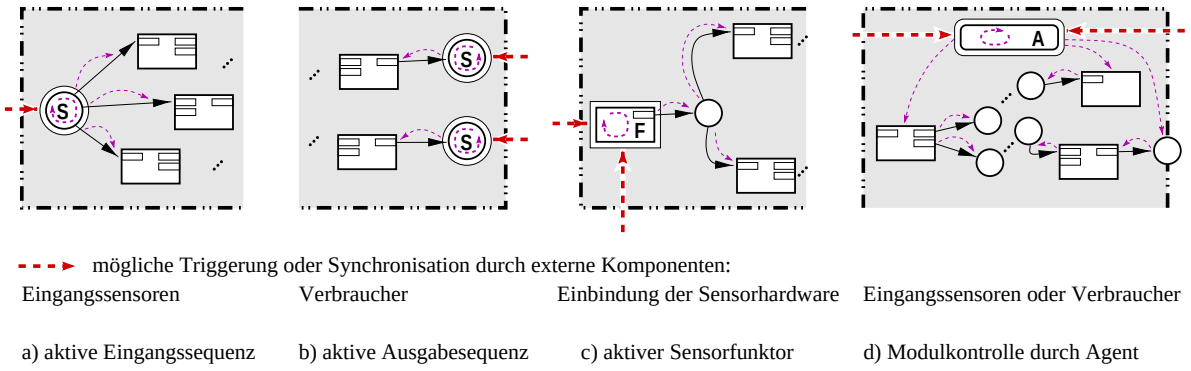
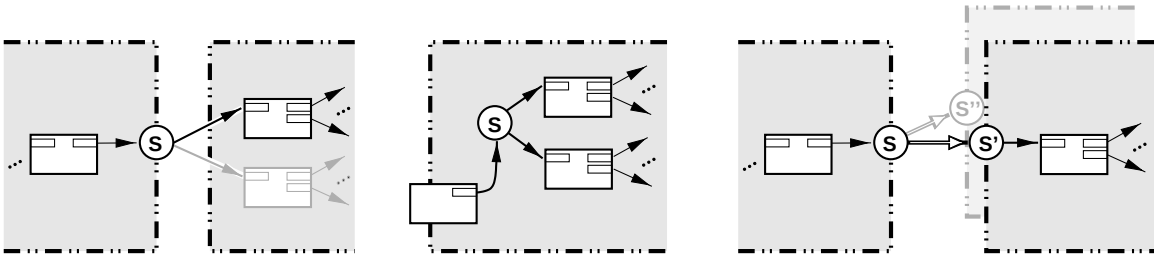


Abbildung 6.24: Modulsteuerung durch aktive Objekte.

Die Ausführung aller Modulzyklen kann in einem einzigen *Thread* erfolgen — genau *ein* Objekt ist aktiv und arbeitet kontinuierlich in einer Schleife, oder aber es wird für jeden Zyklus oder sogar für jede Aktion ein eigener *Thread* gestartet. Sind in einem Modul mehrere Objekte aktiv, muß sichergestellt sein, daß die verschiedenen Kontrollflüsse sich nicht gegenseitig behindern oder blockieren, und daß Zugriffe nur auf konsistente Daten erfolgen dürfen.

Abb. 6.25 illustriert verschiedene Möglichkeiten für die Bereitstellung der durch ein Modul zu verarbeitenden Eingangssensordaten. Dies kann beispielsweise durch einen eigenständigen, unabhängigen Sensor — kontinuierlich oder durch Datenzugriffe getriggert — geschehen, von



a) externe Eingangsdatensequenz b) interne Sensordatensequenz c) lokal gespiegelte Eingangsdatensequenz

Abbildung 6.25: Möglichkeiten für die Bereitstellung der Eingangsdaten eines Moduls.

wo die Daten direkt gelesen werden *(a)*. Greifen allerdings mehrere Funktoren eines Moduls auf diese Datensequenzen zu, ist zu beachten, daß die Sequenzen zwischen zwei Datenzugriffen aktualisiert werden können. Außerdem sind hierbei, falls die Kontrolle exklusiv bei einem einzigen Modulobjekt liegen soll, nur bestimmte Zugriffs- und Steuermechanismen möglich.

Eine zweite Möglichkeit für die Verwaltung der Eingangsdaten besteht darin, die zu verarbeitenden Eingangsdaten durch entsprechende Sensorfunktoren innerhalb des Moduls zu erzeugen *(b)*. Es sind dadurch sowohl die Sensordatensequenzen als auch die Sensorfunktoren ein integraler Bestandteil des Moduls und unterliegen damit direkt der Modulsteuerung. Das bedeutet, daß das Modul selbst und nicht ein externes Signal vorgibt, wann auf den Sensor zugegriffen wird. Dabei ist es unerheblich, ob der Sensorfunktor direkt auf die Einbindung eines Hardwaresensors zugreift oder wiederum logische Sensordaten einer externen Datensequenz verwendet. Diese Methode ist zur ersten äquivalent, wenn bei dieser die Eingangssensoren durch die Zugriffe getriggert werden. In diesem Fall wird der Sensor aus Sicht der Steuerung in das betrachtete Modul integriert.

Schließlich bietet es sich insbesondere für Sensordaten, die extern in einem anderen Prozeßraum oder auf einem anderen Rechner bereitgestellt werden, an, diese Datensequenzen lokal, ggf. sogar exklusiv für jedes Modul, das diese Daten verarbeitet, zu spiegeln *(c)*. Datensequenzen unterstützen dies aktiv durch ihre Möglichkeit, bei entsprechender Konfiguration, nach jeder Aktualisierung die neuen Daten in einem Rechnernetz zu verteilen. Lokale Sequenzkopien in anderen Prozessen und auf anderen Rechnern können diese Daten empfangen und somit ein getreues Abbild der Originalsequenz darstellen.

Ein wichtiger Aspekt bei der Verarbeitung *mehrerer* Eingangsdaten, die von unabhängigen Quellen stammen, ist die Frage, ob diese Daten synchron sind, d.h. die gleiche Datenmeßzeit besitzen, oder nicht. Daher soll zusätzlich zu den in Abschnitt 6.4.2 genannten Netzstrukturen das in Abb. 6.26 gezeigte Teilnetz mit zwei unabhängigen Eingangsdatensequenzen untersucht werden. Diese Eingangsdaten können entweder auf den gleichen (Hardware-) Sensor zurückzuführen sein und synchrone Szenendaten, d.h. Daten mit der exakt gleichen Szenenmeßzeit, enthalten, oder sie stammen von unterschiedlichen, nicht synchronisierten Sensoren und repräsentieren Szeneninformation von verschiedenen Zeitpunkten.

6.4.4 Zeittolerante Datenzugriffe

Die gemeinsame Verarbeitung mehrerer nicht synchron gemessener Eingangsdaten erfordert eine gewisse Toleranz bei der auf der Datenmeßzeiten basierende Überwachung der Zyklen-

1. Sensor:

$\{\mathbf{S_A}\} = \mathbf{F_A}();$

2. Sensor:

$\{\mathbf{S_B}\} = \mathbf{F_B}();$

Verarbeitungsmodul:

$\{\mathbf{S_C}, \mathbf{S_D}\} = \mathbf{F_{CD}}(\mathbf{S_A});$

$\{\mathbf{S_E}\} = \mathbf{F_F}(\mathbf{S_A}, \mathbf{S_B});$

$\{\mathbf{S_G}\} = \mathbf{F_G}(\mathbf{S_D}, \mathbf{S_E});$

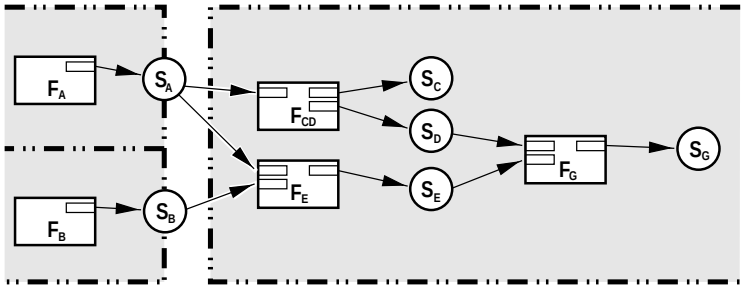


Abbildung 6.26: Teilnetz mit zwei unabhängigen Eingangsdatensequenzen.

konsistenz. Häufig wird für einen Zyklus die Datenmeßzeit von den Daten eines Sensors — dem *primären* oder *dominanten* Sensor — vorgegeben. Da die Datenmeßzeiten anderer — *sekundärer* — Sensoren im allgemeinen davon abweichen, würden die zeitgenauen Zugriffsrelationen $(\mathsf{get}_{\mathsf{t}}^{\triangleleft})$ und $(\mathsf{get}_{\mathsf{t}}^{?})$ nicht zum Erfolg führen. Ein ähnliches Problem ergibt sich, wenn die aktuelle Datenmeßzeit im Moment des Zugriffs (noch) nicht bekannt ist.

Mit Hilfe der im Abschnitt 6.3.3 eingeführten (get)-Relationen lassen sich verschiedene zeittolerante Zugriffsverfahren spezifizieren. Diese gestatten beispielsweise den Zugriff auf die Werte der Sequenz $\mathbf{S_B}$, wenn im Beispielgraphen in Abb. 6.26 die Zyklen durch die Sensordaten von $\mathbf{S_A}$ dominiert werden: $\mathbf{F_E}\,(\mathsf{get})\,\mathbf{S_B}$. Dabei kann der Zugriff an bestimmte Randbedingungen geknüpft sein, die in diesem Abschnitt genauer untersucht werden sollen.

Charakteristisch für diese Zugriffsmethoden ist, daß sie in jedem Fall einen gewissen Zeitfehler mit sich bringen, der sich mehr oder weniger stark auf die anschließenden Verfahren auswirken kann und dort entsprechend berücksichtigt werden sollte. Konkret lassen sich die Zugriffsmechanismen folgendermaßen charakterisieren:

- $(\mathsf{get}_{[\,]}) \quad \rightsquigarrow \quad S_B := \mathsf{get}_{[\,]}(\mathbf{S_B}, 0)$

  Dieser Zugriff liefert immer den neuesten verfügbaren Wert der Sequenz. Dabei ist zu beachten, daß sich zwischen zwei Zugriffen der Wert ändern kann. In Abhängigkeit vom Zeitverhalten der beiden Sensoren können die Datenmeßzeiten relativ stark voneinander abweichen.

- $(\mathsf{get}_{[\mathsf{t}]}) \quad \rightsquigarrow \quad S_B := \mathsf{get}_{[\,]}(\mathbf{S_B}, 0, t_{\mathsf{g}}(S_A))$

  Bei dieser Zugriffsmethode wird aufgrund der übergebenen Referenzzeit immer der gleiche Wert geliefert, auch wenn innerhalb des Moduls mehrfach auf die Sequenz zugegriffen wird und die Sequenz zwischenzeitlich aktualisiert wurde. Das sekundäre Datum S_B^0 ist in jedem Fall älter als das primäre S_A^0, da es während der Messung von S_A^0 — als Referenzzeit wurde die Datenmeßzeit $t_{\mathsf{g}}(S_A^0)$ angegeben — bereits verfügbar gewesen sein muß.

- $(\mathsf{get}_{\mathsf{t}}^{\bowtie}) \quad \rightsquigarrow \quad S_B := \mathsf{get}_{\mathsf{t}}^{\bowtie}(\mathbf{S_B}, t_{\mathsf{g}}(S_A))$

  Hiermit wird der erste *Meßwert* von $\mathbf{S_B}$ nach $\mathbf{S_A}$, d.h. $t_{\mathsf{g}}(S_B^0) \geq t_{\mathsf{g}}(S_A^0)$ zurückgegeben. Steht dieser noch nicht zur Verfügung, wird auf ihn gewartet, was nach den Interpolationsmethoden die beste zeitliche Annäherung der Daten verspricht.

- $(\mathsf{get}_\mathsf{t}^{\sim}) \quad \leadsto \quad S_B := \mathsf{get}_\mathsf{t}^{\sim}(\mathbf{S_B}, t_\mathsf{g}(S_A))$

  Ist diese Relation gesetzt, wird aus den vorhandenen Sequenzwerten ein Wert für die Datenmeßzeit des primären Sensors interpoliert oder prädiziert, je nachdem ob die Sequenz bereits einen neueren Wert enthält ($t_\mathsf{g}(S_B^0) > t_\mathsf{g}(S_A^0)$) oder nicht. Dadurch kann der durch die verzögerte Messung der Daten bedingte Fehler deutlich reduziert werden. Voraussetzung für die Anwendbarkeit dieser Relation ist jedoch, daß die Sequenzdaten interpolierbar sind und die Sequenz einen entsprechenden Interpolationsfunktor $\mathbf{F_{int}}$ besitzt. Für Datentypen wie Bildmatrizen oder Regionen, die nicht ohne weiteres interpoliert werden können, kann jedoch auch eine triviale Interpolationsfunktion gesetzt werden, die den zur Anfragezeit nächsten Wert liefert.

- $(\mathsf{get}_\mathsf{t}^{\widetilde{\bowtie}}) \quad \leadsto \quad S_B := \mathsf{get}_\mathsf{t}^{\widetilde{\bowtie}}(\mathbf{S_B}, t_\mathsf{g}(S_A))$

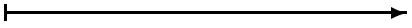
  Diese Zugriffsrelation verhält sich wie $(\mathsf{get}_\mathsf{t}^{\sim})$. Enthält die Sequenz allerdings noch keinen Wert, der neuer ist, als das primäre Sensordatum ($t_\mathsf{g}(S_B^0) < t_\mathsf{g}(S_A^0)$), wird erst auf die Aktualisierung der Sequenz gewartet und dann interpoliert. Damit wird in jedem Fall die Prädiktion in die Zukunft durch eine Interpolation mit den flankierenden Werten ersetzt, was die beste zeitliche Annäherung der Daten bewirkt und somit den Einfluß des Zeitfehlers am besten zu eliminieren hilft. Dafür muß allerdings unter Umständen eine Verzögerung der Berechnung in Kauf genommen werden.

- $(\mathsf{get}_\mathsf{t}^{?}) \quad \leadsto \quad S_B := \mathsf{get}_\mathsf{t}^{?}(\mathbf{S_B}, t_\mathsf{g}(S_A))$

  Auch die $(\mathsf{get}_\mathsf{t}^{?})$-Relation ist in diesem Zusammenhang anwendbar. Voraussetzung ist dafür allerdings, daß für die sekundären Sequenzen ein Gültigkeitsintervall i_v der Länge τ_v entsprechend Abschnitt 5.1.4 definiert wurde. Die Sequenzdaten bleiben dadurch über einen gewissen Zeitraum nach ihrer Messung gültig. Zugriffe auf sie werden — in dem durch τ_v vorgegebenen Zeitrahmen — akzeptiert, obwohl die Daten nicht genau gleichzeitig mit den primären Daten aufgenommen wurden. Unter Umständen kann es bei dieser Zugriffsmethode vorkommen, daß einige Zyklen ausgelassen werden, wenn die Messungen der Eingangsdaten zeitlich zu weit auseinander liegen. Zu beachten ist hierbei allerdings die Datenverzögerung der Sensoren τ_s (vgl. Abschnitt 5.3.2). Ist diese für den sekundären Sensor deutlich größer als für den primären ($\tau_\mathsf{s}(\mathbf{S_B}) > \tau_\mathsf{s}(\mathbf{S_A}) + \tau_\mathsf{v}(\mathbf{S_B})$), wird der Zugriff nie erfolgreich sein.

Um aus diesen Möglichkeiten nun eine konkrete Zugriffsmethode auswählen zu können, muß der Entwickler vor allem das Zeitverhalten der Sensoren, die Semantik und Typen der Daten, und die Dynamik der Szene beachten. Letztere ist entscheidend für das Ausmaß der Fehler verantwortlich, die durch die zeitverschobene Messung der gemeinsam zu verarbeitenden Sensordaten entstehen. Darüber hinaus werden durch die Systemdynamik Vorgaben hinsichtlich der Reaktionszeit der Anwendung auf externe Ereignisse gemacht, was Auswirkungen auf die maximale Datenverzögerung und somit auch auf die maximalen Wartezeiten hat.

Abschließend werden die Zugriffsrelationen gegenübergestellt und hinsichtlich verschiedener Bewertungskriterien miteinander verglichen. Der Entwickler hat nun die Aufgabe diese Kriterien der Aufgabe entsprechend abzuwägen, um zu einer für die jeweilige Anwendung optimalen Vorgehensweise zu gelangen.

Der **Zeitmeßfehler** bezeichnet die Abweichung zwischen dem verfügbaren Wert und dem Wert, der zur angegebenen Zeit theoretisch zu messen gewesen wäre. Reduziert wird er durch Zugriffe, die das Datum mit dem geringst möglichen Abstand zur dominanten Meßzeit liefern oder durch Interpolation.	$(\mathrm{get}_t^{\widetilde{\bowtie}})$ $(\mathrm{get}_t^{?})$ $(\mathrm{get}_{[\,]})$ $(\mathrm{get}_t^{\sim})$ $(\mathrm{get}_t^{\bowtie})$ $(\mathrm{get}_{[t]})$  gering groß
Die **Zugriffskonsistenz** bezeichnet die Eigenschaft, bei wiederholten Zugriffen mit denselben Parametern immer identische Datenwerte zu liefern. Wird die Sequenz zwischen zwei Zugriffen aktualisiert, liefert ein Zugriff ohne Zeitbezug $(\mathrm{get}_{[\,]})$ beim zweiten Aufruf ein anderes Datum als beim ersten. Wurde ein Wert beim ersten Zugriff mit $(\mathrm{get}_t^{\sim})$ prädiziert, kann nach einer Aktualisierung der neue Sequenzwert die Prädiktion bzw. Interpolation zu einem anderen Ergebnis führen. Schließlich kann durch die Aktualisierung ein beim ersten Versuch abgebrochener $(\mathrm{get}_t^{?})$ -Zugriff später erfolgreich sein.	$(\mathrm{get}_t^{\widetilde{\bowtie}})$ $(\mathrm{get}_t^{\sim})$ $(\mathrm{get}_t^{\bowtie})$ $(\mathrm{get}_{[\,]})$ $(\mathrm{get}_t^{?})$ $(\mathrm{get}_{[t]})$  gering hoch
Die **Wartezeit** beschreibt, ob der Zugriff sofort beantwortet wird oder erst nach einer bestimmten Rechen- oder Wartezeit. Sie ist für Methoden, die nur einen Wert auslesen vernachlässigbar gering, muß der Wert erst berechnet werden, ist sie etwas größer, in den meisten Fällen aber immer noch gering. Wird allerdings auf die Aktualisierung der Sequenz gewartet, kann das entscheidende Auswirkungen auf die gesamte Zykluslänge haben.	$(\mathrm{get}_t^{?})$ $(\mathrm{get}_t^{\sim})$ $(\mathrm{get}_t^{\widetilde{\bowtie}})$ $(\mathrm{get}_{[\,]})$ $(\mathrm{get}_{[t]})$ $(\mathrm{get}_t^{\bowtie})$ gering hoch
Verfügbarkeit: gibt an, ob der Aufruf scheitern kann. Bis auf $(\mathrm{get}_t^{?})$ liefern alle Zugriffsmethoden, eine richtige Konfiguration vorausgesetzt, mit Sicherheit einen Wert zurück. Nicht berücksichtigt werden hier mit einer Ausnahmebehandlung einhergehende Methodenabbrüche, die auf eine fehlende Initialisierung oder nicht gesetzte Interpolationsfunktionen zurückzuführen sind.	$(\mathrm{get}_t^{\bowtie})$ $(\mathrm{get}_t^{\widetilde{\bowtie}})$ $(\mathrm{get}_{[\,]})$ $(\mathrm{get}_t^{?})$ $(\mathrm{get}_t^{\sim})$ $(\mathrm{get}_{[t]})$  eingeschränkt sicher

6.4.5 Strategien zum Erkennen und Lokalisieren von Konflikten im Design des Datenflußgraphen

Ein wichtiges Ziel einer umfassenden Datenflußanalyse ist das Erkennen und Lokalisieren von potentiellen Konflikten im Graphen, die zu Problemen wie Verklemmungen, Unerreichbarkeit oder Endlosschleifen führen können. Verursacht werden derartige Konflikte dadurch, daß der Entwickler beim Entwurf Datenflußbeziehungen falsch spezifiziert, indem er z.B. Rückkopplungen oder nicht synchronisierte Eingangsdaten nicht explizit angibt. Ob und in welcher Form ein Konflikt auftritt, hängt maßgeblich von den installierten Steuerungsmechanismen ab.

An dieser Stelle sollen prinzipielle Möglichkeiten für das Erkennen und Lokalisieren derartiger Konflikte untersucht werden. Hierbei können diverse Strategien, die in unterschiedlichen Ebenen ansetzen, verfolgt werden. Sie unterscheiden sich z.T. erheblich hinsichtlich ihres Aufwandes und ihrer Flexibilität, aber auch im Spektrum der erkennbaren Konflikte. Das Ziel ist hier nicht die automatische Beseitigung der Konflikte, sondern beschränkt sich darauf, sie zu erkennen, zu lokalisieren und anzuzeigen. Das ergibt sich aus der Annahme, daß die Grundlage eines korrekten Systementwurfs die korrekte Problemanalyse und damit verbunden die richtige semantische, aber auch zeitliche Einordnung der Daten ist. Designfehler deuten damit auf Fehler bei der Problemanalyse hin und sollten vom Entwickler explizit korrigiert werden.

Lokale Konfliktdiagnose in den beteiligten Objekten: Am flexibelsten und am einfachsten handzuhaben sind Mechanismen, die lokal, bei den unmittelbar beteiligten Objekten einer Aktion ansetzen. Im Idealfall führt beim Aufruf einer Methode ein einfacher Test des eigenen Zustands, d.h. der aktuell bereits laufenden Aktionen sowie der Zustände der benachbarten Objekte dazu, daß ein Konflikt erkannt und sofort durch eine Ausnahmebehandlung signalisiert wird. Der Vorteil einer lokalen Konfliktdiagnose liegt vor allem in deren Unabhängigkeit gegenüber Veränderungen der Netztopologie. Können Konflikte lokal erkannt werden, greift der Mechanismus direkt und ohne zusätzlichen Aufwand nach jeder Neukonfiguration des Graphen.

Potentielle Endlosschleifen stellen einen Konflikt dar, der hier aufgrund der Einschränkung, daß Daten für einen bestimmten Zeitpunkt nicht mehrfach berechnet werden dürfen, leicht lokal zu erkennen ist, da Funktoren und Datensequenzen alle Aktionen mit einem eindeutigen Zeitpunkt verbinden und alte Aktionen eine Zeitlang gespeichert werden.

Es gibt allerdings auch Inkonsistenzen, die nicht ohne weiteres lokal erkannt werden können. Wird beispielsweise ein Methodenaufruf aufgrund fehlender Daten nicht erfolgreich durchgeführt und abgebrochen, ist zu diesem Zeitpunkt i.d.R. nicht entscheidbar, ob die Methode zu einem späteren Zeitpunkt noch einmal aufgerufen wird, und ob dann die Randbedingungen günstiger sein werden. Ein möglicher Ausweg für diese Konfliktklasse besteht darin, bei jeder Aktion deren erfolgreiche Beendigung im vorhergehenden Zyklus zu testen, was aufgrund des gespeicherten Objektstatus leicht möglich ist.

Globale Werkzeuge für die Netzanalyse: Das andere Extrem für die Konfliktdiagnose sind Werkzeuge, die global arbeiten und eine vollständige Analyse der Netztopologie vornehmen. Unter Beachtung der Steuerungsrelationen müssen sie den Graphen explizit nach möglichen Konflikten untersuchen. Dies ist äußerst aufwendig, da ein solches Werkzeug Expertenwissen über alle möglichen Konflikte und Steuermechanismen besitzen muß. Darüber hinaus werden Wege benötigt, die Netzdaten dem Werkzeug mitzuteilen. Bei jeder Änderung der Netztopologie oder einer der gesetzten Relationen muß die Analyse wiederholt werden. Dieser Weg kommt hier aufgrund des Aufwandes und der fehlenden Flexibilität nicht in Betracht.

Zyklusnahe Statusanalyse durch Agenten: Eine weitere Möglichkeit, die hinsichtlich Aufwand und Flexibilität zwischen den bisher genannten Verfahren angesiedelt ist, bieten agentenbasierte Verfahren, bei denen entweder nach jedem Zyklus oder aber nach jeder Änderung der Netztopologie die Zustände der einzelnen Netzelemente untersucht werden. Kennt ein Agent die zu aktualisierenden Datensequenzen, kann er am Ende eines Zyklus überprüfen, ob die Daten tatsächlich aktualisiert wurden. Dieser Test kann unabhängig und ohne genaue Kenntnis der Netztopologie durchgeführt werden, allerdings müssen Änderungen in der Liste der zu aktualisierenden Daten dem Agenten bekannt gemacht werden.

Eine Alternative dazu, die ohne die genaue Kenntnis der zu aktualisierenden Sequenzen auskommt, kann mit Hilfe einer modulweiten Liste ermöglicht werden, in die die Objekt bestimmte ihrer Aktionen wie z.B. den Aufruf der call()-Methode eines Funktors eintragen, sobald diese im Zyklus zum ersten Mal durchgeführt wird. Für Aktionen, die erfolgreich abgeschlossen wurden, löscht das Objekt den entsprechenden Eintrag wieder aus der Liste. Einträge, die bis zum Ende des Zyklus nicht aus der Liste gelöscht wurden, zeigen direkt die Unerreichbarkeit von Daten oder Funktoren an.

Testzyklen mit überwachendem Agenten: Lokale und die zuletzt genannten agentenbasierten Verfahren scheitern, wenn ein Konflikt zu einer Verklemmung führt, da in diesem Fall der Zyklus zu keinem Ende kommt. Besteht diese Gefahr, was insbesondere bei asynchronen Steuerungsmechanismen der Fall ist, sollte nach jeder Änderung des Netzdesigns ein spezieller Analysezyklus durchlaufen werden, der von einem Agenten angestoßen und anschließend ausgewertet werden kann. Hierbei können alle Daten- und Funktionsobjekte ihre jeweiligen Aktionen in eine Liste eintragen. Darüber hinaus besteht die Möglichkeit, daß Funktoren ihre do()-Methode überspringen, wodurch die Laufzeit eines Testlaufs allein durch die Verwaltungsoperationen bestimmt wird und damit sehr kurz und gut abschätzbar ist. Nach Beendigung des Zyklus oder nach einer festen Zeitspanne analysiert der Agent die Liste, wodurch leicht festzustellen ist, ob alle Funktoren aufgerufen und auch wieder ordnungsgemäß beendet wurden.

Auf die Aktionsliste und die anschließende agentenbasierte Auswertung des Analysezyklus kann verzichtet werden, wenn der gesamte Testzyklus in einem ungeteilten Kontrollfluß ausgeführt wird. Wie die folgenden Abschnitte zeigen werden, läßt sich in diesem Fall jede potentielle Verklemmung lokal erkennen.

Voraussetzung für diese Verfahren ist zum einen die Mitwirkung der Funktoren und Datensequenzen, die den Datenflußgraphen bilden, zum anderen ein Objekt, das in der Lage ist, die während eines regulären oder Testzyklus angesammelten Informationen auszuwerten. Beides ist, da als dynamische Elemente des Graphen nur Datensequenzen und Funktoren in Frage kommen, mit relativ geringem Aufwand möglich, und soll daher — wenn lokale Verfahren nicht greifen — als Lösung präferiert werden.

Das Potential möglicher Konflikte hängt nicht nur vom Netzdesign sondern auch wesentlich von der Art der Programm- bzw. Modulsteuerung ab. So führen mehrere, parallel arbeitende Kontrollflüsse, die sich asynchron im Netz ausbreiten, leichter zu Konflikten, die dazu schwerer zu erkennen sind, als eine Steuerung, die synchron in nur einem Kontrollfluß erfolgt. Dieser Punkt wird in den folgenden Abschnitten im Rahmen der Verwendung asynchroner Relationen genauer untersucht werden.

Neben der Beschreibung und einem Vergleich der Ablaufmechanismen und Kontrollflüsse unter den verschiedenen Steuerungskonzepten sollen in den folgenden Abschnitten weitere Designentscheidungen für den objektorientierten Entwurf der beteiligten Daten- und Funktionsobjekte und deren Realisierung in Software abgeleitet werden.

6.4.6 Asynchrone Steuerungskonzepte

Eine Stärke funktionaler Programmbeschreibungen ist, daß sie auf natürliche Art und Weise die potentiellen Nebenläufigkeiten der durch sie repräsentierten Aufgaben ausdrücken können. Damit dieser Vorteil bei der Realisierung des funktionalen Datenflußgraphen in einem Compu-

tersystem zum Tragen kommt, ist es notwendig, die Aspekte der Beschreibung, die die potentiellen Nebenläufigkeiten der Anwendung enthalten, so in die Software zu übertragen, daß diese potentiell oder tatsächlich parallel ausführbar sind. Dabei sind vor allem zwei Aspekte von Bedeutung: der Datenfluß kann sich an bestimmten Stellen im Graphen teilen und an anderen muß er aufgrund bestehender Datenabhängigkeiten wieder zusammengeführt werden.

Für die Umsetzung folgt daraus, daß alle Übergänge des Graphen, in denen sich der Kontrollfluß teilen kann (bzw. soll), asynchron ansteuerbar sein müssen. Dies führt dann dazu, daß die nachfolgende Teilaufgabe in einem eigenen, unabhängigen Programmfaden oder Prozeß abgearbeitet werden kann, und so die Daten sich parallel und unabhängig voneinander im Datenflußgraphen ausbreiten können. Für die Elemente des Graphen hingegen, in denen verschiedene Daten gemeinsam verarbeitet werden, werden Programmkonstrukte benötigt, die die asynchronen Datenströme wieder synchronisieren.

Beide Aufgaben — das Aufteilen des Kontrollflusses und Starten der verschiedenen, parallelen Programmzweige (z.B. als *Threads*) wie auch die Synchronisation der Daten für ihre gemeinsame Verarbeitung (z.B. durch *Semaphore*) — erfordern allerdings einen gewissen zusätzlichen Verwaltungsaufwand bei der Programmabarbeitung. Sind dabei weniger reale Prozessoren verfügbar, als Programmzweige gleichzeitig aktiv, müssen sich mehrere Prozesse einen Prozessor teilen, wofür sie (i.d.R. durch den *Scheduler* des jeweiligen Betriebssystems) nacheinander Rechenzeitfenster zugewiesen bekommen. Die damit verbundenen Prozeßwechselzeiten wirken sich zusätzlich negativ auf die Gesamtperformanz aus. Besitzt das für die Bearbeitung eines Moduls zuständige Hardwaresystem überhaupt nur einen einzigen Prozessor, kann auf den Parallelisierungsoverhead und ein Aufteilen des Kontrollflusses von vornherein verzichtet werden.

Möglichkeiten der Einflußnahme auf den Parallelisierungsgrad: Um nun — und zwar sowohl bei der Realisierung als auch später im laufenden System — die Aufteilung des Kontrollflusses gezielt beeinflussen zu können, kann jede Steuerrelation synchron oder asynchron spezifiziert werden, wodurch festgelegt wird, ob sich der Kontrollfluß an dieser Stelle teilt oder nicht. Dabei ergeben sich zwei grundlegende Wege für den Umgang mit synchronen und asynchronen Relationen. Zum einen kann der Entwickler bei der Realisierung des Programms alle Steuerrelationen des Graphen einheitlich asynchron oder synchron vordefinieren, letzteres z.B. wenn bekannt ist, daß nur ein Prozessor zur Verfügung steht. Er kann aber auch explizit, um das System zu optimieren, wenige ausgewählte Relationen asynchron (oder synchron) setzen, und so gezielt von der Standardvorgabe abweichend bestimmte Teile des Graphen parallelisieren und andere nicht. Diese Aufgabe kann darüber hinaus von Agenten zur Laufzeit übernommen werden, die die Rechenzeiten der verschiedenen Funktoren sowie die Auslastung der vorhandenen Ressourcen überwachen, wodurch die Parallelisierung flexibel für wechselnde Randbedingungen optimiert werden kann.

Durch die nebenläufige Bearbeitung unterschiedlicher Teilnetze ergeben sich, ohne das dies gesondert spezifiziert werden muß, prinzipiell zwei Formen der Parallelisierung innerhalb eines Datenflußgraphen bzw. Moduls. Zum einen können, wenn die bestehenden Abhängigkeiten zwischen den Daten dies erlauben, innerhalb eines Zyklus verschiedene Aufgaben parallel bearbeitet werden, zum anderen ist es möglich, daß einzelne Abschnitte des Graphen nach dem Pipelining-Prinzip in unterschiedlichen Zyklen arbeiten: während sich die letzten Funktoren

noch im i-ten Zyklus befinden, sind die ersten Funktoren schon ein oder mehrere Zyklen weiter.

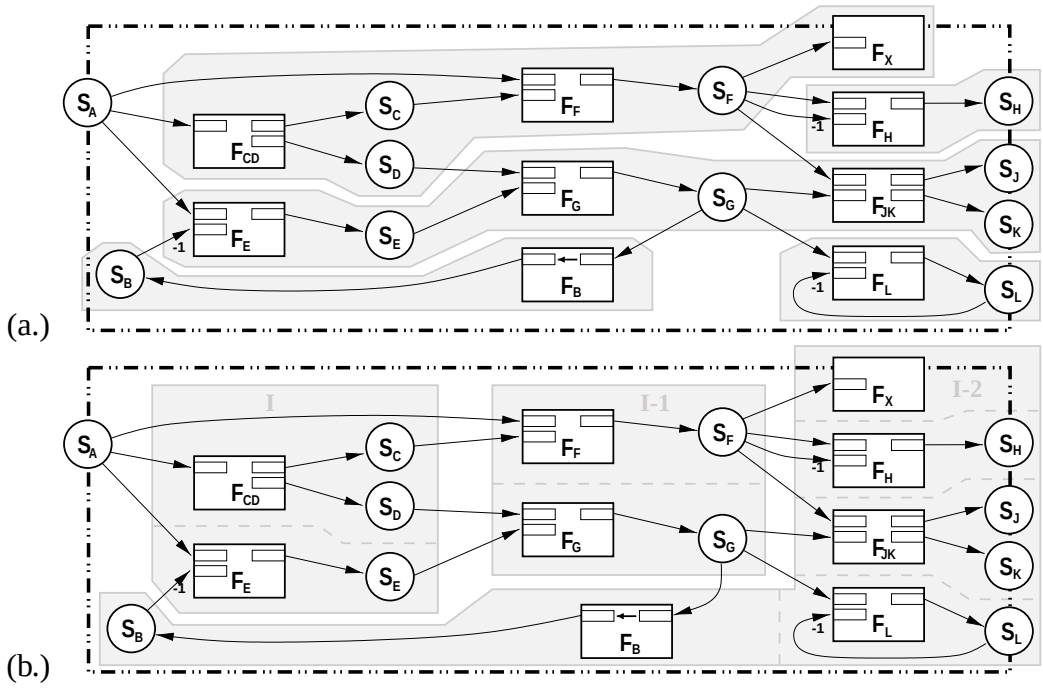


Abbildung 6.27: Parallelisierung eines Beispielgraphen: (a.) mittels Aufgabenparallelität und (b.) durch Pipelining.

Abb. 6.27 illustriert dies für den Beispielgraphen aus Abb. 6.4. Zur Aufgabenparallelität eignen sich z.B. die folgenden Funktoren (vgl. Abb. 6.27 a): $\mathbf{F_{CD}}$ und $\mathbf{F_F}$ parallel zu $\mathbf{F_E}$ und $\mathbf{F_G}$; und die Funktoren $\mathbf{F_X}$, $\mathbf{F_H}$, $\mathbf{F_{JK}}$, $\mathbf{F_L}$ und $\mathbf{F_B}$ können prinzipiell alle parallel ausgeführt werden, z.T. auch parallel zu einer der zuvor genannten Gruppen. Im Pipelining können, eine in etwa gleichmäßige Verteilung der Rechenzeit vorausgesetzt, die folgenden Funktoren zusammengefaßt werden (vgl. Abb. 6.27 b): (I): $\mathbf{F_{CD}}$ mit $\mathbf{F_E}$, $(I-1)$: $\mathbf{F_F}$ mit $\mathbf{F_G}$ sowie $(I-2)$: die restlichen Funktoren. Innerhalb einer Pipeliningruppe ist, wie die Abbildung zeigt, natürlich wiederum die parallele Aufgabenbearbeitung möglich. Auch müssen keine scharfen Grenzen zwischen den Zyklen definiert werden, da sich diese Aufteilung im laufenden Betrieb automatisch ergibt.

Soll der Datenflußgraph explizit entsprechend Abb. 6.27 a oder b parallelisiert werden, bedeutet dies, daß innerhalb einer gemeinsam zu parallelisierenden Objektgruppe nur synchrone Relationen erlaubt sind. Diese Forderung allein reicht allerdings noch nicht aus, um zu garantieren, daß alle Aktionen einer Parallelisierungsgruppe in genau einem *Thread* ablaufen. Um zu verhindern, daß z.B. der Hauptkontrollfluß aufgrund eines fehlenden Eingangsdatums abgebrochen wird und statt dessen von der Seite zusätzliche *Threads* gestartet werden, sind für die Querverbindungen zu anderen Parallelisierungsgruppen nur ausgewählte Relationen (z.B. keine Datenpropagierung (trig), und nur wartende Datenzugriffe $(\mathsf{get}_{\mathsf{t}}^{\bowtie})$) erlaubt.

Werden dagegen prinzipiell alle Relationen asynchron spezifiziert, ergeben sich Parallelisierungsgrad, Ausführungsmodus und die Reihenfolge der Aufrufe automatisch: Sobald die Bedingungen für die Ausführung einer Operation erfüllt sind, wird diese — auf einem freien

(virtuellen) Prozessor — gestartet. Zum Abschluß einer Operation werden eventuelle Folgeoperationen angestoßen und der eigene Programmzweig beendet. Dies hat den Vorteil, daß sich die Parallelisierung automatisch, ohne eine planende Instanz ergibt, und der Kontrollfluß sich selbsttätig einem wechselnden Ressourcenangebot anpassen kann.

Methodenmodell für nebenläufiges Arbeiten: Im Gegensatz zu einer vollständig synchronen Modulsteuerung wird im asynchronen Fall die Aufrufreihenfolge nicht mehr allein durch das Netzdesign bestimmt, und die Methodenaufrufe sind nicht an wenige bestimmte, genau definierte Objektzustände gekoppelt. Verschiedene Aktionen laufen parallel ab, dabei kann die Rechenzeit der Funktoren verfahrensbedingt und in Abhängigkeit von den zu verarbeitenden Daten, aber auch durch Prozeßwechsel und schwankende Ressourcenauslastung von Zyklus zu Zyklus variieren. Welche Operationen dabei zuerst beendet werden, in welcher Reihenfolge nachfolgende Operationen aufgerufen werden und welchen Status schließlich die beteiligten Objekte beim Aufruf einer Operation haben, ist anders als bei der synchronen Ablaufsteuerung daher nicht deterministisch. Das bedeutet, daß während jeder laufenden Aktion beliebige neue Methoden aufgerufen werden können.

Um die Konsistenz der Daten, aber auch der Programmsteuerung sicherzustellen, sind spezielle Mechanismen notwendig, die zum einen verhindern, daß sich verschiedene Aktionen gegenseitig behindern, die zum anderen aber auch eine Aufrufreihenfolge garantieren, die die konsistente Steuerung des Datenflußgraphen erlaubt. Im einzelnen müssen dabei die folgenden Konsistenzforderungen beachtet werden:

Datenkonsistenz: Während der Modifikation einer Gruppe zusammengehöriger Daten dürfen keine weiteren Zugriffe auf diese Daten stattfinden.

Zustandskonsistenz: Die Auswahl einer Aktion muß auf einer konsistenten Analyse des Objektzustandes erfolgen, wofür diese nicht durch andere Zugriffe auf die Zustandsgrößen unterbrochen werden darf.

Verklemmungsfreiheit: Aktionen dürfen sich nicht wechselseitig blockieren und so zu Verklemmungen führen.

Erreichbarkeit: Der Abbruch einer Aktion darf nicht zur Folge haben, daß diese im aktuellen Zyklus überhaupt nicht ausgeführt wird.

Es lassen sich noch zwei weitere Konsistenzkriterien formulieren, auf deren strikte Einhaltung hier jedoch verzichtet werden kann. Das liegt zum einen daran, daß auf der einen Seite ein Verstoß dagegen nur sehr unwahrscheinlich ist und zudem leicht erkannt werden kann, andererseits aber das garantierte Einhalten der Forderung einen sehr hohen Aufwand — etwa einem lokalen Scheduler für jedes Objekt oder den Einsatz eines Echtzeitbetriebssystems — bedeuten würde. Sie werden damit als schwache Konsistenzforderungen formuliert, wobei Verstöße gegen sie durch einen Statustest erkannt werden und eine entsprechenden Reaktion nach sich ziehen können.

Zykluskonsistenz: Zyklen sollten möglichst in der richtigen zeitlichen Reihenfolge abgearbeitet werden, d.h. ein Zyklus, der deutlich weniger Rechenzeit benötigt als der vorhergehende, sollte diesen nicht „überholen“: Prinzipiell erlaubt das Sequenzmodell zwar auch das Einfügen älterer Werte in die Liste, unter Umständen kann das allerdings zu inkonsistenten Zugriffen auf alte Werte führen.

Entscheidungskonsistenz: Auch wenn unmittelbar auf die auf dem aktuellen Objektstatus basierende Auswahl einer internen Aktion ihr Aufruf folgen sollte, kann es vorkommen, daß zwischenzeitlich eine andere Operation auf dem Objekt ausgeführt wird und dabei deren Objektzustand verändert. Hierbei sind zwei mögliche Problemfälle zu unterscheiden: Ein Problem ergäbe sich, wenn eine notwendige und zuvor abgetestete Vorbedingung für den erfolgreichen Methodenaufruf — insbesondere die Verfügbarkeit bestimmter Daten — nach dem Ende der anderen Operation nicht mehr erfüllt wäre und dadurch zur Auswahl einer anderen Aktion geführt hätte. Dies kann jedoch aufgrund des verwendeten Datensequenzmodells ausgeschlossen werden. Ein zweites Problem besteht in der Möglichkeit, daß der Methodenaufruf aufgrund einer zweiten, zwischendurch ausgeführten Aktion überflüssig geworden ist. Dem kann einfach mit einem erneuten Statustest zu Beginn der internen Aktion, innerhalb eines geschützten Bereiches und damit sicher vor erneuten Unterbrechungen begegnet werden.

Um diese Ziele zu erreichen, werden verschiedene Steuerungsmechanismen in einem allgemeinen Methodenmodell zusammengeführt. Neben passiven Mitteln wie dem Schutz bestimmter Daten oder Programmbereiche durch Semaphore spielt dabei vor allem die aktive Analyse des Zustands der an einer Aktion beteiligten Objekte eine wichtige Rolle: In Abhängigkeit vom Ergebnis dieser Statusanalyse kann eine Aktion explizit abgebrochen oder mit einer internen Methode fortgesetzt werden, wobei sie im letzten Fall ggf. auf das Ende einer bereits laufenden Aktion zu warten hat. Während Semaphore einfach mit Betriebssystemmitteln umzusetzen sind, kann ein Statustest flexibel auf den Kontext eines Aufrufs und die unterschiedlichsten Randbedingungen eingehen, insbesondere wenn durch veränderte Steuer- und Zugriffsrelationen das Verhalten der Objekte und ihr Verhältnis zueinander verändert wird. Darüber hinaus erfolgt am Anfang der meisten Aktionen aufgrund des zugrundeliegenden Steuerungskonzeptes und der im Abschnitt 6.4.5 besprochenen lokalen Konflikterkennung bereits eine Statusanalyse. Sie dient beispielsweise dazu, die Aktualität der beteiligten Objekte zu überprüfen und in Abhängigkeit davon eine bestimmte Aktion auszuwählen.

In Abb. 6.28 wird ein einheitliches Modell für die Bearbeitung von komplexeren Interfacemethoden M der relevanten Graphenobjekte $\mathbf{O} : \mathbf{O} \in \{\mathbf{S}\} \cup \{\mathbf{F}\}(\cup\{\mathbf{A}\})$ und damit eine einheitliche Vorgehensweise für eine bestimmte Klasse von Methoden beschrieben. Eine Methode beginnt diesem Modell entsprechend mit einer Zustandsanalyse, in der anhand verschiedener Statusgrößen wie Zugriffs- und Objektzeit sowie anhand der aktuell laufenden oder in diesem Zyklus bereits abgeschlossenen Methoden entschieden wird, ob der Aufruf — weil er nicht mehr nötig oder noch nicht möglich ist — einfach abzubrechen ist, ob ein Fehlerfall, d.h. ein Konflikt entsprechend Abschnitt 6.4.5 vorliegt, der eine Ausnahmebehandlung erfordert, oder ob mit einer internen Methode fortgefahren werden soll. Allgemein bedeutet das: über einen logischen Ausdruck $\mathbf{C}^M$ kann für jede Methode eine Abbruchbedingung und mit $\mathbf{C}^M_{\mathbf{Err}}$ für lokal erkennbare Konflikte eine Ausnahmebehandlung formuliert werden. Sollte ein Abbruch an dieser Stelle nicht vorgesehen oder der Konflikt lokal nicht erkennbar sein, können $\mathbf{C}^M$ und $\mathbf{C}^M_{\mathbf{Err}}$ auch konstant auf 'false' gesetzt werden.

Die gesamte Zustandsanalyse wird über ein objektweit gültiges Statussemaphor $\zeta^{State}_{\mathbf{O}}$ geschützt, welches verhindert, daß zwischen Test und Auswahl einer Aktion eine andere Methode die Kontrolle erlangt und den Zustand verändert. Sind weder $\mathbf{C}^M$ noch $\mathbf{C}^M_{\mathbf{Err}}$ erfüllt, wird entsprechend der eigentlichen Aufgabe der Aktion eine interne Methode M_X ausgewählt, dies im Status vermerkt und das Eingangssemaphor $\zeta^{State}_{\mathbf{O}}$ freigegeben. Die internen Operationen

$\mathbf{O}.M([t,]\ldots)$: **Aufruf der Methode M, ggf. mit t als Parameter**

— Beginn des initialen Statustests —
TAKE $\zeta_{\mathbf{O}}^{State}$; Statustest exklusiv betreten
IF $(\mathbf{C}^{M} = \mathsf{true})$: Ist die allgemeine Abbruchbedingung erfüllt?
  $A[M,t] := \mathsf{canceled}$; Statusgrößen anpassen
  FREE $\zeta_{\mathbf{O}}^{State}$; Geschützten Bereich für Statustest wieder freigeben
  RETURN $\emptyset$; Methode abbrechen

IF $(\mathbf{C}_{\mathbf{Err}}^{M} = \mathsf{true})$: Liegt ein Konflikt vor?
  $A[M,t] := \mathsf{error}$; Statusgrößen anpassen
  FREE $\zeta_{\mathbf{O}}^{State}$; Geschützten Bereich für Statustest wieder freigeben
  EXCEPTION $(\mathbf{O}, M, t)$; Methode mit Ausnahmebehandlung abbrechen

$A[M_X,t] := \mathsf{started}$; $(t_{\mathsf{c}} := t;)$ Statusgrößen anpassen: geplante Aktion, Zykluszeit
FREE $\zeta_{\mathbf{O}}^{State}$; Statustest beenden und geschützten Bereich wieder freigeben
— Ende des initialen Statustests —

$\mathbf{O}.M_X\,([t,]\ldots)$: **Aufruf der internen Methode M_X**

  TAKE $\zeta_{\mathbf{O}}^{X}$; Interne Methode M_X gegen mehrfaches Betreten schützen
  (TAKE $\zeta_{\mathbf{O}}^{State}$;) ggf. Statustest schützen
  IF $(\mathbf{C}^{X} = \mathsf{true})$: Ist die interne Abbruchbedingung erfüllt?
    $A[M_X,t] := \mathsf{canceled}$; Statusgrößen anpassen
    (FREE $\zeta_{\mathbf{O}}^{State}$;) FREE $\zeta_{\mathbf{O}}^{X}$; Statustest und Methode freigeben
    RETURN $\emptyset$; Methode abbrechen

  IF $(\mathbf{C}_{\mathbf{Err}}^{X} = \mathsf{true})$: Liegt ein interner Konflik vor?
    $A[M_X,t] := \mathsf{error}$; Statusgrößen anpassen
    (FREE $\zeta_{\mathbf{O}}^{State}$;) FREE $\zeta_{\mathbf{O}}^{X}$; Statustest und Methode freigeben
    EXCEPTION $(\mathbf{O}, M, t)$; Methode mit Ausnahmebehandlung abbrechen

  $A[M_X,t] := \mathsf{running}$; Statusgrößen anpassen
  (FREE $\zeta_{\mathbf{O}}^{State}$;) ggf. Statustest freigeben

  — Beginn der Bearbeitung der eigentlichen Methodenoperationen —
    $\vdots$
  (TAKE $\zeta_{\mathbf{O}}^{\chi}$;) ggf. Datenbereich schützen
    $\vdots$ Datenzugriff
  (FREE $\zeta_{\mathbf{O}}^{\chi}$;) ggf. Datenbereich wieder freigeben
    $\vdots$
  — Ende der eigentlichen Methodenoperationen

  $A[M_X,t] = \mathsf{finished}$; Statusgrößen anpassen
  FREE $\zeta_{\mathbf{O}}^{X}$; Geschützten Methoden-Bereich wieder freigeben

RETURN ...; Methode beenden

Abbildung 6.28: Einheitliches Methodenmodell für Objekte in Datenflußgraphen.

können nun wiederum durch ein Semaphor $\zeta_{\mathbf{O}}^{X}$ geschützt werden, das unmittelbar nach der Freigabe von $\zeta_{\mathbf{O}}^{State}$ gesetzt wird. Operationen, die auf geschützte Datenbereiche χ zugreifen, können darüber hinaus durch Datensemaphore $\zeta_{\mathbf{O}}^{\chi}$ geklammert werden. Für interne Methoden,

deren Aufgabe sich im wesentlichen darauf beschränkt, einen solchen Datenzugriff zu kapseln, kann das Datensemaphor $\zeta_{\mathbf{O}}^{\chi}$ zugleich auch die Rolle des Methodensemaphors $\zeta_{\mathbf{O}}^{X}$ übernehmen: $\zeta_{\mathbf{O}}^{X} \equiv \zeta_{\mathbf{O}}^{\chi}$.

Bevor nun die internen Operationen in M_X ausgeführt werden, ist unter Umständen eine erneute Zustandsanalyse mit $\mathbf{C}^{x}$ und $\mathbf{C}_{\mathbf{Err}}^{x}$ notwendig. In dieser können die Ergebnisse eines vorangegangenen Aufrufs der gleichen Methode ${M_X}^{(i-1)}$ berücksichtigt werden, die während der initialen Statusanalyse noch lief und auf deren Ende der aktuelle Aufruf hatte warten müssen. Diese Tests können ebenfalls zum Abbruch der (Gesamt-) Methode führen, insbesondere dann, wenn der Aufruf durch die vorhergehende Methode überflüssig geworden sein sollte. Komplexere Zustandstests (vor allem im Zusammenhang mit der Abfrage anderer Objektzustände) können wie die initiale Zustandsanalyse durch das Statussemaphor $\zeta_{\mathbf{O}}^{State}$ geschützt werden, um sicherzustellen, daß Zugriffe auf die Zustandsgrößen zu einem definierten Zeitpunkt erfolgen. Zum Abschluß der Methode wird der Objektzustand entsprechend der Ergebnisse der Operationen angepaßt und die Methode wieder freigegeben.

Zerfällt eine Aktion intern in mehrere Abschnitte M_X, M_Y, ... (wie z.B. die set()-Methode in die Phasen 'Anpassen der internen Datenliste': add() und 'Datenpropagation': post()), kann jeder Teil separat ($\zeta_{\mathbf{O}}^{X}$, $\zeta_{\mathbf{O}}^{Y}$, ...) geschützt werden. Da der Wechsel von einem geschützten Bereich in einen anderen keine unteilbare Operation ist, und auch nicht durch ein Semaphor geschützt werden kann — Objekte, die auf den nächsten Bereich warten, würden den vorhergenden nicht freigeben und dadurch möglicherweise andere Aktionen blockieren — könnte an dieser Stelle (wie auch schon nach dem Statustest) eine Methode durch eine andere, die später gestartet wurde, überholt werden.

In den folgenden Abschnitten wird insbesondere im Zusammenhang mit der asynchronen Modulsteuerung dieses Methodenmodell eine wichtige Rolle spielen. Für die verschiedenen Steuerungskonzepte ist dabei zu untersuchen, wie die Abbruchbedingungen und die Auswahl der internen Operationen zu gestalten sind, um eine konsistente, konfliktfreie Programmsteuerung zu erhalten.

6.4.7 Datengetriebene Modulsteuerung

Das in diesem Abschnitt beschriebene Konzept für die Steuerung von Datenflußgraphen basiert ausschließlich auf Mechanismen, die eine Weitergabe der Programmsteuerung vorwärts, d.h. in Richtung des Datenflusses erlauben (vgl. Abb. 6.15 a). Erreicht wird dies durch die Verwendung von Relationen, die in Datenflußrichtung ($\mathbf{S} \overset{\rightarrow}{\times} \mathbf{F}$ und $\mathbf{F} \overset{\rightarrow}{\times} \mathbf{S}$) die Abgabe der Kontrolle ermöglichen, in Gegenrichtung ($\mathbf{S} \overset{\leftarrow}{\times} \mathbf{F}$ und $\mathbf{F} \overset{\leftarrow}{\times} \mathbf{S}$) jedoch nicht.[2] Dadurch wird die Menge der hierbei eingesetzten Relationen von den in Abschnitt 6.3.3 beschriebenen auf die folgenden, sowie die von ihnen abgeleiteten Relationen eingeschränkt:

Datenpropagation — $\mathbf{S} \overset{\rightarrow}{\times} \mathbf{F} : (\mathsf{trig})$: Triggern von Funktoren durch ihre Eingangsdatensequenzen, nachdem diese aktualisiert wurden. Beim Triggern wird die Steuerung an den getriggerten Funktor abgegeben.

[2] Eine „Abgabe der Kontrolle" bedeutet hier den Aufruf von Methoden, die ihrerseits wiederum Methoden von anderen Objekten aufrufen, wobei sich unter Umständen verschiedene Alternativen für die Fortführung des Kontrollflusses ergeben. Beim bloßen Abfragen eines Wertes wird die Kontrolle jedoch nicht abgegeben, da die Programmsteuerung in jedem Fall zum aufrufenden Objekt zurückkehrt.

Datenaktualisierung — $\mathbf{F} \overset{\rightarrow}{\times} \mathbf{S} : (\mathsf{set})$: Eintragen der durch einen Funktor berechneten Datenwerte in die entsprechenden Ausgangsdatensequenzen; diese Relation folgt immer zwingend aus dem Datenfluß. Die Steuerung wird an die zu aktualisierende Sequenz übertragen, wobei sich echte Alternativen für den Kontrollfluß nur dann ergeben, wenn die Datensequenz nachfolgende Funktoren triggert.

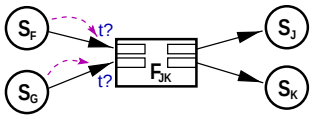
Passive Datenzugriffe — $\mathbf{F} \overset{\leftarrow}{\times} \mathbf{S} : (\mathsf{get}_{[\,]}), (\mathsf{get}_{[\mathrm{t}]}), (\mathsf{get}^{?}_{\mathrm{t}}), (\mathsf{get}^{\bowtie}_{\mathrm{t}})$: Rein anfragende Datenzugriffe ohne Abgabe der Programmsteuerung an die Eingangsdatensequenz; als Standardzugriff soll die Relation $(\mathsf{get}^{?}_{\mathrm{t}})$ verwendet werden, da sie sicherstellt, daß Funktoren erst dann, wenn alle Eingangsdaten aktuell sind, abgearbeitet werden. Wartende Zugriffe mit $(\mathsf{get}^{\bowtie}_{\mathrm{t}})$ sind nur im Zusammenhang mit in einem unabhängigen Kontrollfluß bereitgestellten Eingangsdaten sinnvoll.

Interpolation/Prädiktion — $\mathbf{F} \overset{\leftarrow}{\times} \mathbf{S} : (\mathsf{get}^{\sim}_{\mathrm{t}}), (\mathsf{get}^{\widetilde{\bowtie}}_{\mathrm{t}})$: Im Falle der gemeinsamen Verarbeitung nicht synchron aufgenommener Eingangsdaten durch einen Funktor kann es notwendig sein, die Datenwerte einer Sequenz durch Interpolation zeitlich an die der anderen anzupassen. Auch hier wird die Kontrolle nicht abgegeben, allerdings kann $(\mathsf{get}^{\widetilde{\bowtie}}_{\mathrm{t}})$ gegebenfalls vor einer Interpolation auf die Aktualisierung der Sequenz warten.

Relationen, die die Programmsteuerung im Datenflußdiagramm rückwärts, d.h. entgegen dem Datenfluß weitergeben, werden an dieser Stelle nicht verwendet. Es handelt sich dabei um:

- die von einer Sequenz aktiv betriebene Datenaktualisierung (upd) und
- insistente Datenzugriffe $(\mathsf{get}^{\triangleleft}_{\mathrm{t}}), (\mathsf{get}^{\widetilde{\triangleleft}}_{\mathrm{t}})$.

Aus diesen Einschränkungen folgt, daß die zu setzenden Relationen sich, wie gefordert, weitestgehend automatisch aus dem Datenfluß ergeben. Für den Datenzugriff wird mit $(\mathsf{get}^{?}_{\mathrm{t}})$ eine sinnvolle Defaultrelation ausgewählt. Abb. 6.29 zeigt anhand eines einzelnen Funktors die standardmäßig gesetzten Relationen. Möglichkeiten, im Rahmen des Steuerungskonzeptes auf den Kontrollfluß Einfluß zu nehmen, ergeben sich bei der Art des Zugriffs auf die Eingangsdaten, durch die Möglichkeit einen Funktor nicht durch alle seine Eingangsdaten zu triggern sowie hinsichtlich des Grades der Parallelisierung, der sich mit Hilfe der synchronen oder asynchronen Varianten der gegebenen Relationen (trig) und (set) spezifizieren läßt.



$$\{\mathbf{S_J}, \mathbf{S_K}\} = \mathbf{F_{JK}}\big(\,\mathbf{S}^{?}_{\mathbf{F}}(t), \mathbf{S}^{?}_{\mathbf{G}}(t)\,\big)_{\leftarrow \mathbf{S_F}, \mathbf{S_G}};$$

$$\begin{array}{ll} \mathbf{S_F}\ (\mathsf{trig})\ \mathbf{F_{JK}} & \mathbf{F_{JK}}\ (\mathsf{get}^{?}_{\mathrm{t}})\ \mathbf{S_F} \\ \mathbf{S_G}\ (\mathsf{trig})\ \mathbf{F_{JK}} & \mathbf{F_{JK}}\ (\mathsf{get}^{?}_{\mathrm{t}})\ \mathbf{S_G} \\ & \mathbf{F_{JK}}\ (\mathsf{set})\ \mathbf{S_J} \\ & \mathbf{F_{JK}}\ (\mathsf{set})\ \mathbf{S_K} \end{array}$$

Abbildung 6.29: Standardrelationen für ein rein vorwärts gesteuertes Datennetz.

Beim Parallelisierungsgrad der Ablaufsteuerung wird von den synchronen bzw. sequentiellen Relationen $(\mathsf{trig}_{\circ})$ und $(\mathsf{set}_{\circ})$ als Default ausgegangen, da hier die Zusammenhänge leichter zu modellieren sind. Anschließend werden die Möglichkeiten und Konsequenzen, die sich

aus einem Übergang zu den asynchronen bzw. parallelen Relationen $(\mathsf{trig}_{\|})$ und $(\mathsf{set}_{\|})$ ergeben, untersucht.

Die beteiligten Objekte — Datensequenzen und Funktoren — können entsprechend der Abschnitte 4.2.2 und 4.3.5 die folgenden Aktionen ausführen:

Sequenzen:

- Setzen eines neuen Wertes mit $\mathsf{set}()$. Dabei werden zuerst die internen Sequenzdaten angepaßt: $\mathsf{add}()$. Im Anschluß daran kann der neue Datenwert im Netz propagiert werden: $\mathsf{prop}()$, wofür die von der Sequenz abhängigen Funktoren $\{\mathbf{F_{dep}}\}^n$ getriggert werden.
- Lesen eines Elements der Sequenz mit und ohne vorherigen Test der Aktualität der Sequenz: $\mathsf{get}_t^?()$ und $\mathsf{get}_{[]}()$; $\mathsf{get}_t^{\bowtie}()$ wartet ggf. zuvor auf die Aktualisierung der Sequenz.
- Interpolieren oder Prädizieren von Sequenzwerten mit $\mathsf{get}_t^{\sim}()$, auch hier kann zuvor auf die Aktualisierung der Sequenz gewartet werden: $\mathsf{get}_t^{\widetilde{\bowtie}}()$. Intern stützen sich beide Methoden auf $\mathsf{int}()$ ab.

Funktoren:

- Von außen kann ein Funktor über die $\mathsf{call}()$-Methode von seinen Eingangsdatensequenzen gestartet werden. Diese stützt sich auf die folgenden internen Aktionen ab:
  - Sammeln der Eingangsdaten mit $\mathsf{pre}()$;
  - Ausführen der eigentlichen Funktoroperatoren mit $\mathsf{do}()$;
  - Setzen der Ausgangsdaten mit $\mathsf{post}()$.

Synchrone Ablaufsteuerung

Die synchrone Modulsteuerung ist dadurch gekennzeichnet, daß der Kontrollfluß sich nicht teilt und alle Aktionen eines Zyklus nacheinander in *einem* Programmfaden ausgeführt werden. Damit müssen auch alle Eingangsdaten, durch die ein Teilnetz getriggert werden kann, in ein und demselben *Thread* laufen. Die Reihenfolge der Aufrufe ergibt sich dabei im wesentlichen direkt aus dem Netzdesign. Sie ist deterministisch und unabhängig von der Rechenzeit der verschiedenen Aktionen.

In Abb. 6.30 wird der Kontrollfluß für einen Funktor und die mit ihm benachbarten Eingangs- und Ausgangsdatensequenzen (vgl. Abb. 6.29) als Sequenzdiagramm[3] dargestellt. Ausgangspunkt des hier betrachteten Ausschnitts des Kontrollflusses ist die Aktualisierung der Sequenz $\mathbf{S_F}$ durch den Aufruf von $\mathsf{set}(\mathbf{S_F}, t_g(S), S)$. Nach dem Eintragen des neuen Datums in die interne Sequenzwertliste mit $\mathsf{add}(\mathbf{S_F}, t_g(S), S)$ wird aufgrund der gesetzten Relation $\mathbf{S_F}\ (\mathsf{trig})\ \mathbf{F_{JK}}$ der Funktor $\mathbf{F_{JK}}$ angestoßen, wobei die Datenmeßzeit des neuen Sequenzwertes als Triggerzeit

[3] Sequenzdiagramme sind ein Mittel des objektorientierten Entwurfs zur Beschreibung von Kontrollflüssen zwischen interagierenden Objekten. Der Begriff „Sequenzdiagramm“ ist daher unabhängig von den hier dargestellten Sequenzobjekten zu sehen. Die hier verwendete Notation orientiert sich an der UML [FS97, BJR98]. Die in dieser Abbildung gewählte ausführliche Darstellungsform wird in den folgenden Diagrammen auf wesentliche Aspekte reduziert, um eine übersichtlichere und kompaktere Darstellung mit weniger Redundanz zu erhalten. So wird beispielsweise zukünftig auf die explizite Darstellung der Selbstdelegation (Aufruf der $\mathsf{prop}()$-, $\mathsf{pre}()$-, $\mathsf{do}()$- und $\mathsf{post}()$-Methoden), auf die Angabe von Bedingungen für den Aufruf oder Abbruch bestimmter Aktionen sowie auf Methodenparameter verzichtet.

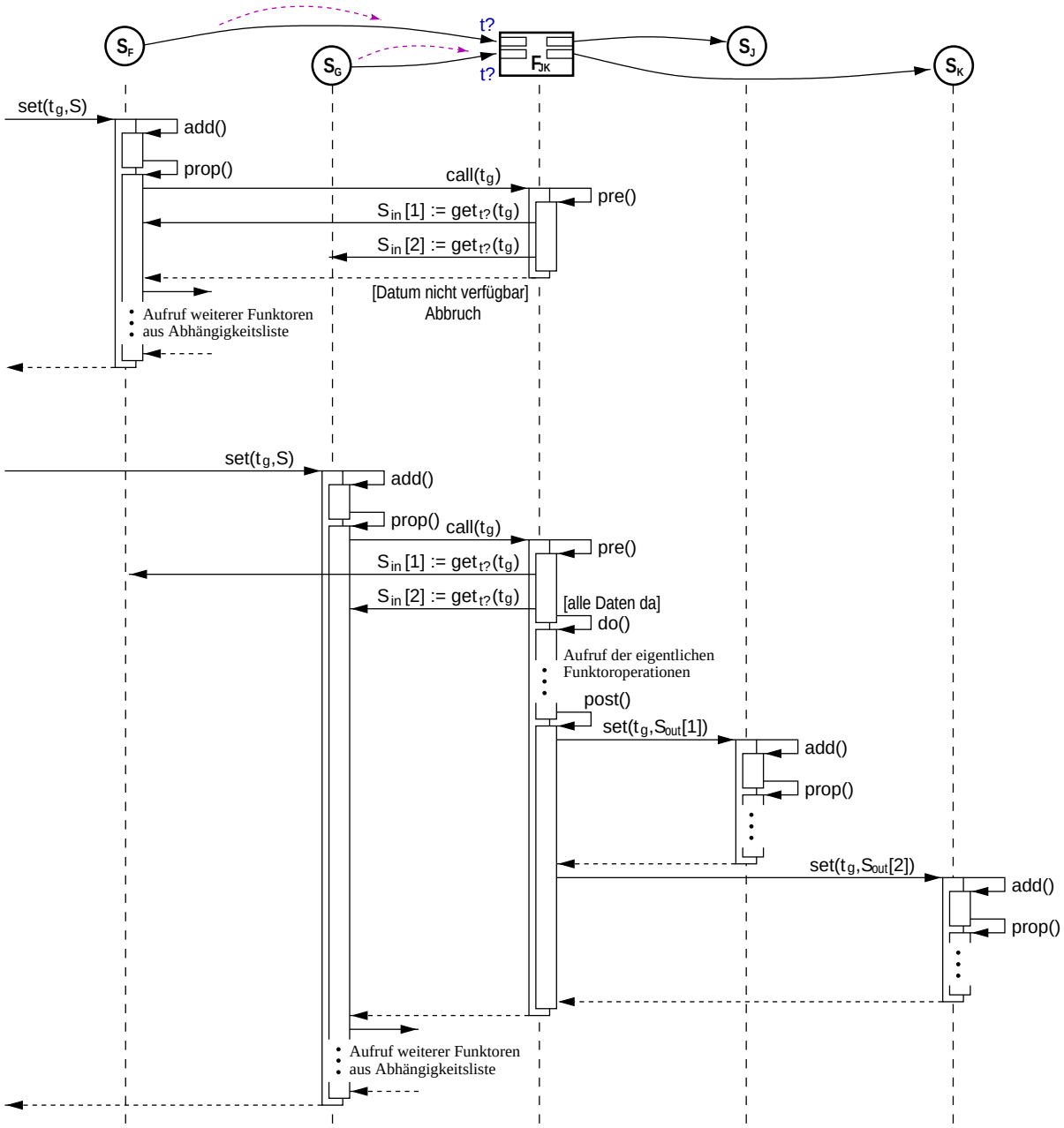


Abbildung 6.30: Ausführliches Sequenzdiagramm zur Darstellung des Kontrollflusses eines synchron vorwärts gesteuerten Funktors.

verwendet wird: $\mathsf{call}(\mathbf{F_{JK}}, t_\mathrm{g}(S_F^0))$. Der Funktor versucht nun, seine Eingangsdaten mit den aktuellen Werten der entsprechenden Datensequenzen $\mathbf{S_F}$ und $\mathbf{S_G}$ zu besetzen. Für $S_{In_1} = \mathbf{S_F}(t_\mathrm{g})$ gelingt dies, da $\mathbf{S_F}$ im aktuellen Zyklus bereits aktualisiert wurde. $S_{In_2} = \mathbf{S_G}(t_\mathrm{g})$ hingegen wurde in diesem Szenario noch nicht aktualisiert, weswegen der Funktoraufruf scheitert und abgebrochen wird. Die Kontrolle kehrt zur Sequenz $\mathbf{S_F}$ zurück, die nun, falls vorhanden, weitere Folgefunktoren aufruft. Erst nachdem diese bearbeitet oder wie $\mathbf{F_{JK}}$ aufgrund fehlender Eingangsdaten abgebrochen wurden, ist die Aktualisierung von $\mathbf{S_F}$ beendet und der Kontrollfluß geht an den aufrufenden Funktor zurück.

Wird nun die Sequenz $\mathbf{S_G}$ ebenfalls mit der Datenzeit t_g aktualisiert, wiederholt sich der Aufruf von $\mathbf{F_{JK}}$. Da jetzt alle Eingangsdaten zur Verfügung stehen, kann der Funktor seine intern gekapselten Operationen in der do()-Methode ausführen und anschließend in post() die von ihm berechneten Ausgangsdaten in die entsprechenden Sequenzen schreiben. Dies geschieht mit der set()-Methode nacheinander für alle Ausgangsdaten, wobei sich für jede Ausgangssequenz das hier beschriebene Prozedere wiederholt, d.h. die Sequenz bekommt die Kontrolle übergeben, ruft eventuell vorhandene Folgefunktoren auf, wartet auf deren Abarbeitung und gibt die Kontrolle an $\mathbf{F_{JK}}$ zurück. Sind alle Ausgangsdatensequenzen aktualisiert, gibt der Funktor die Kontrolle an $\mathbf{S_G}$ zurück. Sind weitere Folgefunktoren vorhanden, werden diese in der gleichen Form abgearbeitet, und schließlich wird die Aktualisierung der Sequenz $\mathbf{S_G}$ beendet.

In diesem Diagramm wird davon ausgegangen, daß $\mathbf{S_F}$ das erste Eingangsdatum von $\mathbf{F_{JK}}$ ist und vor $\mathbf{S_G}$ aktualisiert wird. Es läßt sich jedoch leicht zeigen, daß der Mechanismus ähnlich arbeitet, wenn die Eingabeparameter vertauscht werden, die Aktualisierung der Sequenzen in umgekehrter Reihenfolge erfolgt oder ein Funktor beliebig viele Eingangsdaten besitzt. Auch ein Verschieben des Funktors $\mathbf{F_{JK}}$ in den Listen der von $\mathbf{S_F}$ und $\mathbf{S_G}$ zu triggernden Funktionen $\{\mathbf{F_{dep}}\}^n$ ändert nichts Grundsätzliches an dem aufgezeigten Kontrollfluß.

Es ist ebenfalls offensichtlich, daß Funktoren ohne Ausgangsdaten wie $\{\,\} = \mathbf{F_X}(\mathbf{S_F})$ ohne Probleme in diesen Aufrufmechanismus integriert werden können. Bei ihnen entfällt lediglich die post()-Methode des Funktoraufrufs. Und schließlich fügen sich auch Funktoren, die wie $\mathbf{F_H}$ ein aktuelles Datum zusammen mit älteren Sequenzwerten der gleichen Sequenz verarbeiten: $\mathbf{S_H} = \mathbf{F_H}(\mathbf{S_F^{(0,-1)}})$, ohne weiteres in den Mechanismus ein. Im Rahmen der Aktualisierung von $\mathbf{S_F}$ wird $\mathbf{F_H}$ aufgerufen, und, da alle Eingangsdaten zur Verfügung stehen, kann dieser sofort seine Arbeit aufnehmen. Zu beachten ist dabei, daß nach Abschnitt 5.1.2 die Datenmeßzeit des Ausgangsdatums $\mathbf{S_H}$ zwar standardmäßig das Intervall über alle Eingangsmeßintervalle umfaßt (z.B. $i_g(S_H^0) = [i_g^-(S_F^{-1}), i_g^+(S_F^0)]$), für Gültigkeitstests nachfolgender Funktoren aber nur der Endzeitpunkt dieses Intervalls herangezogen wird: $t_g(S_H^0) = i_g^+(S_F^0) = t_g(S_F^0)$.

Asynchrone Ablaufsteuerung

Kennzeichen einer asynchronen Ablaufsteuerung ist, daß der Kontrollfluß sich an bestimmten Punkten im Programm bzw. im Datenflußgraphen teilt. Verschiedene Programmzweige werden dadurch nebenläufig bearbeitet. Darüber hinaus müssen die Daten- oder Funktionsobjekte nicht auf die Beendigung der von ihnen in anderen Objekten gestarteten Aktionen warten, bevor die eigene Methode abgeschlossen werden kann, d.h. ein Funktor kann so z.B. bereits den nächsten Zyklus beginnen, während nachfolgende Funktoren noch die Daten eines vorhergehenden Zyklus bearbeiten.

Bei der datengetriebenen Modulsteuerung lassen sich mit Hilfe der folgenden Relationen asynchrone Programmzweige spezifizieren: $(\mathsf{set}_{\|})$ kennzeichnet das Setzen eines neuen Sequenzwert in einem eigenen Programmfaden und $(\mathsf{trig}_{\|})$ sorgt für den Aufruf eines nachfolgenden Funktors — ebenfalls in einem eigenen Programmfaden. Einfache Datenzugriffe mit $(\mathsf{get}_t^?)$ und $(\mathsf{get}_{[]})$ werden nicht asynchron definiert, da sie nur einen Wert abfragen, und die Kontrolle sofort zum aufrufenden Funktor zurückkehrt — eine Parallelisierung ist an dieser Stelle nicht notwendig.

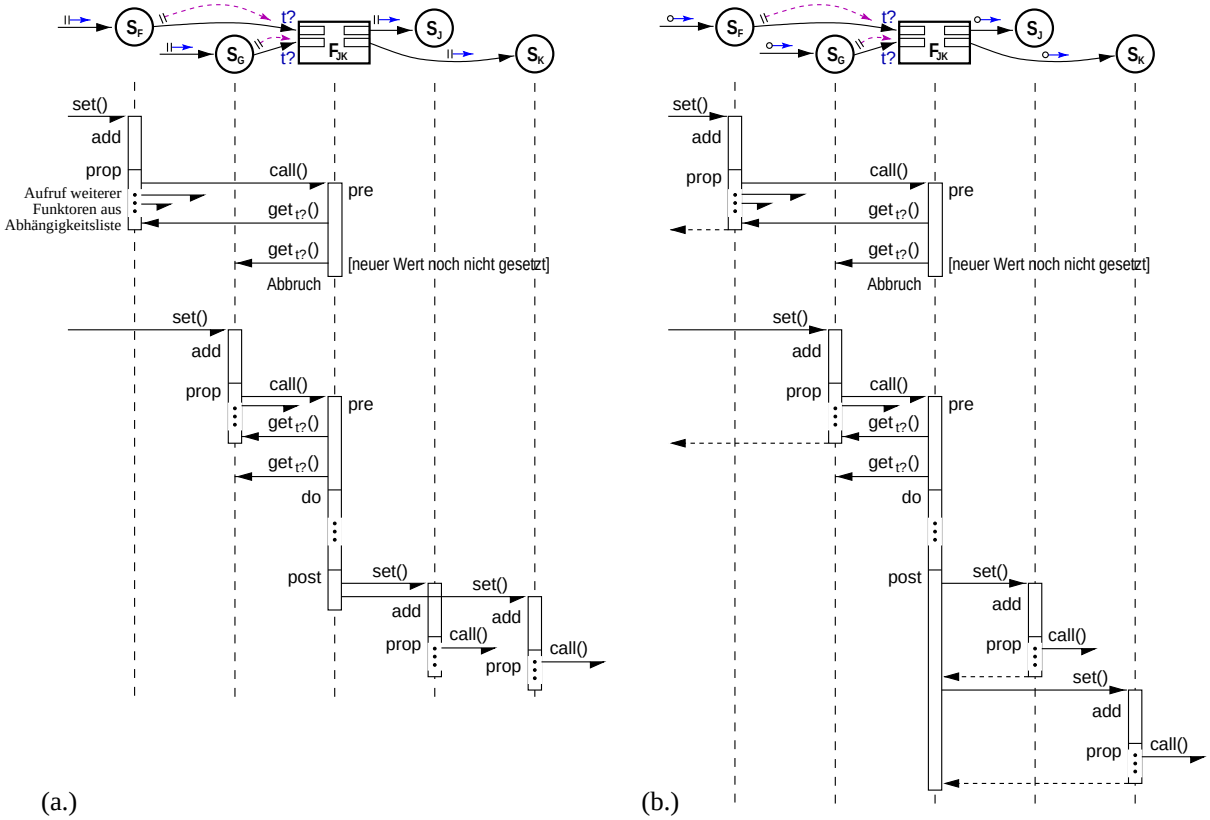


Abbildung 6.31: Mögliches Sequenzdiagramm zur Darstellung des Kontrollflusses eines asynchron vorwärts gesteuerten Funktors: (a.) mit asynchroner $(\mathtt{set}_{||})$- und (b.) mit synchroner $(\mathtt{set}_{\circ})$-Relation.

Abb. 6.31 a zeigt *einen* möglichen asynchronen Kontrollfluß für einen einzelnen Funktor, bei dem sowohl die $(\mathtt{set}_{||})$- als auch die $(\mathtt{trig}_{||})$-Relationen gesetzt sind. Da bei einem asynchronen Aufruf der Folgefunktoren die Dauer der Aktualisierung einer Sequenz im wesentlichen durch die Verwaltungsoperationen für das Einfügen des neuen Datums in die interne Werteliste sowie das reine Anstoßen der Funktoren bestimmt wird, und damit sehr kurz ist, kann auf die asynchrone $(\mathtt{set}_{||})$-Relation auch verzichtet und diese durch die synchrone Variante $(\mathtt{set}_{\circ})$ ersetzt werden. Die Berechnung der Sequenzwerte und ihr Einfügen in die entsprechenden Sequenzen geschieht dann in ein und demselben Programmfaden. Abb. 6.31 b zeigt das dazugehörige Sequenzdiagramm mit einem entsprechenden Kontrollfluß.

Wie im Abschnitt 6.4.6 bereits ausführlich diskutiert wurde, ist die Aufrufreihenfolge der einzelnen Aktionen nicht deterministisch, d.h. die in Abb. 6.31 dargstellten Sequenzdiagramme stellen jeweils nur einen möglichen Kontrollfluß dar. Dabei ist zu beachten, daß ein Funktor auch dann asynchron angesteuert werden kann, wenn die unmittelbar angrenzenden Relationen synchron definiert sind. Um sich mit asynchronen Steuerungsmechanismen befassen zu müssen, reicht es, daß zwei der Eingangsdaten eines Funktors parallel in unterschiedlichen *Threads* bereitgestellt werden.

Die Zeitpunkte der verschiedenen Aktionen können sich beliebig verschieben, so daß sich zum einen die Aufrufreihenfolgen ändern und zum anderen mehrere Aktionen zusammenfallen können. Daher ist im folgenden die Frage zu beantworten, welche Programm- oder Datenbe-

reiche zu schützen sind, um Inkonsistenzen zu vermeiden. Es muß untersucht werden, wo dies mit Semaphoren geschehen kann und unter welchen Bedingungen ein Aufruf abzubrechen ist, um die Konsistenz der Daten und der Programmsteuerung zu gewährleisten.

In einem vorwärtsgesteuerten Netz kann für eine Sequenz eine der folgenden Methoden aufgerufen werden: $\mathsf{get}_t^?()$, $\mathsf{get}_{[]}()$ und $\mathsf{get}_t^\sim()$ — rein lesende Zugriffe ohne Warten auf bestimmte Aktionen (mit und ohne Test sowie interpolierend), $\mathsf{get}_t^{\bowtie}()$ und $\mathsf{get}_t^{\widetilde{\bowtie}}()$ — lesende Zugriffe, auf die Aktualisierung der Sequenz wartend, $\mathsf{set}()$ — Setzen eines neuen Sequenzwertes mit den internen Methoden $\mathsf{add}()$ und $\mathsf{prop}()$. Hierbei sind insbesondere das gleichzeitige Setzen und Abfragen von Werten kritisch, da beim Setzen der internen Daten diese kurzzeitig inkonsistent sein können. Datenzugriffe parallel zur $\mathsf{prop}()$-Methode stellen dagegen kein Problem dar und kommen, wie Abb. 6.30 zeigt, auch bei der synchronen Programmsteuerung vor.

Für Funktoren sind hier parallele Mehrfachaufrufe der $\mathsf{call}()$-Methode zu untersuchen. Diese ruft nacheinander die internen Methoden $\mathsf{pre}()$ (Einlesen der Eingangsdaten), $\mathsf{do}()$ (eigentliche Datenverarbeitung) und post (Ausgangsdaten in die entsprechenden Sequenzen schreiben) auf.

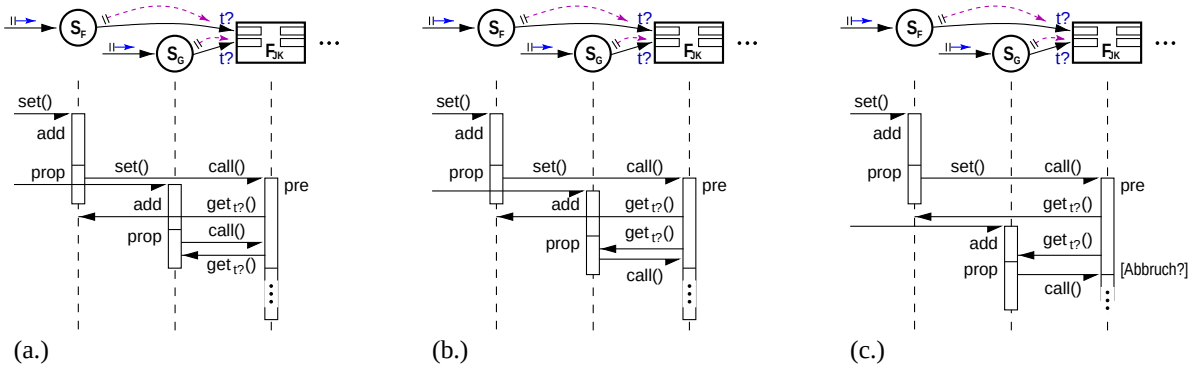


Abbildung 6.32: Parallele Methodenaufrufe durch asynchrone Programmsteuerung.

Abb. 6.32 zeigt mögliche Überschneidungen von Funktionsaufrufen, die durch ein paralleles Setzen der Eingangsdaten im Kontrollfluß aus Abb. 6.31 entstehen können. Im folgenden sollen diese parallelen Aufrufe hinsichtlich möglicher Konsistenzprobleme und Aufrufkonflikte näher untersucht werden. Das Ziel dabei ist, Konsistenzprobleme aufzuzeigen und daraus die notwendigen Aufruf- bzw. Abbruchbedingungen sowie zusätzliche Schutzmaßnahmen zur Lenkung der Kontrollflüsse abzuleiten. Als wichtige Nebenbedingung bei jeder Einflußnahme auf den Kontrollfluß ist die korrekte Funktionsweise der zuvor vorgestellten Steuerungsmechanismen aufrecht zu erhalten. Ausgangspunkt der Untersuchungen ist eine für das Objekt $\mathbf{O}$: $\mathbf{O} \in \{\mathbf{S}\} \cup \{\mathbf{F}\}$ bereits laufende Methode $M()^{(i-1)} \in \{\mathsf{get}_t^?, \mathsf{set}, \ldots, \mathsf{call}\}$ (ggf. in einer internen Methode $M_X()^{(i-1)} \in \{\mathsf{add}, \ldots, \mathsf{post}\}$), zu der zusätzlich eine weitere Methode $M()^{(i)}$ aufgerufen wird.

- $\boldsymbol{M()^{(i-1)} \in \{\mathbf{get}_t^?(), \mathbf{get}_{[]}(), \mathbf{get}_t^\sim(), \mathbf{get}_t^{\bowtie}(), \mathbf{get}_t^{\widetilde{\bowtie}}()\},\ \mathbf{O} = \mathbf{S}:}$ Während ein Funktor aus seiner $\mathsf{pre}()$-Methode heraus auf eine seiner Eingangsdatensequenzen $\mathbf{S}_{\mathbf{In_i}} = \mathbf{S}$ zugreift, kann in einem anderen Programmzweig auf die gleiche Sequenz $\mathbf{S}$ zugegriffen werden, entweder um ebenfalls einen Wert zu lesen, oder um die Sequenz zu aktualisieren. Handelt es sich

dabei ebenfalls um eine rein lesende Methode, ist der Zugriff unkritisch. Interessanter sind die folgenden Konstellationen:

- Beim Auslesen eines Sequenzwertes mit einer $\mathsf{get}()^{(i-1)}$-Methode wird die Aktualisierung der Sequenz mit $\mathsf{set}()^{(i)}$ gestartet. Um Dateninkonsistenzen zu vermeiden, sind die Zugriffe auf die Liste durch ein Semaphor $\zeta_{\mathbf{S}}^{\mathsf{values}}$ zu schützen. Da sich die nicht wartenden Zugriffsmethoden $\mathsf{get}_{\mathsf{t}}^{?}()$ und $\mathsf{get}_{[]}()$ auf den Datenzugriff beschränken, kann für sie Methoden- und Datensemaphor identisch sein: $\zeta_{\mathbf{S}}^{\mathsf{get}} \equiv \zeta_{\mathbf{S}}^{\mathsf{values}}$. Beim wartenden Zugriff $\mathsf{get}_{\mathsf{t}}^{\bowtie}()$ darf dagegen nur der eigentliche Datenzugriff, nicht jedoch der wartende Bereich durch ein Semaphor geschützt werden.
- Wird die $\mathsf{get}_{\mathsf{t}}^{\sim}()$-Methode mehrfach mit der gleichen Interpolationszeit aufgerufen, sollte der Wert nur einmal — durch den ersten Aufruf — berechnet werden. Dafür wird die gesamte Interpolationsmethode durch ein Semaphor $\zeta_{\mathbf{S}}^{\mathsf{int}}$ geschützt. Am Beginn der internen Methode steht dann ein Test, ob der interpolierte Wert bereits berechnet wurde und direkt zurückgeliefert werden kann. Für $\mathsf{get}_{\mathsf{t}}^{\widetilde{\bowtie}}()$ gilt wie für $\mathsf{get}_{\mathsf{t}}^{\bowtie}()$, daß der Bereich, der auf die Aktualisierung der Sequenz wartet, nicht durch ein Semaphor geklammert werden darf.

Für wartende Methoden ist die Gefahr, Programmbereiche zu blockieren, permanent gegeben, weswegen ihre Anwendung besondere Aufmerksamkeit verlangt und nicht für Standardzugriffe verwendet werden sollte. Da bei den anderen Zugriffsmethoden die Steuerung nicht abgegeben wird, besteht für sie nicht die Gefahr, daß sich mehrere Aufrufe gegenseitig blockieren.

• $\boldsymbol{M()^{(i-1)} = \mathbf{set}(),\ (M_x()^{(i-1)} \in \{\mathbf{add}(), \mathbf{prop}()\}),\ \mathbf{O} = \mathbf{S}:}$ Parallel zu einer bereits laufenden Sequenzaktualisierung mit $\mathsf{set}()^{(i-1)}$ kann lesend auf die Sequenz zugegriffen, aber auch ein weiterer Aktualisierungsversuch gestartet werden:

- Die $\mathsf{get}_{[]}()$- und die $\mathsf{get}_{\mathsf{t}}^{\sim}()$-Methode können ohne Probleme parallel zur $\mathsf{set}()$-Methode gestartet werden, lediglich der unmittelbare Sequenzzugriff muß, wie oben bereits gezeigt wurde, geschützt werden.
- Für den Aufruf der $\mathsf{get}_{\mathsf{t}}^{?}()^{(i)}$-Methode ergeben sich zwei Möglichkeiten: Beim initialen Test der Aktualität einer Sequenz kann eine laufende $\mathsf{add}()^{(i-1)}$-Methode (Status $=$ running) berücksichtigt werden oder nicht. Im ersten Fall ist $\mathbf{C}^{\mathsf{get}_{\mathsf{t}}^{?}} = \mathsf{false}$ und $\mathsf{get}_{\mathsf{t}}^{?}()$ wartet auf das Ende von $\mathsf{add}()$, was leicht dadurch zu realisieren ist, daß die interne $\mathsf{get}_{\mathsf{t}}^{?}()$-Methode wie $\mathsf{add}()$ mit dem Semaphor $\zeta_{\mathbf{S}}^{\mathsf{add}} \equiv \zeta_{\mathbf{S}}^{\mathsf{values}}$ geschützt ist. Im zweiten Fall wird, da der Wert nicht unmittelbar zur Verfügung steht, $\mathsf{get}_{\mathsf{t}}^{?}()^{(i)}$ und damit der Funktor $\mathbf{F}$, von dem dieser Aufruf ausging, abgebrochen: $\mathbf{C}^{\mathsf{get}_{\mathsf{t}}^{?}} = \mathsf{true}$. Dabei wird davon ausgegangen, daß nach dem Ende der laufenden Aktualisierung von $\mathbf{S}$ der Funktor $\mathbf{F}$ durch $\mathbf{S}$ erneut getriggert wird: die $\mathsf{prop}()$-Methode wird direkt im Anschluß an die noch laufende $\mathsf{add}()^{(i-1)}$-Methode aufgerufen.

  Trifft ein $\mathsf{get}_{\mathsf{t}}^{?}()^{(i)}$-Aufruf auf eine laufende $\mathsf{set}()^{(i-1)}$-Methode, *bevor* diese ihre interne Methode $\mathsf{add}()^{(i-1)}$ gestartet hat (Status $=$ started), wird er wie ein Zugriff auf eine noch nicht aktualisierte Sequenz abgebrochen, Zugriffe nach Beendigung der $\mathsf{add}()^{(i-1)}$-Methode (Status $=$ finished) werden problemlos — auch parallel zur $\mathsf{prop}()^{(i-1)}$-Methode — beantwortet.

- Erfolgt parallel zu einer bereits laufenden $\mathsf{set}()^{(i-1)}$-Methode ein zweiter $\mathsf{set}()^{(i)}$-Aufruf, kann das eine der folgenden Ursachen haben: Ist $\mathsf{set}()^{(i)}$ an den gleichen Zeitpunkt gebunden, wie $\mathsf{set}()^{(i-1)}$, liegt, da die Zuordnung zwischen Zeitpunkt und Datenwert immer eindeutig sein muß, ein Fehler im Netzdesign vor, auf den mit einer Ausnahmebehandlung reagieren wird ($\mathbf{C}^{\mathsf{set}}_{\mathbf{Err}} = \mathsf{true}$): ein Datum darf in einem Zyklus nicht mehrfach berechnet und im Nachhinein verändert werden. Unterscheiden sich die Zeiten hingegen, liegt entweder der Wert des nächsten oder eines vorhergehenden Zyklus an, was beides kein prinzipielles Problem darstellt. Durch das Methodensemaphor $\zeta^{\mathsf{add}}_{\mathbf{S}}$ wird die anstehende Aktualisierung solange verzögert, bis die laufende fertig ist. Gleiches gilt für die nachfolgende Datenpropagierung, die durch ein Semaphor $\zeta^{\mathsf{prop}}_{\mathbf{S}}$ geschützt wird. Zu Verklemmungen könnte es durch diese Semaphore nur kommen, wenn die $\mathsf{set}()$- und $\mathsf{prop}()$-Methoden Bestandteil von Schleifen bei der Zyklusbearbeitung wären, die sind jedoch per Definition verboten (vgl. Abschnitt 6.4.2).

• $M()^{(i-1)} = \mathbf{call}(),\ (M_X^{(i-1)} \in \{\mathbf{pre}(), \mathbf{do}(), \mathbf{post}()\}),\ \mathbf{O} = \mathbf{F}$: Ein Funktor $\mathbf{F}$ kann gleichzeitig von verschiedenen seiner Eingangsdatensequenzen $\mathbf{S}_{\mathbf{In}_i}$ und somit auch mehrfach innerhalb eines Zyklus mit $\mathsf{call}()$ gestartet werden. Dabei sind die folgenden Punkte zu beachten:

- Ein Funktor darf pro Zyklus nur einmal seine $\mathsf{do}()$-Methode aufrufen und so das Ende der $\mathsf{call}()$-Methode erreichen. Ziel der wiederholten Triggerung ist, den Funktor von jeder Änderung der Eingangsdaten zu unterrichten, damit er, sobald alle Eingangsdaten zur Verfügung stehen, $\mathsf{do}()$ starten und seine Aufgabe bearbeiten kann. Ob dies der Fall ist, wird in der Vorbereitungsphase $\mathsf{pre}()$ durch den Zugriff auf die Eingangsdaten $\mathbf{S}_{\mathbf{In}}$ mit deren Methode $\mathsf{get}^{?}_{\mathsf{t}}()$ überprüft. $\mathsf{pre}()$ wird dabei durch ein Semaphor $\zeta^{\mathsf{pre}}_{\mathbf{F}}$ geschützt und steht nach der initialen Statusanalyse ($\mathbf{C}^{\mathsf{call}}$ und $\mathbf{C}^{\mathsf{call}}_{\mathbf{Err}}$) am Anfang des Funktoraufrufs. Die Datenzugriffe mit $\mathsf{get}^{?}_{\mathsf{t}}()$ sind dabei Bestandteil der methodeninternen Statusanalyse $\mathbf{C}^{\mathsf{pre}}$, die bei fehlenden Daten zum Abbruch der ganzen Methode führen kann.

- Da neue Sequenzdaten $S_{in_i}(t)$ immer schon kurz *vor* dem Aufruf der $\mathsf{call}()^{(i)}$-Methode verfügbar sind, ist es möglich, daß bereits der zuvor gestartete Funktoraufruf $\mathsf{call}()^{(i-1)}$ erfolgreich auf alle Eingangsdaten zugreifen konnte und nicht erst, wie in der synchronen Steuerung, der letzte $\mathsf{call}()$-Aufruf im Zyklus. Dadurch besteht die Möglichkeit, daß die zuletzt gestartete Aktion $\mathsf{call}()^{(i)}$ überflüssig ist und abgebrochen werden kann. In Abb. 6.32 a und b wird diese Situation illustriert. In der internen Statusanalyse $\mathbf{C}^{\mathsf{pre}}$ muß somit überprüft werden, ob eine eventuell vorher aktive Vorbereitungsphase $\mathsf{pre}()^{(i-1)}$ erfolgreich abgeschlossen und durch die $\mathsf{do}()^{(i-1)}$-Methode abgelöst wurde. Darüber hinaus kann in der initialen Statusanalyse $\mathbf{C}^{\mathsf{call}}$ bereits getestet werden, ob $\mathsf{do}()$ oder $\mathsf{post}()$ mit der aktuellen Aufrufzeit bereits läuft oder vielleicht sogar schon erfolgreich beendet wurde. In diesem Fall kann der Aufruf noch vor dem Start von $\mathsf{pre}()^{(i)}$ abgebrochen werden.

- Erfolgt der $\mathsf{call}()^{(i)}$-Aufruf während einer laufenden $\mathsf{pre}()^{(i-1)}$-Methode, ergeben sich verschiedene Alternativen für die Fortsetzung der $\mathsf{call}()^{(i)}$-Methode. Hier ist es sowohl möglich, $\mathsf{call}()^{(i)}$ einfach abzubrechen, als auch den Aufruf fortzusetzen um das Ende von $\mathsf{pre}()^{(i-1)}$ und damit dessen Erfolg oder Mißerfolg abzuwarten. War $\mathsf{pre}()^{(i-1)}$ und

somit auch $\mathsf{call}()^{(i-1)}$ nicht erfolgreich, kann dann mit $\mathsf{call}()^{(i)}$ der versuchte Funktoraufruf wiederholt werden.

Eine Analyse möglicher Programmabläufe ergibt, daß dies genau dann notwendig ist, wenn die Sequenz $\mathbf{S_{in_i}}$, die für die aktuelle Triggerung $\mathsf{call}()^{(i)}$ von $\mathbf{F}$ verantwortlich ist, beim Zugriff auf sie aus der $\mathsf{pre}()^{(i-1)}$-Methode heraus noch nicht aktualisiert war. Das heißt aber auch, daß sowohl die Aktualisierung dieser Sequenz als auch der anschließende Aufruf der $\mathsf{call}()^{(i)}$-Methode zwischen dem fehlgeschlagenen Datenzugriff und dem Verlassen der $\mathsf{pre}()^{(i-1)}$-Methode erfolgt sein muß. Dies ist zwar sehr unwahrscheinlich, kann ohne besondere Schutzmaßnahmen, die die Unteilbarkeit von Datenzugriff und -auswertung bewirken, jedoch nicht vollkommen ausgeschlossen werden.

Die Alternative, das Ergebnis der Datenzugriffe abzuwarten und danach entsprechend zu reagieren, würde in einem rein vorwärts gesteuerten Graphen keine Probleme bereiten, da die $\mathsf{pre}()$-Methode mit den rein lesenden Datenzugriffen die Kontrolle nicht abgibt und somit keine Aufrufabhängigkeiten bestehen. Greift der Funktor allerdings auf die Eingangsdaten (oder auch nur auf eines) aktiv zu ($\mathsf{get}_\mathsf{t}^{\triangleleft}()$), kann es bei entsprechendem Netzdesign und ungünstig gesetzten Relationen zu Verklemmungen kommen.

Aus diesem Grund wird hier der erste Weg eingeschlagen und die $\mathsf{call}()^{(i)}$-Methode abgebrochen, wenn beim Statustest der Funktor sich bereits im $\mathsf{pre}()^{(i-1)}$-Aufruf mit der gleichen Zugriffszeit befindet (Status $=$ running). Um das oben beschriebene Problem zu umgehen, wird in der $\mathsf{pre}()$-Methode jeder $\mathsf{get}_\mathsf{t}^{?}()$-Zugriff — vom Aufruf bis zum erfolgreichen Ende oder dem Kennzeichnen eines folgenden Abbruchs in den Statusgrößen (Status $=$ canceled) — mit dem Statussemaphor $\zeta_{\mathbf{F}}^{State}$ geklammert. Erfolgt der Aufruf von $\mathsf{call}()^{(i)}$ mit der Statusabfrage $\mathbf{C^F}$ vor dem Datenzugriff, ist das Datum $S_{in_i}(t)$ beim Zugriff auf die Sequenz auf jeden Fall verfügbar. Wird $\mathsf{call}()^{(i)}$ später aufgerufen, erfolgt die Statusabfrage auf jeden Fall nach dem Datenzugriff und der Anpassung der Statusvariablen, so daß ein eventueller Abbruch der Methode dort bereits vermerkt wurde. Nur in diesem Fall wird die $\mathsf{call}()^{(i)}$-Methode mit $\mathsf{pre}()^{(i)}$ fortgesetzt.

Mögliche Inkonsistenzen im Netzdesign und deren Auswirkungen

In diesem Abschnitt soll untersucht werden, wodurch es zu Inkonsistenzen in der Programmsteuerung kommen kann, ob und wie diese durch eine lokale Konfliktanalyse mit $\mathbf{C_{Err}}$ erfaßt werden können, und schließlich wie sie umgangen werden können. Prinzipiell können zwei unterschiedliche Ursachen für derartige Problem verantwortlich sein:

1. Der Entwickler baut bei dem Versuch, durch Änderungen der Objektrelationen die Steuerung seinen Bedürfnissen anzupassen, einen Fehler ein, so daß der Kontrollfluß nicht wie erwartet alle Objekte erreicht oder von ihm formulierte Abhängigkeiten nicht aufgelöst werden können.

2. Der Datenflußgraph enthält bestimmte Bestandteile oder Teilnetze, die nicht den Annahmen entsprechen, die für die Anwendung der Standardrelationen vorausgesetzt wurden, und der Entwickler versäumt es, durch Angabe zusätzlicher Informationen die Zusammenhänge entsprechend zu spezifizieren, bzw. die Relationen anzupassen.

Die erste Fehlerursache mit allen möglichen Relationenkombinationen umfassend zu untersuchen, würde an dieser Stelle zu weit führen. Auf Konsistenzfehler im Zusammenhang mit aktiven Zugriffsrelationen wird darüber hinaus in Abschnitt 6.4.8 eingegangen. Damit die datengetriebene Modulsteuerung, wie zuvor beschrieben, allein mit den Standardzuordnungen zwischen Datenflußbeziehungen und Steuerungsrelationen funktioniert, und alle Sequenzen einmal je Zyklus aktualisiert werden, muß sichergestellt sein, daß der Graph die folgenden Voraussetzungen erfüllt:

1. Ausgehend von den Eingangsdaten des Moduls ist es notwendig, daß alle Eingangsdaten eines Funktors bereitgestellt werden können, ohne daß dafür der Funktor selbst zuvor aufgerufen werden muß
   $\leadsto$ keine Rückkopplungen;
2. Alle Daten eines Zyklus, die über die Standardrelationen miteinander verknüpft werden, müssen die gleiche Datenmeßzeit t_{g} besitzen
   $\leadsto$ keine Eingangsdaten, die zu unterschiedlichen Zeiten *gemessen* wurden.

Sind diese Bedingungen nicht erfüllt und keine zusätzlichen Informationen verfügbar, kann der Funktor mit den Standardrelationen nicht erfolgreich zu Ende geführt werden, da die pre()-Methode aufgrund eines veralteten Eingangsdatums immer abgebrochen wird, und der Funktor seine do()-Methode nicht erreicht. Zu Verklemmungen würde es dabei zwar nicht kommen, da weder Funktoren noch Datensequenzen auf das Erreichen eines bestimmten Zustands der beteiligten Eingangsdaten warten, die Folge wäre jedoch, daß der Kontrollfluß abreißt und sowohl die Ausgangsdaten des Funktors als auch alle anderen davon abhängigen Daten nicht berechnet werden.

Um diese zwei Sonderfälle — Rückkopplungszweige und asynchrone Eingangsdaten — trotzdem in das Steuerungskonzept integrieren zu können, ist es notwendig, diese beim Erstellen des Datenflußgraphen explizit anzuzeigen, damit dies beim Setzen der Relationen berücksichtigt werden kann. Oder der Entwickler gibt direkt die zu verwendenden Steuerungsrelationen vor. Ohne diese Informationen liegt ein Fehler im Design des Datenflußgraphen vor, der zu dem oben beschriebenen Konflikt führt. Dieser sollte entsprechend Abschnitt 6.4.5 nach Möglichkeit erkannt und lokalisiert, und der Entwickler auf den Designfehler hingewiesen werden.

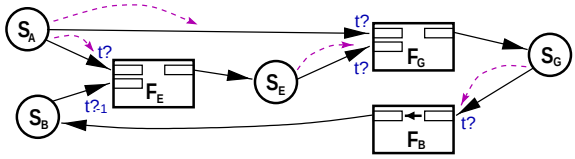
Wird ein Funktor aufgerufen und aufgrund fehlender Eingangsdaten abgebrochen, ist zu diesem Zeitpunkt nicht entscheidbar, ob der Funktor später noch einmal aufgerufen wird, und ob dann alle Daten gültig sein werden. Da allerdings in den Statusvariablen der Abbruch mit Status $=$ canceled vermerkt wurde, der erfolgreiche Abschluß (Status $=$ finished) aber ausblieb, kann das Problem im nächsten Zyklus (oder bei möglicher Pipeliningbearbeitung über eine Reihe von aufeinanderfolgenden Zyklen) lokal erkannt werden. Verklemmungen können, solange nur die Standardrelationen verwendet werden, nicht auftreten.

Rückkopplungszweige im Datenflußgraphen

Eine Rückkopplung muß, wie es anhand der Abb. 6.23 gezeigt wurde, explizit einen Zugriff auf einen Wert des vorhergehenden Zyklus beinhalten. Dieser Rückgriff auf einen alten Wert kann auch dann positiv beantwortet werden, wenn im aktuellen Zyklus der Sequenzwert noch nicht bestimmt wurde. Dafür ist es nicht notwendig, die Standardzugriffsrelation $(\mathrm{get}_{\mathrm{t}}^{?})$ durch eine andere zu ersetzen. Die Entscheidung darüber, ob der Zugriff scheitert oder nicht, wird in der

Zugriffsmethode $\mathsf{get}_t^?()$ anhand des Relativindex des angeforderten Wertes getroffen. Dieser muß für ein altes Datum kleiner Null sein: $S_B^{-1}(t) := \mathsf{get}_t(\mathbf{S_B}, t, -1)$. Die Berechnung des geforderten Wertes erfolgt dann jeweils einen Zyklus zuvor.

Liegt im Rückkopplungszweig mindestens ein Funktor, d.h. der Rückkopplungszweig enthält wie in Abb. 6.34 nacheinander mehrere Datenzugriffe: $\mathbf{F_B}\,(\mathsf{get})\,\mathbf{S_G}$ und $\mathbf{F_E}\,(\mathsf{get})\,\mathbf{S_B}$, muß in diesem Steuerungskonzept der in Datenflußrichtung letzte Zugriff derjenige sein, der den alten Wert verwendet ($\mathbf{S_B^{(-1)}}$ und nicht $\mathbf{S_G^{(-1)}}$). Der Funktor $\mathbf{F_B}$ rechnet dabei mit den aktuellen Daten, seine Ausgabewerte werden jedoch immer erst einen Zyklus später verwendet. Eine alternative Strategie, für die allerdings der Steuerungsmechanismus zu wechseln wäre, ist, den Funktor $\mathbf{F_B}$ erst aufzurufen, wenn der Wert $\mathbf{S_B}$ tatsächlich gebraucht wird. In diesem Fall würden die Eingabedaten von $\mathbf{F_B}$ bei dessen Ausführung aus dem vorhergehenden Zyklus stammen.



$$\{\mathbf{S_E}\} = \mathbf{F_E}(\,\mathbf{S_A^?}(t), \mathbf{S_B^{(-1)}}(t)\,)_{\leftarrow \mathbf{S_A}};$$
$$\{\mathbf{S_G}\} = \mathbf{F_G}(\,\mathbf{S_A^?}(t), \mathbf{S_E^?}(t)\,)_{\leftarrow \mathbf{S_A},\mathbf{S_E}};$$
$$\{\mathbf{S_B}\} = \mathbf{F_B}(\,\mathbf{S_G^?}(t)\,)_{\leftarrow \mathbf{S_G}};$$

$$\mathbf{S_A}\ (\mathsf{trig})\ \mathbf{F_E} \qquad \mathbf{F_E}\ (\mathsf{get}_t^?)\ \mathbf{S_A}$$
$$\mathbf{F_E}\ (\mathsf{get}_t^?)\ \mathbf{S_B^{(-1)}}$$
$$\mathbf{S_A}\ (\mathsf{trig})\ \mathbf{F_G} \qquad \mathbf{F_G}\ (\mathsf{get}_t^?)\ \mathbf{S_A}$$
$$\mathbf{S_E}\ (\mathsf{trig})\ \mathbf{F_G} \qquad \mathbf{F_G}\ (\mathsf{get}_t^?)\ \mathbf{S_E}$$
$$\mathbf{S_G}\ (\mathsf{trig})\ \mathbf{F_B} \qquad \mathbf{F_B}\ (\mathsf{get}_t^?)\ \mathbf{S_G}$$
$$\mathbf{F_E}\ (\mathsf{set})\ \mathbf{S_E}$$
$$\mathbf{F_G}\ (\mathsf{set})\ \mathbf{S_G}$$
$$\mathbf{F_B}\ (\mathsf{set})\ \mathbf{S_B}$$

Abbildung 6.33: Relationen für die konsistente synchrone Vorwärtssteuerung eines rückgekoppelten Datenflußgraphen.

Wird im Datenflußgraphen schließlich ein solcher Zugriff mit negativem Index spezifiziert: $\mathbf{F_E}\,(\mathsf{get}_t^?)\,\mathbf{S_B^{(-1)}}$, kann auch auf das regelmäßige Propagieren der Daten $\mathbf{S_B}$ an den Folgefunktor $\mathbf{F_E}$ und somit auf das Setzen der Steuerrelation (trig) für die dazugehörige Sequenz-Funktor-Relation verzichtet werden ($\mathbf{S_B}\ \cancel{(\mathsf{trig})}\ \mathbf{F_E}$): Damit die Sequenz $\mathbf{S_B}$ überhaupt aktualisiert werden kann, muß der Funktor $\mathbf{F_E}$ bereits erfolgreich ausgeführt worden sein, ein erneuter Aufruf in diesem Zyklus ist daher nicht mehr notwendig. Wird die Relation trotzdem gesetzt, ist dies unkritisch, da der Funktor $\mathbf{F_E}$ erkennt, daß dieser Zyklus schon erfolgreich bearbeitet wurde, woraufhin er den $\mathsf{call}()$-Aufruf bereits im Statustest $\mathbf{C^{F_E}}$ abbricht. Für die mehrstufige Rückkopplung aus Abb. 6.22 zeigt Abb. 6.33 alle gesetzten Relationen, und in Abb. 6.34 wird das dazugehörige Sequenzdiagramm angegeben, das den Kontrollfluß für die synchrone Steuerung dieses Teilnetzes beschreibt.

Das asynchrone Sequenzdiagramm unterscheidet sich nur unwesentlich von dem synchronen, insbesondere die Abarbeitungsreihenfolge der Funktoren bleibt gleich, da sie durch die Datenabhängigkeiten festgelegt ist. Allerdings tritt hier ein potentielles Problem im Zusammenhang mit einer Pipelinigbearbeitung zutage. Beim Aufruf von $\mathsf{call}(\mathbf{F_E})^{(i)}$ wird der letzte Wert von $\mathbf{S_B}$ verwendet. Währenddessen kann aufgrund des Pipelinings der Funktor $\mathbf{F_B}$ noch einen vorhergehenden Zyklus bearbeiten: $\mathsf{call}(\mathbf{F_B})^{(i-1)}$. Das aber heißt, daß der Zugriff auf den alten Wert von $\mathbf{S_B}$, wenn er mit $\mathsf{get}_t^?(\mathbf{S_B}, t, -1)$ erfolgt, nicht den Wert des vorhergehenden Zyklus liefert, sondern einen noch älteren. In einem solchen Fall kann jedoch nur der Entwick-

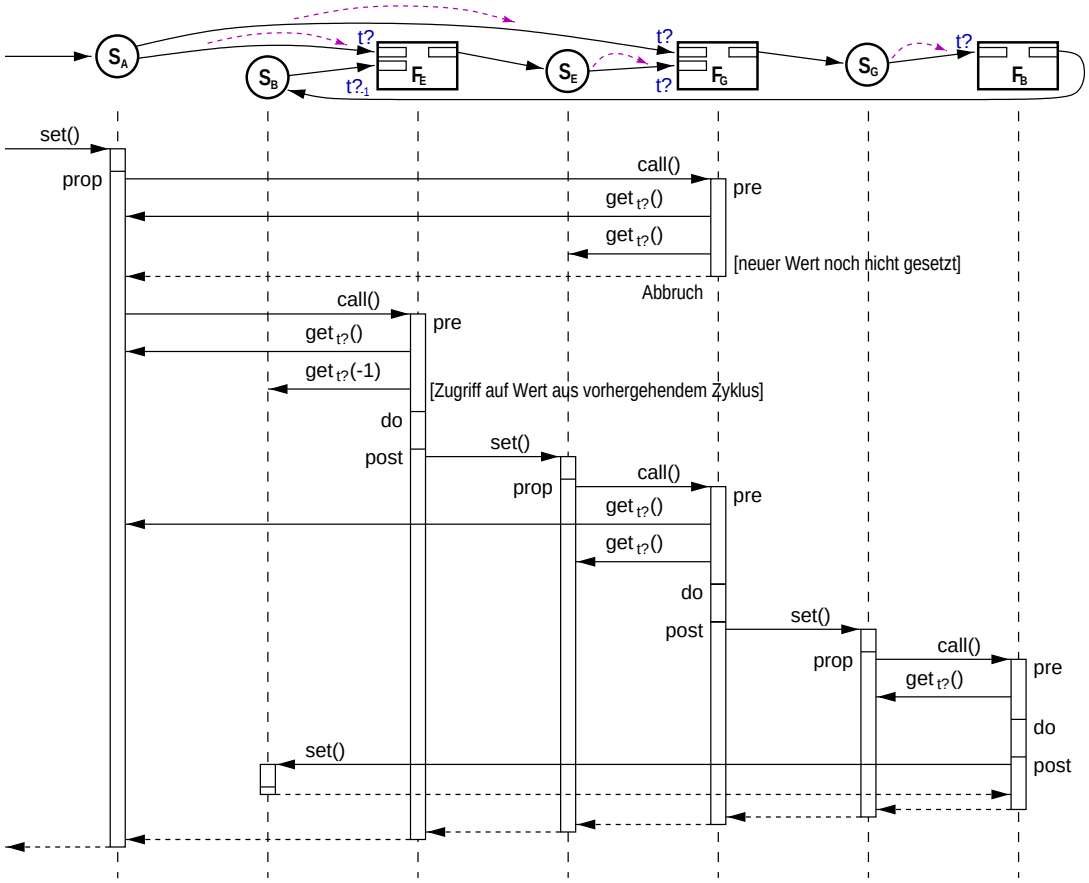


Abbildung 6.34: Sequenzdiagramm mit dem Kontrollfluß eines Datenflußgraphen, der synchron, vorwärts angesteuert wird und einen Rückkopplungszweig enthält.

ler entscheiden, ob dem Pipelining mit möglicherweise etwas älteren Daten oder der möglichst zeitnahen Verarbeitung der Daten aus $\mathbf{S_B}$ ohne Pipelining der Vorzug zu geben ist.

Alternativ zum zuvor beschriebenen Rückgriff auf den letzten vorhandenen Wert der Sequenz ist aber auch eine andere Vorgehensweise möglich: Statt einfach nur einen alten Wert abzufragen, kann aus mehreren alten Sequenzdaten für den aktuellen Zyklus ein neuer Wert prädiziert werden. Ist dieses Verhalten gewünscht, muß dies vom Entwickler explizit so spezifiziert werden. Ausgedrückt wird dies durch die Relation $(\mathrm{get}_t^{\sim})$. Für den Beispieldatenflußgraphen in Abb. 6.33 hätte das die folgende Auswirkung, wobei auch diese Relation die Steuerung nicht abgibt und auch nicht auf die Aktualisierung der Sequenz wartet, sondern unmittelbar nach dessen Berechnung den prädizierten Wert zurückliefert, d.h. das Sequenzdiagramm ändert sich dadurch nicht:

$$\{\mathbf{S_E}\} = \mathbf{F_E}(\,\mathbf{S_A^?}(t), \mathbf{S_B^{\sim}}(t)\,)_{\leftarrow \mathbf{S_A}}; \qquad \text{Relation:} \quad \mathbf{F_E}\ (\mathrm{get}_t^{\sim})\ \mathbf{S_B}$$

Bei der Prädiktion zukünftiger Datenwerte ist allerdings zu beachten, daß sie nicht für beliebige Daten sinnvoll ist, und daher unter Umständen nicht frei im Rückkopplungszweig verschoben werden kann. Das bedeutet, daß es nicht immer möglich ist, sie an das Ende eines Rückkopplungszweiges zu setzen. Dies läßt sich für das Teilnetz in Abb. 6.33 leicht anhand einer Beispielbelegung mit realen Daten aus dem RoboCup zeigen.

In Abb. 6.35 wird dafür ein (vereinfachtes) Teilnetz für die Extraktion und Lokalisation des Balles dargestellt. Zuerst wird aus dem Kamerabild $\mathbf{S_{Img}}$ mit Hilfe von $\mathbf{F_{BExt}}$ die Region bestimmt, in der der Ball zu sehen ist: $\mathbf{S_{BR}}$. Um die Suche zu beschleunigen wird dazu der Suchbereich mit einer *Region-of-Interest* $\mathbf{S_{ROI}}$ eingeschränkt. Aus der Ballregion und dem Eingangsbild wird daraufhin mit $\mathbf{F_{BLoc}}$ die 3D-Position des Balles $\mathbf{S_{BP}}$ bestimmt.

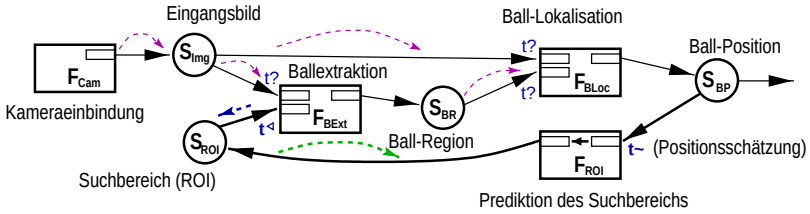


Abbildung 6.35: Vereinfachter rückgekoppelter Datenflußgraph für die Repräsentation einer Bildverarbeitungskomponente aus dem RoboCup, die in den Kamerabildern den Ball lokalisiert.

Die Ballposition, die das Ergebnis dieses Teilgraphen darstellt, bildet die Grundlage für die Bestimmung des Bildbereichs, in dem im nächsten Zyklus nach der Ballregion zu suchen ist. Dafür wird beim Sequenzzugriff mit $\mathsf{get}_t^{\sim}(\mathbf{S_{BP}}, t)$ aus den letzten Werten der Ballposition die neue Position geschätzt. $\mathbf{F_{ROI}}$ projiziert diese daraufhin ins Bild und spannt um diese Bildposition herum den neuen Suchbereich auf.

In diesem Szenario wäre es nun nicht sinnvoll, statt der Balldaten die ROI aus den vergangenen Regionen prädizieren zu wollen, nur damit im Rückkopplungszweig der letzte Datenzugriff — wie für die datengetriebene Steuerung oben gefordert wurde — derjenige ist, der in die Vergangenheit reicht. Darüber hinaus ist in diesem Szenario der durch die Vorwärtssteuerung realisierte Grundsatz, im laufenden Zyklus alle Daten so weit wie möglich zu propagieren und im gleichen Zyklus zu verarbeiten, nicht optimal. Dies würde hier bedeuten, daß im Zyklus i die Prädiktion für den folgenden Zyklus durchgeführt werden müßte, obwohl für diesen im allgemeinen die Zykluszeit noch nicht bekannt ist. Aus diesem Grund wird, wie die Relationen im Datenflußgraphen zeigen, an dieser Stelle der Steuerungsmechanismus gewechselt, und die ROI erst, wenn sie gebraucht wird — d.h. anfragegetrieben, bestimmt. Diese Art der Steuerung wird im Abschnitt 6.4.8 detailliert beschrieben. Sie kann dabei auch auf den Rückkopplungszweig beschränkt bleiben.

Gleichungen:	Relationen:
$\{\mathbf{S_{BR}}\} = \mathbf{F_{BExt}}(\,\mathbf{S}^{?}_{\mathbf{Im}}(t), \mathbf{S}^{\triangleleft}_{\mathbf{ROI}}(t)\,)_{\leftarrow \mathbf{S_{Im}}};$	$\mathbf{F_{BExt}}\ \ (\mathsf{get}_t^{\triangleleft})\ \ \mathbf{S_{ROI}}$
	$\mathbf{S_{ROI}}\ \ (\mathsf{upd})\ \ \mathbf{F_{ROI}}$
$\{\mathbf{S}^{\rightarrow}_{\mathbf{ROI}}\} = \mathbf{F_{ROI}}(\mathbf{S}^{\sim}_{\mathbf{BP}}(t));$	$\mathbf{F_{ROI}}\ \ (\mathsf{get}_t^{\sim})\ \ \mathbf{S_{BP}}$

Verarbeitung nichtsynchronisierter Eingangsdaten

Asynchrone Sensoren, deren Daten gemeinsam zu verarbeiten sind, spielen in komplexeren praktischen Anwendungen häufig eine Rolle. Im RoboCup liefern beispielsweise Kameras und Roboterodometrie voneinander unabhängige Datenfolgen. Ultraschallsensoren oder Laserrangefinder stellen zusätzliche Quellen mit nicht synchronen Daten dar. Darüber hinaus können

die von den verschiedenen Robotern bestimmten Objektdaten als Daten unabhängiger logischer Sensoren aufgefaßt werden.

Anders als bei Rückkopplungen, für die es ausreicht, sie zu benennen, um automatisch eine sinnvolle und konsistente Steuerung zu etablieren, bieten sich für die gemeinsame Verarbeitung von nicht synchronen Eingangsdaten in einem Funktor eine ganze Reihe alternativer Vorgehensweisen an. Für die Auswahl einer optimalen Kombination von Steuerungs- und Zugriffsrelationen sind zahlreiche Faktoren, wie die jeweiligen Datenraten, die Anwendbarkeit von Interpolationsverfahren, die Arbeitsweise und Charakteristik der konkreten Sensoren, vor allem aber die Semantik der einzelnen Daten zu beachten. Aus diesem Grund soll es in der Hand des Entwicklers bleiben, geeignete Zugriffs- und Steuerrelationen auszuwählen und beim Programmentwurf zu spezifizieren.

Ausgangspunkt für die folgenden Überlegungen ist der in Abb. 6.26 dargestellte Datenflußgraph, in dem ein Verarbeitungsmodul Daten zweier unabhängiger Sensoren ($\mathbf{S_A}$ und $\mathbf{S_B}$) verarbeitet. Das Verarbeitungsmodul soll intern datengetrieben gesteuert werden. Gegenstand der Untersuchungen ist nun, welche Kombinationen der zuvor besprochenen Steuer- und Zugriffsrelationen für die Datenbeziehungen, die die Modulgrenzen überschreiten, möglich sind, und welche Konsequenzen sich daraus für die Modulsteuerung insgesamt ergeben.

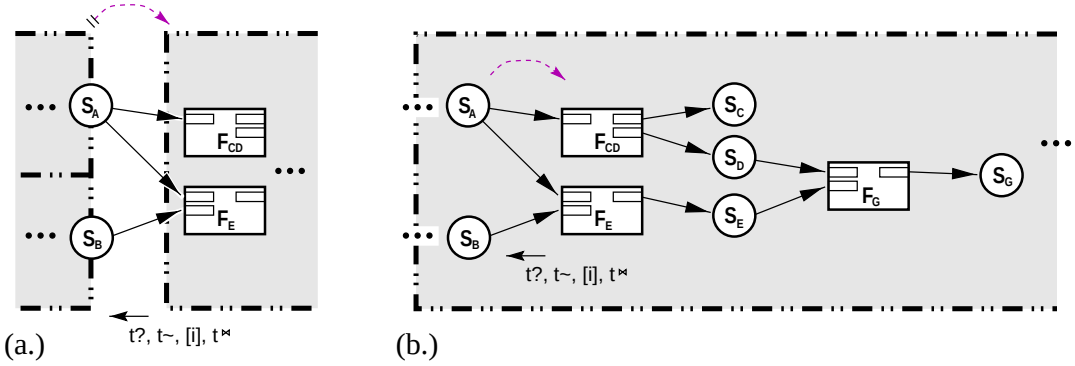


Abbildung 6.36: Datenfluß zwischen den nicht synchronen Eingangsdaten eines Moduls $\mathbf{S_A}$ und $\mathbf{S_B}$ und den sie verarbeitenden Funktoren. Die Datensequenzen können (a.) unabhängig vom betrachteten Modul existieren oder (b.) ein integraler Bestandteil des Moduls sein.

Die relevanten Objekte werden mit ihren Datenflußbeziehungen und den möglichen Relationen in Abb. 6.36 dargestellt. Dabei soll von der Steuerrelation (trig) ausgegangen werden. Diese kann für jede Datenflußbeziehung gesetzt oder nicht gesetzt sein. Es ergeben sich somit die folgenden Alternativen für die Steuerung:

1. Die (trig)-Relation ist für alle $\mathbf{S} \vec{\times} \mathbf{F}$-Datenflußbeziehungen gesetzt,
2. nur eine Eingangssequenz triggert einen oder auch mehrere Funktoren, oder
3. (trig) ist für keine der relevanten $\mathbf{S} \vec{\times} \mathbf{F}$-Beziehungen gesetzt, es wird also kein Funktor von außen getriggert.

Werden die Eingangssensoren unabhängig vom betrachteten Modul bereitgestellt, erfolgt der Funktoraufruf durch die Sensordaten in jedem Fall asynchron (Abb. 6.36 a). Alternativ dazu

können die Sensordaten auch innerhalb des Moduls bereitgestellt werden (vgl. Abb. 6.25). In diesem Fall können auch synchrone Steuerungsmechanismen gesetzt werden (Abb. 6.36 b).

In Abhängigkeit von den verschiedenen Möglichkeiten, die (trig)-Relationen zu setzen oder nicht, ergeben sich für die Modulsteuerung und die zu spezifizierenden Zugriffsrelationen die folgenden Konsequenzen. In welcher Form die in diesem Zusammenhang notwendigen zeittoleranten Datenzugriffe realisiert werden können und welche Eigenschaften diese mit sich bringen, wird in Abschnitt 6.4.4 diskutiert.

(trig) ist für alle Eingangsdaten gesetzt, d.h. alle Sensoren steuern gleichberechtigt das Modul:

- Jedes neue Eingangsdatum, egal von welchem Sensor es kommt, startet einen neuen Zyklus, d.h. es gibt Zyklen, die mit $\mathbf{S_A}$ synchron sind ($t_\mathsf{c} = t_\mathsf{g}(\mathbf{S_A})$) und andere, die $\mathbf{S_B}$ folgen.
- Teilnetze, die nicht von dem gerade aktiven Eingangsdatum abhängen, bleiben während des aktuellen Zyklus inaktiv: z.B. $\mathbf{F_{CD}}$ bei der Propagierung eines neuen Werts von $\mathbf{S_B}$.
- Damit die Zyklen erfolgreich zu Ende geführt werden können, müssen für beide Eingangsdaten zeittolerante Datenzugriffe möglich sein, und zwar an jeder Stelle im Netz, an der auf ein vom aktiven Sensor unabhängiges Teilnetz zugegriffen wird: Im Beispiel betrifft das die Zugriffe von $\mathbf{F_E}$ auf $\mathbf{S_A}$ und $\mathbf{S_B}$ sowie von $\mathbf{F_G}$ auf $\mathbf{S_D}$ (da $\mathbf{S_E}$ von beiden Eingangsdaten und damit auch immer vom aktiven Sensor abhängt, können alle Zugriffe mit der aktuellen Zykluszeit erfolgen).

(trig) ist nur für ein Eingangsdatum gesetzt, d.h. ein — dominanter oder *primärer* — Sensor steuert das Modul, die anderen — *sekundären* — Eingangsdaten werden nur gelesen:

- Ein neuer Zyklus startet nur, wenn die primäre Datensequenz aktualisiert wird.
- Enthält der Graph Funktoren, die *nicht* von der primären Datensequenz abhängen ($\mathbf{F_{CD}}$ bei primärer Sequenz $\mathbf{S_B}$; ist $\mathbf{S_A}$ dominant, gibt es kein solches Teilnetz), müssen diese durch einen anderen Mechanismus (Agent oder anfragegesteuert) aufgerufen werden.
- Zeittolerante Datenzugriffe sind nur für Zugriffe auf Daten, die ausschließlich von Sekundärsensoren abhängen, nötig. Für alle anderen Daten ist die aktuelle Zykluszeit gleich ihrer Datenmeßzeit t_g bzw. in ihr enthalten i_g (Intervall über die Datenmeßzeiten der von einem Funktor gemeinsam verarbeiteten Daten).

(trig) ist für kein Eingangsdatum gesetzt, d.h. das Modul wird nicht direkt durch die Eingangsdaten gesteuert:

- In diesem Fall muß das Modul anderweitig, z.B. durch einen Agenten oder über Datenanfragen angesteuert werden.

6.4.8 Anfragegetriebene Modulsteuerung

In diesem Abschnitt soll nun ein grundsätzlich anderer Mechanismus als der in Abschnitt 6.4.7 für die Steuerung der Sensordatenmodule vorgestellt und näher untersucht werden. Während im vorigen Abschnitt die Bereitstellung neuer *Eingangs*daten, die *vorwärts* durch den Datenflußgraphen propagiert werden, für die Triggerung der Modulzyklen verantwortlich war, werden

hier nun die Zyklen durch Datenzugriffe auf die *Ausgangs*daten angestoßen (vgl. Abb. 6.15 b). Anhand von Anfrage- und Zykluszeiten kann eine Datensequenz, von der ein Wert angefragt wurde, lokal entscheiden, ob sie diesen Zugriff mit den vorhandenen Daten befriedigen kann, oder ob die Sequenz zuvor aktualisiert werden muß, wofür sie die Anfrage *rückwärts* durch den Graphen weiterreicht. Das führt dazu, daß ein neuer Zyklus durch die Anfrage eines Ausgangssequenzwertes des Moduls $\mathbf{S}_{\mathcal{M},\mathbf{Out}}$ für einen aktuellen Zeitpunkt t_c eingeleitet und diese Anfrage über den für die Aktualisierung dieser Sequenz zuständigen Funktor $\mathbf{F}_{\mathbf{upd}}$, dessen Eingangsdaten $\mathbf{S}_{\mathbf{F},\mathbf{In}}$ usw. bis an die Eingangsdaten des Moduls $\mathbf{S}_{\mathcal{M},\mathbf{In}}$ durchgereicht wird. Von dort aus werden — entsprechend der bestehenden Datenabhängigkeiten — die offenen Datenzugriffe nach und nach befriedigt, bis auch das angeforderte Ausgangsdatum bestimmt werden konnte.

Umgesetzt wird dieses Steuerungskonzept, wie schon der zuvor beschriebene datengetriebenen Mechanismus, durch das Setzen ausgewählter Steuerungs- und Zugriffsrelationen für die bestehenden Datenflußbeziehungen. Diese stellen zur Laufzeit die Grundlage für die Auswahl der aufzurufenden Methoden dar. Hier spielen nun vor allem die Relationen, die die Kontrolle rückwärts, d.h. entgegen der Datenflußrichtung weitergeben, eine entscheidende Rolle. Das Spektrum der zu verwendenden Relationen wird dadurch wie folgt eingeschränkt. Erlaubt sind die Relationen:

Insistente Datenzugriffe — $\mathbf{F} \overset{\leftarrow}{\times} \mathbf{S} : (\mathsf{get}_t^{\triangleleft})$**:** Dieser Zugriff besteht auf einen aktuellen Wert, d.h. er geht mit der Forderung an die Datensequenz einher, den Wert durch ihren Aktualisierungsfunktor bestimmen zu lassen, falls er sich noch nicht in der Sequenz befindet. Dafür wird die Kontrolle an die Sequenz übertragen.

Einfordern der Datenaktualisierung — $\mathbf{S} \overset{\leftarrow}{\times} \mathbf{F} : (\mathsf{upd})$**:** Das Setzen dieser Relation ist notwendig, damit eine Sequenz ihre Aktualisierung aktiv vorantreiben und einen insistenten Zugriff auf ihre Daten beantworten kann. Dabei wird die Kontrolle an den Aktualisierungsfunktor $\mathbf{F}_{\mathbf{upd}}$ übertragen, der neben dem angeforderten Wert noch weitere von ihm bestimmte Ausgangsdaten setzen kann, bevor (bei synchronen Aufrufen) die Kontrolle an die aufrufende Sequenz zurückkehrt.

Neue Datenwerte setzen — $\mathbf{F} \overset{\rightarrow}{\times} \mathbf{S} : (\mathsf{set})$**:** Eintragen der durch einen Funktor berechneten Datenwerte in die entsprechenden Ausgangsdatensequenzen; diese Relation folgt immer zwingend aus dem Datenfluß. Da bei diesem Steuerungsmechanismus die Daten nicht mit der (trig)-Relation weiterpropagiert werden, ergeben sich durch das Setzen keine alternativen Programmzweige und die Kontrolle kehrt nach Anpassung der internen Werteliste sofort zum aufrufenden Funktor zurück.

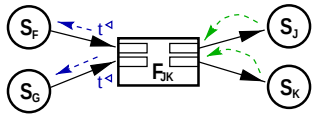
Passive Datenzugriffe — $\mathbf{F} \overset{\leftarrow}{\times} \mathbf{S} : (\mathsf{get}_{[\,]}), (\mathsf{get}_{[t]}), (\mathsf{get}_t^{?}), (\mathsf{get}_t^{\bowtie})$**:** Rein anfragende Datenzugriffe ohne Abgabe der Programmsteuerung an die Eingangsdatensequenz sind auch in diesem Steuerungsmechanismus erlaubt. Da sie allerdings nicht auf die Aktualisierung der Sequenz bestehen, ist ihre Anwendung auf die Eingangsdaten des Moduls sowie einige Sonderfälle beschränkt. Für die so zugegriffenen Daten wird damit vorausgesetzt, daß sie in einem unabhängigen Kontrollfluß bereitgestellt werden.

Interpolation/Prädiktion — $\mathbf{F} \overset{\leftarrow}{\times} \mathbf{S} : (\mathsf{get}_t^{\sim}), (\mathsf{get}_t^{\widetilde{\bowtie}})$**:** Im Falle der gemeinsamen Verarbeitung nicht synchron aufgenommener Eingangsdaten durch einen Funktor kann es wie bei der datengetriebenen Steuerung notwendig sein, die Datenwerte einer Sequenz durch

Interpolation zeitlich an die der anderen anzupassen. Die Kontrolle wird dabei nicht abgegeben, allerdings kann $(\mathsf{get}_t^{\widetilde{\bowtie}})$ gegebenfalls vor der Interpolation auf die Aktualisierung der Sequenz warten.

Die (trig)-Relation, die die Programmsteuerung im Datenflußdiagramm vorwärts, also in Datenflußrichtung weitergibt, wird hier nicht verwendet.

Aus diesen Einschränkungen folgt wiederum, daß für alle Datenflußbeziehungen Standardrelationen festgelegt werden können, die in Abb. 6.37 für einen einzelnen Funktor angegeben werden. Die Gestaltungsmöglichkeiten bei der Programmsteuerung ohne Wechsel des Steuerungskonzeptes beschränken sich auf die unterschiedlichen Zugriffsmethoden und auf den Grad der Parallelisierung, wobei bei den folgenden Untersuchungen wiederum von den synchronen Relationen $(\mathsf{get}_{to}^{\triangleleft}), \ldots$ ausgegangen werden soll.



$$\{\mathbf{S}_{\mathbf{J}}^{\rightarrow}, \mathbf{S}_{\mathbf{K}}^{\rightarrow}\} = \mathbf{F}_{\mathbf{JK}}(\, \mathbf{S}_{\mathbf{F}}^{\triangleleft}(t), \mathbf{S}_{\mathbf{G}}^{\triangleleft}(t)\,);$$

$$\begin{array}{ll} & \mathbf{F_{JK}}\ (\mathsf{get}_t^{\triangleleft})\ \mathbf{S_F} \\ & \mathbf{F_{JK}}\ (\mathsf{get}_t^{\triangleleft})\ \mathbf{S_G} \\ \mathbf{S_J}\ (\mathsf{upd})\ \mathbf{F_{JK}} & \mathbf{F_{JK}}\ (\mathsf{set})\ \mathbf{S_J} \\ \mathbf{S_K}\ (\mathsf{upd})\ \mathbf{F_{JK}} & \mathbf{F_{JK}}\ (\mathsf{set})\ \mathbf{S_K} \end{array}$$

Abbildung 6.37: Standardrelationen für ein rein vorwärts gesteuertes Datennetz.

Für die beteiligten Sequenz- und Funktorobjekte können entsprechend der Abschnitte 4.2.2 und 4.3.5 die folgenden Methoden aufgerufen werden:

Sequenzen:

- Setzen eines neuen Wertes mit $\mathsf{set}()$. Da die Liste der zu triggernden Funktoren $\{\mathbf{F_{dep}}\}^n$ in diesem Steuerungsmodell per Definition leer ist, beschränkt sich der Aufruf darauf, die internen Sequenzdaten anzupassen: $\mathsf{add}()$. Eine Propagation der neuen Datenwerte findet nicht statt, d.h. der $\mathsf{prop}()$-Aufruf wird sofort wieder beendet.
- Zugriff auf den aktuellen Sequenzwert: $\mathsf{get}_t^{\triangleleft}()$. Diese Methode testet zuerst, ob die Sequenz den geforderten Wert enthält. Ist dies der Fall, wird der Wert zurückgeliefert und der Aufruf beendet. Anderenfalls wird die interne $\mathsf{update}()$-Methode aufgerufen, die mit Hilfe des entsprechenden Funktors $\mathsf{call}(\mathbf{F_{upd}})$ die Sequenz aktualisiert. Im Anschluß daran kann der geforderte Wert zurückgegeben werden.

  Nach erfolgter Aktualisierung können für den eigentlichen Datenzugriff drei Fälle unterschieden werden: In einem rein synchron angesteuertem Graphen genügt es, in $\mathsf{get}_t^{\triangleleft}()$ auf das Ende der $\mathsf{update}()$-Methode zu warten. Die Sequenz ist dann auf jeden Fall aktuell, da $\mathsf{update}()$ erst nach Beendigung des Funktoraufrufs die Kontrolle zurückbekommt. Der Wert kann nun mit einem einfachen internen $\mathsf{get}()$-Zugriff ausgelesen und zurückgegeben werden.

  Wird $\mathsf{get}_t^{\triangleleft}()$ asynchron gestartet, bekommt der Funktor, der den Wert anfragt, unmittelbar nach dem Start von $\mathsf{get}_t^{\triangleleft}()$ die Kontrolle zurück, so daß er weitere Datenwerte anfordern kann. Parallel dazu wird in $\mathsf{get}_t^{\triangleleft}()$ die Aktualität der Sequenz überprüft und ggf. die $\mathsf{update}()$-Methode gestartet. Vor dem Start seiner $\mathsf{do}()$-Methode muß der Funktor auf

die Bereitstellung der angeforderten Daten warten und sie schließlich entgegennehmen, wofür er z.B. die $\mathsf{get}_t^{\bowtie}()$-Methode verwenden kann.

Schließlich ist auch ein gemischter Betrieb möglich, bei dem zwei synchron angesteuerte Teilnetze nebenläufig arbeiten und asynchron auf eine Sequenz oder einen Funktor zugreifen. In diesem Fall muß die synchron aufgerufene $\mathsf{get}_t^{\triangleleft}()$-Methode sicherstellen, daß die Kontrolle an den aufrufenden Funktor in jedem Fall erst nach der Aktualisierung der Sequenz zurückgegeben wird. Dies kann dadurch geschehen, daß der einfache $\mathsf{get}()$-Zugriff auf die interne Datenliste durch die wartende Version $\mathsf{get}_t^{\bowtie}()$ ersetzt wird. Dies stellt sicher, daß durch $\mathsf{get}_t^{\triangleleft}()$ kein alter Wert zurückgeliefert wird, selbst wenn der Funktoraufruf in $\mathsf{update}()$ wegen eines bereits laufenden vorzeitig beendet werden sollte.

Da die $\mathsf{get}_t^{\bowtie}()$-Methode sich bei einer bereits aktualisierten Sequenz wie ein einfacher $\mathsf{get}()$-Zugriff verhält, kann sie diesen ersetzen, ohne daß das auf den zuerst betrachteten, synchronen Fall Auswirkungen hätte. In der asynchronen $\mathsf{get}_t^{\triangleleft}()$-Variante kann dagegen auf den internen Zugriff auch ganz verzichtet werden.

- Neben dem insistenten Zugriff mit $\mathsf{get}_t^{\triangleleft}()$ sind auch rein lesende oder interpolierende Zugriffe, wie sie auch in der Vorwärtssteuerung Verwendung finden — mit oder ohne warten — erlaubt: $\mathsf{get}_t^{?}()$, $\mathsf{get}_{[]}()$, $\mathsf{get}_t^{\bowtie}()$, $\mathsf{get}_t^{\sim}()$, $\mathsf{get}_t^{\widetilde{\bowtie}}()$, wobei der wartende Zugriff $\mathsf{get}_t^{\bowtie}()$ bei asynchronem $\mathsf{get}_t^{\triangleleft}()$-Aufruf, wie zuvor gezeigt wurde, die Synchronisation zwischen der Datenbereitstellung und dem Funktoraufruf übernehmen kann.

Funktoren:

- Von außen kann ein Funktor mit Hilfe der $\mathsf{call}()$-Methode gestartet werden, wofür in diesem Modell die Ausgangsdatensequenzen verantwortlich sind. $\mathsf{call}()$ stützt sich dabei auf seine internen Methoden: $\mathsf{pre}()$ zum Anfordern und Sammeln der Eingangsdaten, $\mathsf{do}()$ für die Ausführung der eigentlichen Funktoroperatoren und $\mathsf{post}()$, um in den Ausgangsdatensequenzen die neu bestimmten Werte zu setzen.

Synchrone Ablaufsteuerung

In Abb. 6.38 wird der synchrone Kontrollfluß für einen Funktor und die mit ihm benachbarten Eingangs- und Ausgangsdatensequenzen (vgl. Abb. 6.37) als Sequenzdiagramm dargestellt. Alle Operationen werden sequentiell in einem *Thread* ausgeführt, wodurch die Aufrufreihenfolge nur von der Topologie des Graphen abhängt und deterministisch ist. Darüber hinaus ist im Moment eines Methodenaufrufs die Anzahl der möglichen Objektzustände, in dem sich die Sequenz oder der Funktor befinden kann, beschränkt. Bei der beschriebenen Arbeitsweise der Standardrelationen wird davon ausgegangen, daß alle durch sie verbundenen Objekte einen zyklenfreien Graphen bilden und daß die verwendeten Daten nicht durch andere aktive Komponenten kontinuierlich bereitgestellt werden. Rückkopplungen, die als Zugriff auf den vorhergehenden Zyklus spezifiziert werden, sowie Zugriffe auf unabhängig bereitgestellte Eingangsdaten sollen anschließend gesondert behandelt werden.

In diesem Sequenzdiagramm wird von dem im aktuellen Zyklus ersten insistenten Zugriff auf $\mathbf{S_J}$ (und $\mathbf{S_K}$) — durch einen Verbraucher, einen anderen Modulfunktor oder aber einen den Zyklus steuernden Agenten — ausgegangen: $\mathsf{get}_t^{\triangleleft}(\mathbf{S_J}, t_c)$. Da bisher weder auf $\mathbf{S_J}$ noch auf $\mathbf{S_K}$ zugegriffen wurde, ist auch der Funktor $\mathbf{F_{JK}}$ im laufenden Zyklus noch nicht aufgerufen

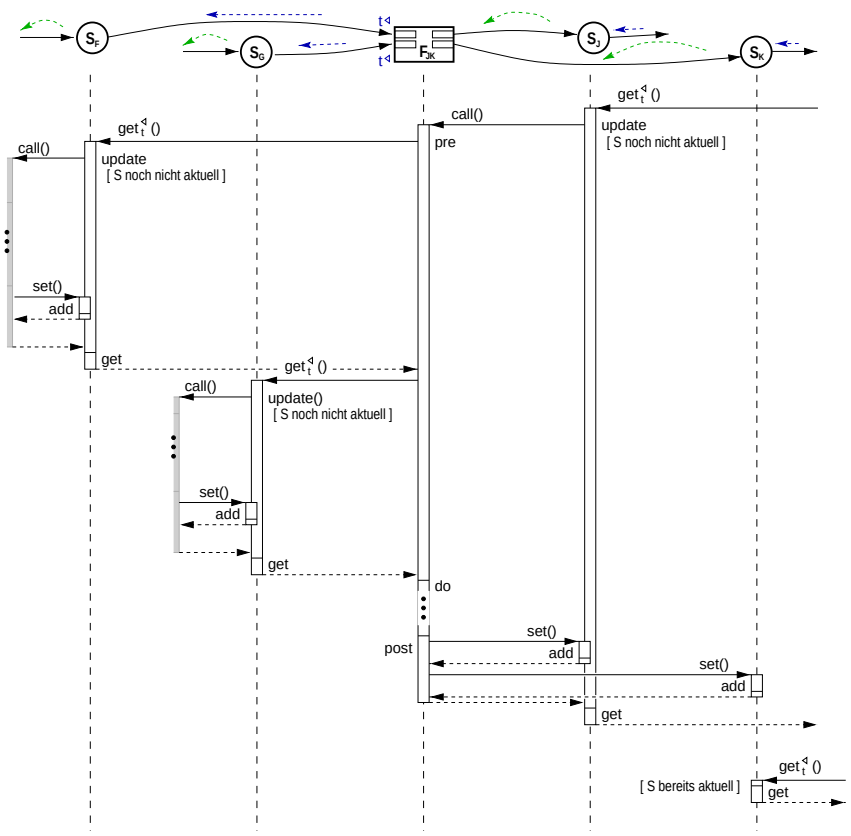


Abbildung 6.38: Sequenzdiagramm mit dem Kontrollfluß eines Funktors in einem synchronen, anfragegetriebenen Datenflußgraphen.

worden, d.h. $\mathbf{S_J}$ wurde noch nicht aktualisiert. Die Sequenz erkennt das anhand der Zykluszeit: $t_c(\mathbf{S_J}) = t_c(\mathbf{F_{JK}}) < t_c$. Sie startet ihre interne update()-Methode, die aufgrund der gesetzten $\mathbf{S_J}$ (upd) $\mathbf{F_{JK}}$-Relation den Funktor $\mathbf{F_{upd}} = \mathbf{F_{JK}}$ über dessen call()-Methode aufruft.

Damit hat nun der Funktor $\mathbf{F_{JK}}$ die Kontrolle. Als erstes fordert er in seiner pre()-Methode nacheinander die Eingangsdaten $\mathbf{S_F}$ und $\mathbf{S_G}$ an. Durch die synchronen Zugriffsrelationen geschieht dies streng sequentiell: erst wenn ein Datum (in diesem Fall S_F) bestimmt und der Wert zurückgeliefert wurde, wird das nächste Eingangsdatum (S_G) angefordert. Jede Anfrage wird rekursiv bis zu den Eingangsdaten, oder bis auf eine bereits aktualisierte Sequenz gestoßen wird, weiter gereicht, und in umgekehrter Reihenfolge, d.h. in Datenflußrichtung beantwortet. Dabei werden nur die Funktoren aufgerufen, von denen mindestens ein Ausgangsdatum für die Berechnung des angeforderten Datums notwendig ist und die im laufenden Zyklus noch nicht aktiviert waren. Nachdem $\mathbf{F_{JK}}$ den Sequenzwert S_F übernommen hat, wiederholt sich die gleiche Prozedur für S_G. Sollte die Sequenz $\mathbf{S_G}$ Ausgangsdatum eines Funktors sein, der für die Berechnung von $\mathbf{S_F}$ notwendig war, wäre $\mathbf{S_G}$ bereits während der $\mathsf{update}(\mathbf{S_F}, t_c)$-Methode gesetzt worden, und der $\mathsf{get}_t^{\triangleleft}(\mathbf{S_G}, t_c)$-Zugriff würde sofort den geforderten Wert liefern können. Im Beispielszenario ist dies nicht der Fall, weswegen nun $\mathbf{S_G}$ ihren Aktualisierungsfunktor aufruft. Sobald der neue Wert in $\mathbf{S_G}$ eingetragen und der Funktoraufruf beendet wurde, liefert $\mathsf{get}_t^{\triangleleft}(\mathbf{S_G}, t_c)$ den neuen Wert S_G an $\mathbf{F_{JK}}$. Dieser kann nun in $\mathsf{do}()$ seine eigentliche Aufgabe bearbeiten und die neuen Werte S_J und S_K bestimmen. Seine $\mathsf{post}()$-Methode trägt diese anschließend in die entsprechenden Datensequenzen $\mathsf{set}(\mathbf{S_J}, t_c, S_J)$ und $\mathsf{set}(\mathbf{S_K}, t_c, S_K)$ ein.

Nach Beendigung des Funktoraufrufs hat sowohl $\mathbf{S_J}$ als auch $\mathbf{S_K}$ den aktuellen Wert. Die Kontrolle kehrt zur Sequenz $\mathbf{S_J}$ zurück, die nun die laufende Datenanfrage mit einem internen Zugriff auf ihre Werteliste befriedigen kann. Wird später im Zyklus auch S_K angefordert, kann der Wert aufgrund des bereits erfolgten Aufrufs von $\mathbf{F_{JK}}$ sofort zurückgegeben werden.

Durch ein Vertauschen einzelner Eingangs- oder Ausgangsdatensequenzen der Funktoren, kann sich zwar die Reihenfolge der Funktoraufrufe und Datenaktualisierungen verändern, es treten aber keine grundsätzlich anderen Konstellationen als die zuvor beschriebenen auf.

Es ist offensichtlich, daß Funktoren ohne Ausgangsdaten wie $\{\ \} = \mathbf{F_X}(\mathbf{S_F})$ mit diesem Aufrufmechanismus nicht direkt angesprochen werden können. Sie müssen entweder über ihre Eingangsdaten oder durch einen Agenten getriggert werden. Repräsentiert ein solcher Funktor einen unabhängigen Verbraucher, kann er auch selber aktiv sein.

Funktoren, die wie $\mathbf{F_H}$ ein aktuelles Datum zusammen mit älteren Sequenzwerten der gleichen Sequenz verarbeiten: $\mathbf{S_H} = \mathbf{F_H}(\mathbf{S_F^{(0,-1)}})$, fügen sich dagegen ohne weiteres in den Mechanismus ein. Wird $\mathbf{F_H}$ durch seine Ausgangssequenz $\mathbf{S_H}$ aufgerufen, fordert $\mathbf{F_H}$ von der Eingangssequenz $\mathbf{S_F}$ einen neuen Wert an. $\mathbf{S_F}$ wird daraufhin aktualisiert und $\mathbf{F_H}$ kann sowohl auf den aktuellen als auch auf den vorherigen Wert zugreifen.

Asynchrone Ablaufsteuerung

In der anfragegetriebenen Modulsteuerung lassen sich die folgenden Relationen asynchron spezifizieren: $(\mathsf{get}^{\triangleleft}_{\mathsf{t}||})$, $(\mathsf{set}_{||})$ und $(\mathsf{upd}_{||})$. Dabei ist für die Parallelisierung der verschiedenen Teilaufgaben des Moduls vor allem die $(\mathsf{get}^{\triangleleft}_{\mathsf{t}||})$-Relation von Bedeutung, da sie es ermöglicht, mehrere Eingangsdaten eines Funktors nebenläufig in eigenen *Threads* anzufordern und bereitzustellen. Bei der asynchronen $(\mathsf{set}_{||})$-Relation werden die von einem Funktor bestimmten Ausgangsdaten parallel und nicht nacheinander gesetzt. Da dies allerdings keine rechenintensive Aufgabe ist, soll an dieser Stelle auf die Parallelisierung verzichtet werden.

Damit ein Funktor seine Eingangsdaten parallel bestimmen lassen kann, ist eine Zweiteilung der Datenzugriffe notwendig. In einem ersten Schritt werden die Daten angefordert, wofür die $\mathsf{get}^{\triangleleft}_{\mathsf{t}}()$-Methode asynchron gestartet wird. Die Konrolle kehrt sofort zum Funktor zurück, so daß diese Aktion für alle Eingangsdaten wiederholt werden kann. In einem zweiten Schritt muß nun auf die Daten gewartet werden. Eine einfache Möglichkeit, dies mit den vorgegebenen Mitteln zu realisieren, ist, unmittelbar nach der Initialisierung aller Datenanfragen, die $\mathsf{get}^{\bowtie}_{\mathsf{t}}()$-Methode beginnend beim ersten Eingangsdatum aufzurufen. Da dabei die Synchronisation der Eingangsdaten über die $\mathsf{get}^{\bowtie}_{\mathsf{t}}()$-Methode in Verbindung mit dem Eintragen der Daten in die Sequenz, nicht jedoch über das Ende der Aktualisierungsanforderung $\mathsf{call}()$ erfolgt, bringt es keinen Vorteil, den Funktoraufruf in einem eigenen *Thread* zu starten, d.h. die (upd)-Relation asynchron zu spezifizieren. Abb. 6.39 zeigt einen möglichen Kontrollfluß, wobei hier nur die $\mathsf{get}^{\triangleleft}_{\mathsf{t}}()$-Methode asynchron aufgerufen wird.

Auch hier können sich aufgrund unterschiedlicher Operatorlaufzeiten die verschiedenen Methodenaufrufe gegeneinander verschieben oder Aktionen zusammenfallen, die bei einer synchronen Ansteuerung streng sequentiell abgearbeitet werden. Um die Konsistenz der Daten und der Programmsteuerung zu gewährleisten, stellt sich damit wiederum die Frage, nach dem Schutz einzelner Aktionen oder Datenbereiche gegen eine parallele Bearbeitung sowie der Garantie von Aufrufreihenfolgen, um bestehende Abhängigkeiten bei der Datenverarbeitung und der Steuerung zu berücksichtigen.

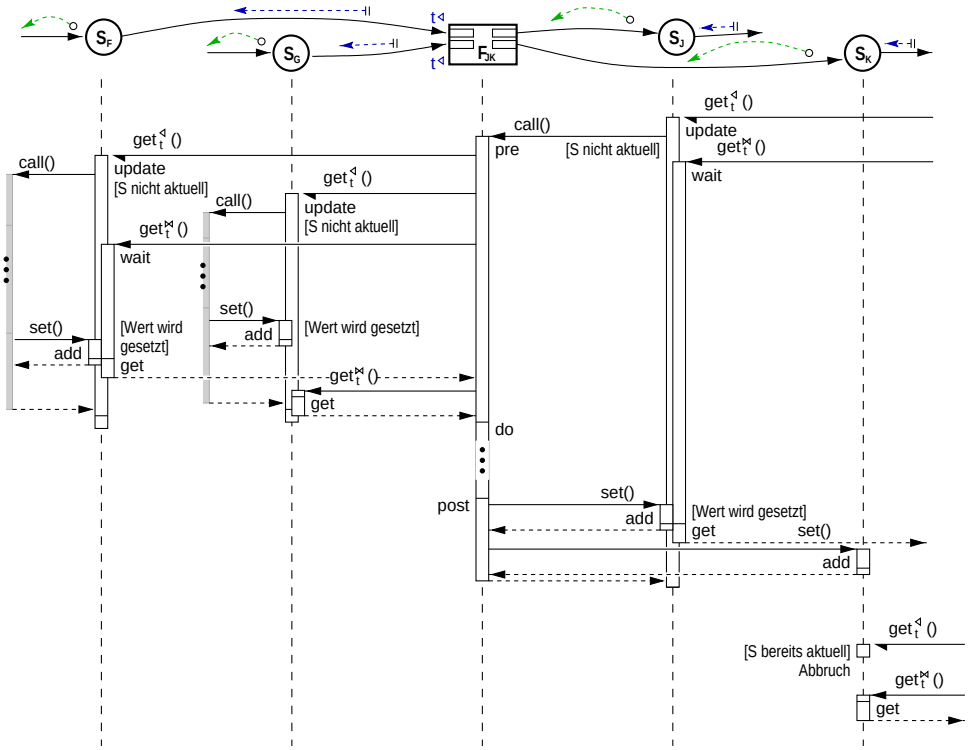


Abbildung 6.39: Sequenzdiagramm mit asynchronem Kontrollfluß eines anfragegetriebenen Funktors.

In einem anfragegetriebenen Netz können in einer Sequenz die folgenden Methoden, ggf. auch parallel, aufgerufen werden: $\mathsf{get}_{\mathsf{t}}^{\triangleleft}()$ — insistenter Datenzugriff mit den internen Methoden $\mathsf{update}()$ und $\mathsf{get}_{\mathsf{t}}^{\bowtie}()$ — letztere kann bei einem asynchronen Methodenstart entfallen; $\mathsf{get}_{\mathsf{t}}^{\bowtie}()$ und $\mathsf{get}_{\mathsf{t}}^{\widetilde{\bowtie}}()$ — lesende Zugriffe, die auf die Aktualisierung der Sequenz warten; $\mathsf{get}_{\mathsf{t}}^{?}()$, $\mathsf{get}_{[]}()$ und $\mathsf{get}_{\mathsf{t}}^{\sim}()$ — rein lesende Zugriffe ohne Warten auf bestimmte Aktionen (mit und ohne Test sowie interpolierend); und $\mathsf{set}()$ — Setzen eines neuen Sequenzwertes mit den internen Methoden $\mathsf{add}()$ und $\mathsf{prop}()$, wobei letztere Methode hier leer ist. Für das gleichzeitige Setzen und Abfragen von Werten mit $\mathsf{set}()$ und den verschiedenen, nicht wartenden oder insistenten $\mathsf{get}()$-Methoden, gilt im Wesentlichen das in Abschnitt 6.4.7 Gesagte. In Abb. 6.40 a und b werden die zu untersuchenden Parallelaufrufe dargestellt. Hier sind vor allem die zwei Fragen zu klären: „Wann kann der Funktoraufruf $\mathsf{call}(\mathbf{F}_{\mathbf{upd}})$ entfallen?“ und „Wie erfolgt die Synchronisation mit den noch zu setzenden Daten?“ Für Funktoren sind parallele Mehrfachaufrufe der $\mathsf{call}()$-Methode mit den internen Methoden $\mathsf{pre}()$, $\mathsf{do}()$ und $\mathsf{post}()$ zu untersuchen. Dies wird in Abb. 6.40 c dargestellt.

Ausgangspunkt der Untersuchungen ist wie in Abschnitt 6.4.7 eine für das Objekt $\mathbf{O}$: $\mathbf{O} \in \{\mathbf{S}\} \cup \{\mathbf{F}\}$ bereits laufende Methode $M()^{(i-1)} \in \{\mathsf{get}_{\mathsf{t}}^{\triangleleft}, \mathsf{set}, \ldots, \mathsf{call}\}$ (gegebenenfalls in einer internen Methode $M_X()^{(i-1)} \in \{\mathsf{add}, \ldots, \mathsf{post}\}$), zu der zusätzlich eine weitere Methode $M()^{(i)}$ aufgerufen wird. Dabei sollen hier nur die für die anfragegetriebene Steuerung typischen Fälle genauer untersucht werden.

- $M()^{(i-1)} \in \{\mathbf{get}_{\mathbf{t}}^{?}(), \mathbf{get}_{[]}(), \mathbf{get}_{\mathbf{t}}^{\sim}(), \mathbf{get}_{\mathbf{t}}^{\bowtie}(), \mathbf{get}_{\mathbf{t}}^{\widetilde{\bowtie}}()\}$, $\mathbf{O} = \mathbf{S}$: Parallele Zugriffe mit rein lesenden Methoden sind in jedem Fall unkritisch. Der eigentliche Zugriff auf die Sequenz-

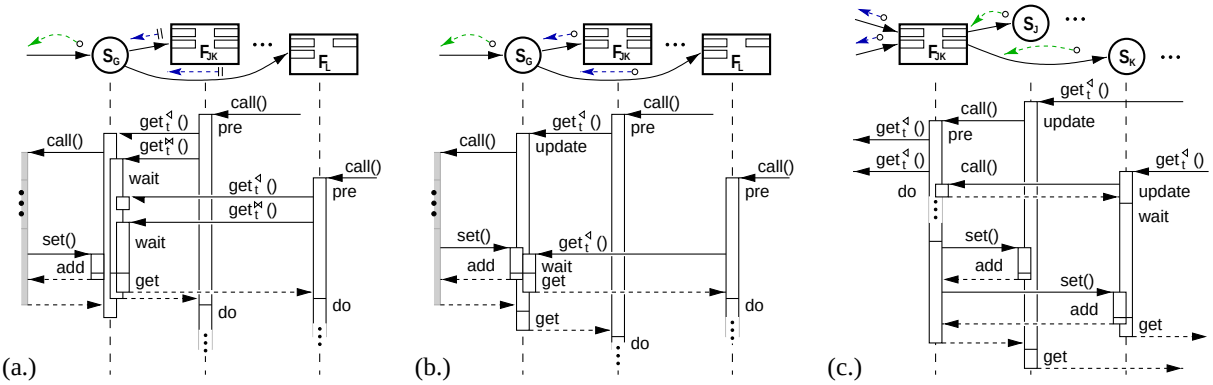


Abbildung 6.40: Parallele Methodenaufrufe durch asynchrone Programmsteuerung.

werte ist durch ein Semaphor $\zeta_{\mathbf{S}}^{\mathsf{values}}$ geschützt und erfolgt daher sequentiell. Auch mehrere parallel wartende Zugriffe ($\mathsf{get}_t^{\bowtie}()$ oder $\mathsf{get}_t^{\widetilde{\bowtie}}()$) stellen kein Problem dar, da diese vollkommen unabhängig voneinander sind.

Parallel zu wartenden Zugriffen muß natürlich eine Aktualisierung der Sequenz mit $\mathsf{set}()$ möglich sein, und, nachdem diese erfolgt ist, müssen die wartenden Zugriffe wieder aktiviert werden, so daß sie überprüfen können, ob der geforderte Wert jetzt verfügbar ist oder weiter gewartet werden muß. Insistente Datenzugriff mit $\mathsf{get}_t^{\triangleleft}()$ können ebenfalls parallel zu wartenden Zugriffen auftreten. Sie können unabhängig von anderen Zugriffsmethoden bearbeitet werden und die Aktualisierung der Sequenz einleiten.

- $M()^{(i-1)} = \mathbf{get}_{\mathbf{t}}^{\triangleleft}() \quad (M_x()^{(i-1)} \in \{\mathbf{update}(), \mathbf{get}_{\mathbf{t}}^{\bowtie}()\}),\ \mathbf{O} = \mathbf{S}$: Wird während der laufenden $\mathsf{get}_t^{\triangleleft}()^{(i-1)}$-Methode diese Aktion mit der gleichen Anfragezeit erneut gestartet, braucht der Funktor $\mathbf{F_{upd}}$ nicht angestoßen zu werden, d.h. die interne $\mathsf{update}()^{(i)}$-Methode kann entfallen, und $\mathsf{get}_t^{\triangleleft}()^{(i)}$ kann — als synchrone Methode gestartet (vgl. Abb. 6.40 b) — direkt zu dem internen $\mathsf{get}_t^{\bowtie}()$-Aufruf übergehen und auf das neue Datum warten. Im asynchronen Fall wird $\mathsf{get}_t^{\triangleleft}()^{(i)}$ sofort beendet, da mit dem Start des Funktors die Aufgabe bereits erledigt ist (vgl. Abb. 6.40 a). Wird $\mathsf{get}_t^{\triangleleft}()^{(i)}$ mit einer anderen Anfragezeit als der laufende Zugriff $\mathsf{get}_t^{\triangleleft}()^{(i-1)}$ aufgerufen, sind beide Aufrufe parallel zu bearbeiten und mit $\mathsf{update}()^{(i)}$ wird der Aktualisierungsfunktor erneut gestartet.

Parallel zur laufenden $\mathsf{get}_t^{\triangleleft}()^{(i-1)}$-Methode mit $\mathsf{set}()^{(i)}$ einen neuen Wertes zu setzen, ist selbstverständlich ohne Probleme möglich. Für die Sequenz, die den Aktualisierungsfunktor aufgerufen hatte, bedeutet dies das reguläre Verhalten.

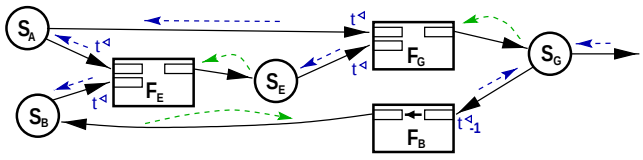
- $M()^{(i-1)} = \mathbf{set}() \quad (M_x()^{(i-1)} \in \{\mathbf{add}(), \mathbf{prop}()\}),\ \mathbf{O} = \mathbf{S}$: Bei einem Zugriff auf die Sequenz mit $\mathsf{get}_t^{\triangleleft}()^{(i)}$ parallel zu einer bereits laufenden Sequenzaktualisierung $\mathsf{set}()^{(i-1)}$ mit der gleichen Zykluszeit braucht die $\mathsf{update}()^{(i)}$-Methode nicht mehr gestartet zu werden, und $\mathsf{get}_t^{\triangleleft}()^{(i)}$ kann direkt auf den Wert warten (synchroner Aufruf) bzw. die Methode beenden (asynchroner Aufruf). Unterscheiden sich die Zykluszeiten und ist der geforderte Wert nicht verfügbar, wird regulär die $\mathsf{update}()^{(i)}$-Methode gestartet. Darüber hinaus gilt für das gleichzeitige Setzen und Lesen von Sequenzwerten das im Abschnitt 6.4.7 Gesagte.

- $M()^{(i-1)} = \mathbf{call}() \quad (M_X^{(i-1)} \in \{\mathbf{pre}(), \mathbf{do}(), \mathbf{post}()\}),\ \mathbf{O} = \mathbf{F}$: Ein Funktor $\mathbf{F}$ kann in einem Netz mit asynchronen Relationen von verschiedenen seiner Ausgangsdatensequenzen $\mathbf{S_{Out_i}}$ gleichzeitig aufgerufen werden. Da bei der Triggerung durch ein Ausgangsdatum $\mathbf{S}$

davon ausgegangen werden muß, daß der Funktor $\mathbf{F}$ selbst für die Bereitstellung der erforderlichen Eingangsdaten sorgt, kann der Funktoraufruf $\mathsf{call}(\mathbf{F}, t_c)^{(i)}$ sofort abgebrochen werden, wenn für den gleichen Zeitpunkt bereits ein Aufruf $\mathsf{call}(\mathbf{F}, t_c)^{(i-1)}$ läuft (Abb. 6.40 b). Die Synchronisation des Kontrollflusses mit den angeforderten Daten, für deren Bereitstellung der Funktor gestartet wurde, geschieht in der Sequenz $\mathbf{S}$ bzw. bei einem asynchronen Zugriff auf die Sequenz, in dem Funktor $\mathbf{F_X}$, der die Daten ursprünglich bei $\mathbf{S}$ angefordert hatte.

Rückkopplungszweige im Datenflußgraphen

Rückkopplungen sind auch in diesem Steuerungsmodell möglich. Abb. 6.41 zeigt ein Teilnetz, in dem der Datenflußpfad $\mathbf{S_G} \longrightarrow \mathbf{F_B} \longrightarrow \mathbf{S_B} \longrightarrow \mathbf{F_E}$ einen Rückkopplungszweig bildet. Dieser Pfad muß — entsprechend Abschnitt 6.4.2 — explizit einen Zugriff auf einen Wert des vorhergehenden Zyklus enthalten, in diesem Fall geschieht dies beim Zugriff von $\mathbf{F_B}$ auf $\mathbf{S_G}$. Eine Änderung der Standardzugriffsrelation $(\mathsf{get}_t^{\triangleleft})$ ist, wie die Abbildung zeigt, dabei nicht notwendig. Stattdessen wird der insistenten Zugriffsmethode $\mathsf{get}_t^{\triangleleft}()$ neben der Zykluszeit, ein negativer Relativindex übergeben, der den Zugriff auf einen alten Sequenzwert kennzeichnet: $\mathsf{get}_t^{\triangleleft}(\mathbf{S_G}, t_c, -1)$. Anhand der Zykluszeit kann die Sequenz $\mathbf{S_G}$ entscheiden, welcher Wert aus der internen Sequenzwertliste zurückgeliefert werden muß. Dies ist davon abhängig, ob die Sequenz im laufenden Zyklus bereits aktualisiert wurde oder noch nicht. Ersteres ist bei Zugriffen auf vergangene Werte ohne Rückkopplung (z.B. Funktor $\mathbf{F_H}(\mathbf{S_F^{(0,-1)}})$ in Abb. 6.4) möglich. Liegt ein solcher Fall vor, hat der neueste verfügbare Sequenzwert den Relativindex 0.



$\mathbf{F_G}\ (\mathsf{get}_t^{\triangleleft})\ \mathbf{S_A}$
$\mathbf{F_G}\ (\mathsf{get}_t^{\triangleleft})\ \mathbf{S_E}$
$\mathbf{F_E}\ (\mathsf{get}_t^{\triangleleft})\ \mathbf{S_A}$
$\mathbf{F_E}\ (\mathsf{get}_t^{\triangleleft})\ \mathbf{S_B}$
$\mathbf{F_B}\ (\mathsf{get}_t^{\triangleleft})\ \mathbf{S_G^{(-1)}}$

$\{\mathbf{S_G}\} = \mathbf{F_G}(\,\mathbf{S_A}^{\triangleleft}(t), \mathbf{S_E}^{\triangleleft}(t)\,);$
$\{\mathbf{S_E}\} = \mathbf{F_E}(\,\mathbf{S_A}^{\triangleleft}(t), \mathbf{S_B}^{\triangleleft}(t)\,);$
$\{\mathbf{S_B}\} = \mathbf{F_B}(\,\mathbf{S_G^{(-1)}}^{\triangleleft}(t)\,);$

$\mathbf{S_G}\ (\mathsf{upd})\ \mathbf{F_G}$
$\mathbf{S_E}\ (\mathsf{upd})\ \mathbf{F_E}$
$\mathbf{S_B}\ (\mathsf{upd})\ \mathbf{F_B}$

$\mathbf{F_G}\ (\mathsf{set})\ \mathbf{S_G}$
$\mathbf{F_E}\ (\mathsf{set})\ \mathbf{S_E}$
$\mathbf{F_B}\ (\mathsf{set})\ \mathbf{S_B}$

Abbildung 6.41: Relationen für die konsistente synchrone Rückwärtssteuerung eines rückgekoppelten Datengraphen.

Im zweiten, für Rückkopplungen typischen Fall zeigt aufgrund des neuen Zyklus und der noch nicht durchgeführten Aktualisierung der Relativindex -1 auf den letzten verfügbaren Wert der Liste. Ein Wert mit dem Index 0 ist zu diesem Zeitpunkt nicht verfügbar. Vorausgesetzt wird dabei, daß die Sequenz regelmäßig aktualisiert wird, was aufgrund der bestehenden Datenabhängigkeiten natürlich erst im Vorwärtszweig des Datenflußgraphen, d.h. *nach* dem zuvor beschriebenen Zugriff von $\mathbf{F_B}$ auf $\mathbf{S_G}$ erfolgen kann.

Als Nebenbedingung läßt sich daraus ableiten, daß die so zugegriffene Sequenz $\mathbf{S_G}$ anderweitig aktualisiert werden muß. Liegt diese (bzw. eine andere Ausgangssequenz des Funktors, der sie aktualisiert) in einem Vorwärtszweig des Graphen, geschieht die Aktualisierung automatisch bei jeder Datenanfrage, die über diesen Vorwärtszweig bearbeitet wird. Dargestellt

wird dies im Sequenzdiagramm in Abb. 6.42, das den Kontrollfluß für die Bearbeitung dieses Teilnetzes mit synchronen Relationen zeigt.

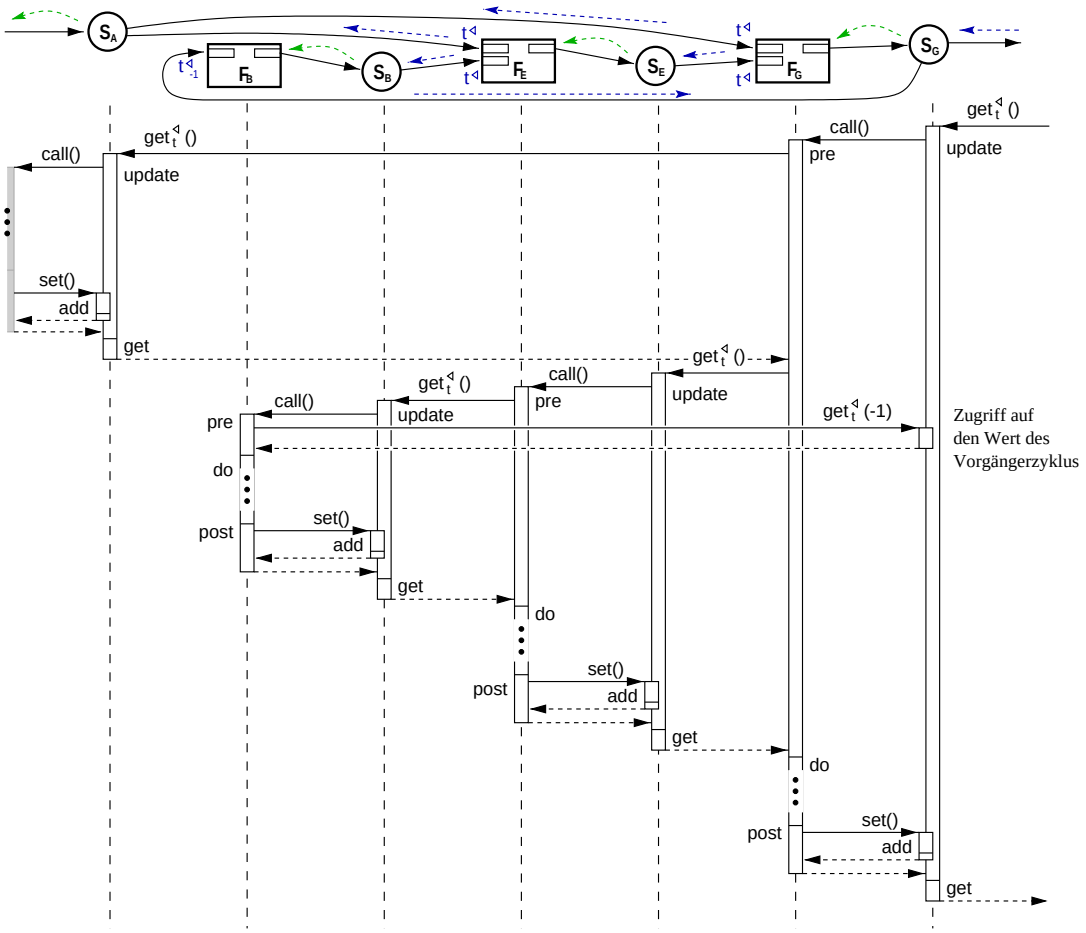


Abbildung 6.42: Kontrollfluß eines rückgekoppelten Datengraphen, mit synchronen, anfragegetriebenen Steuerungsmechanismen.

Liegen Funktor und Sequenz jedoch ausschließlich im Rückwärtszweig — wie das bei $\mathbf{F_B}$ und $\mathbf{S_B}$ der Fall ist, muß der Funktor entweder mit einem anderen Steuerungsmechanismus getriggert werden, oder der Rückgriff auf den alten Wert muß — insofern die Semantik der Daten dies erlaubt — im Rückkopplungszweig verschoben werden, wie dies in dem in Abb. 6.41 dargestellten Beispiel gegenüber der Darstellung in Abb. 6.33 geschehen ist: Anstelle des Zugriffs auf $\mathbf{S_B} : \mathbf{F_E}(\ldots, \mathbf{S_B^{(-1)}}) \Rightarrow \mathbf{F_E}(\ldots, \mathbf{S_B})$ wird nun von $\mathbf{S_G} : \mathbf{F_B}(\mathbf{S_G}) \Rightarrow \mathbf{F_B}(\mathbf{S_G^{(-1)}})$ der alte Wert gefordert.

Für asynchrone Zugriffsrelationen bietet sich noch eine weitere Möglichkeit an. Da sich hier der Kontrollfluß teilt und die Steuerung sofort zum Funktor zurückkehrt, kann $\mathtt{get}_t^{\triangleleft}(\mathbf{S_B}, t_c, -1)$ zum einen den alten Wert an den aufrufenden Funktor zurückgeben und parallel dazu mit `update()` die Bearbeitung des Rückkopplungszweiges einfordern. Dabei muß auf die Daten des Vorwärtszweiges gewartet werden, da dieser jedoch parallel weiterläuft, kann die offene Anforderung später im Zyklus beantwortet werden. In Abb. 6.43 wird der daraus resultierende

Kontrollfluß dargestellt, wobei hier wie in den vorangegangenen Darstellungen des Datenflußgraphen der Rückgriff auf den Vorgängerzyklus bei $\mathbf{S_B}$ erfolgt: $\mathbf{F_E}(\ldots, \mathbf{S_B^{(-1)}})$

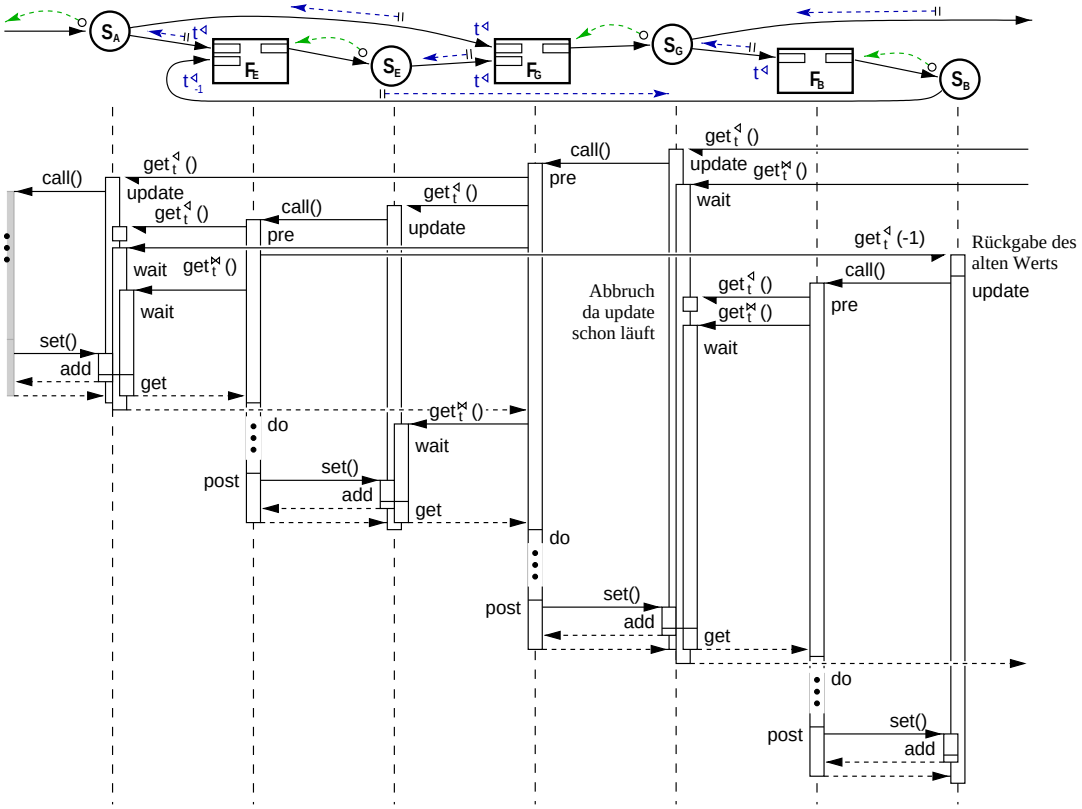


Abbildung 6.43: Kontrollfluß eines rückgekoppelten Datengraphen, mit asynchronen, anfragegetriebenen Steuerungsmechanismen. Im Gegensatz zu Abb. 6.42 erfolgt hier der Rückgriff auf den alten Wert von $\mathbf{S_B}$ und nicht von $\mathbf{S_G}$. Durch den asynchronen Zugriff auf $\mathbf{S_B}$ teilt sich der Kontrollfluß und bewirkt so die Aktualisierung dieser Sequenz *nach* dem Zugriff.

Neben dem direkten Zugriff auf einen alten Wert ist hier selbstverständlich auch eine Prädiktion für den aktuellen Wert aus den Sequenzdaten möglich. Wie bei einfachen Zugriffen auch muß dabei sichergestellt sein, daß die verwendete Sequenz regelmäßig aktualisiert wird. Im Gegensatz zur datengetriebenen Steuerung ist hierbei allerdings zu beachten, daß die Zykluszeit t_c, mit der auf die Daten zugegriffen wird, nicht unbedingt gleich der Datenmeßzeit t_g ist. Um den zu prädizierenden Wert zeitlich an den aktuellen Zyklus anzupassen, muß daher zuerst eine Zuordnung zwischen Datemeß- und Zykluszeit erfolgen. Eine solche Zuordnung kann modulweit erfolgen — dann muß zumindest ein Eingangsdatum des Moduls eingelesen worden sein — oder sie wird nur lokal im Funktor durchgeführt — dann muß bereits ein Eingangsdatum des Funktors bestimmt worden sein:

$$\{\mathbf{S_E}\} = \mathbf{F_E}(\mathbf{S_A^{\triangleleft}}(t = t_c) \circ \mathbf{S_B^{\sim}}(t = t_g(S_A^0)));$$

Relationen: $\mathbf{F_E}\ (\mathrm{get}^{\triangleleft}_{t\circ})\ \mathbf{S_A}$, $\mathbf{F_E}\ (\mathrm{get}^{\sim}_{t})\ \mathbf{S_B}$

Konflikte durch Rückkopplungszweige: Wie auch bei der Vorwärtssteuerung stellt es einen Designfehler dar, wenn ein vorhandener Rückkopplungszweig nicht durch einen Zu-

griff auf den Vorgängerzyklus angezeigt wird. Dieser sollte nach Möglichkeit erkannt und lokalisiert werden. Geschieht dies nicht, wird durch die wartenden Zugriffe eine Verklemmung provoziert: Trifft der Anfragestrom auf eine Sequenz, die bereits aktiv ist, wartet er auf das Ergebnis der dort laufenden Aktualisierung. Wird dafür jedoch die Antwort der laufenden Anfrage benötigt, kommt der Zyklus zu keinem Ende und der Prozeß hängt.

Hier ergeben sich nun entscheidende Unterschiede zwischen einem synchronen und einem asynchronen Kontrollfluß. Wird das Modul ausschließlich synchron angesteuert, d.h. alle Relationen sind synchron spezifiziert und der komplette Zyklus läuft in einem einzigen *Thread* ab, kann es bei regulärem Netzdesign nicht vorkommen, daß ein Funktor mehrfach mit $\mathsf{call}()$ aufgerufen, oder daß für eine Sequenz während einer bereits laufenden $\mathsf{get}_t^{\triangleleft}()$-Methode ein erneuter Zugriff mit $\mathsf{get}_t^{\triangleleft}()$ gestartet wird. Hat der Datenflußgraph jedoch einen Rückkopplungszweig ohne den obligatorischen Rückgriff auf den vorangegangenen Zyklus, tritt genau einer dieser beiden Fälle ein. Dies kann lokal leicht erkannt werden, und zu einer entsprechenden Ausnahmebehandlung führen.

Können im Modul jedoch auch asynchrone Programmzweige vorkommen, ist dieses Kriterium nicht geeignet, einen Konflikt zu erkennen, da auch im regulären Betrieb, wie Abb. 6.40 zeigt, parallele Mehrfachaufrufe dieser Methoden auftreten. Lokal ist dabei nicht zu unterscheiden, ob wartende Aufrufe durch einen noch aktiven *Thread* beantwortet werden können, oder ob durch das Anhalten eines Ausführungszweiges, um auf ein Datum zu warten, genau dies verhindert wird. Hier können nur weitergehende Kontrollmechanismen, wie sie in Abschnitt 6.4.5 beschrieben wurden, zum Erkennen von Fehlern im Netzdesign beitragen. Um im regulären Betrieb den Overhead aufwendiger Tests zu vermeiden, bietet es sich beispielsweise an, zu Beginn des Programms und nach jeder Änderung der Netztopologie einen Testzyklus — synchron oder durch einen Agenten überwacht — zu starten. Ein solcher Mechanismus kann leicht in die Sequenz- und Funktorobjekte integriert werden.

Zugriffe auf die Eingangsdaten des Moduls

Anders als bei der datengetriebenen Steuerung, wo die Zykluszeit direkt von den (dominanten) Eingangsdaten abhängt ($t_c = t_g(S_{\mathcal{M},In})$), wird in anfragegetriebenen Modulen die Zykluszeit i.d.R. durch die anfragende Instanz vorgegeben, ohne daß dabei die durch die Eingangssensoren vorgegebenen Datenmeßzeiten bekannt wären. Zeittolerante Datenzugriffe, wie sie in Abschnitt 6.4.4 diskutiert wurden, spielen daher für die Zugriffe auf die Eingangsdaten des Moduls in diesem Steuerungsmodell eine wichtige Rolle, selbst wenn keine unsynchronisierten Eingangsdaten zu verarbeiten sind.

Welches Zugriffsverfahren über die Relationen $(\mathsf{get}_{[]})$, $(\mathsf{get}_{[t]})$, $(\mathsf{get}_t^{\bowtie})$, $(\mathsf{get}_t^{\sim})$ und $(\mathsf{get}_t^{\widetilde{\bowtie}})$ im einzelnen auszuwählen ist, muß sich auch hier an der Netztopologie, den Eigenschaften der externen Sensoren sowie an Kriterien, wie Zugriffskonsistenz, Datenverzögerung, Wartezeit und Anwendbarkeit orientieren. Da diese Problematik bereits ausführlich in Abschnitt 6.4.4 diskutiert wurde, soll an dieser Stelle nur kurz auf die verschiedenen Zugriffsrelationen eingegangen werden.

$(\mathsf{get}_{[]})$ liefert den neuesten in der Sequenz verfügbaren Wert und stellt damit die einfachste Zugriffsmethode dar. Aufgrund des fehlenden Zeitbezugs ist diese Relation allerdings relativ ungeeignet für Datenflußgraphen, in denen mehrere Funktoren auf ein Eingangsdatum zugreifen. Wird zwischen zwei Zugriffen die Sequenz aktualisiert, rechnen die Funktoren mit in-

konsistenten Daten. Für solche Datenflußgraphen sind Mechanismen notwendig, die über die Zykluszeit die Datenzugriffe synchronisieren. Dies kann mit $(\mathsf{get}_{[t]})$ über den Zeitpunkt der Verfügbarkeit des Datums und mit $(\mathsf{get}_t^{\bowtie})$ anhand der Datenmeßzeit erfolgen.

Eine Anwendung von Interpolationsverfahren $(\mathsf{get}_t^{\sim})$ und $(\mathsf{get}_t^{\widetilde{\bowtie}})$ ist i.d.R. nur für sekundäre Daten sinnvoll, um diese zeitlich an die primären Daten anzupassen. Dies setzt voraus, daß ein primäres Datum bereits eingelesen wurde, und so eine eindeutige Zuordnung zwischen Datenmeßzeit und Zykluszeit möglich ist. Ist dies gegeben, gilt im wesentlichen das in Abschnitt 6.4.7 Gesagte.

Eine solche Zuordnung kann modulweit gültig sein und erfolgen, wenn in dem aktuellen Arbeitszyklus erstmals auf eine Eingangsdatensequenz $\mathbf{S}_{\mathcal{M},\mathbf{In}}$ zugegriffen wird, sie kann aber auch lokal im Funktor durchgeführt werden und dabei rekursiv durch den Graphen propagiert werden: nachdem ein Eingangsdatum $\mathbf{S}_{\mathbf{F},\mathbf{In}}$ gelesen wurde, steht die zeitliche Einordnung für die anderen Eingangsdaten, die Funktorbearbeitung und die von ihm ermittelten Ausgangsdaten fest. Die modulweite Zuordnung hat dabei den Vorteil, daß bereits nach dem ersten Datenzugriff die Zeit im ganzen Modul verfügbar ist, wodurch der parallele Zugriff auf die Eingangsdaten unterstützt wird. Der eingelesene Wert definiert dabei die Datenmeßzeit für den aktuellen Bearbeitungszyklus: $t_{\mathsf{c}} \hat{=} t_{\mathsf{g}}(S_{\mathcal{M},In})$. Zusammen mit der Zykluszeit charakterisiert die Meßzeit alle in einem Zyklus ermittelten Sequenzwerte. Wichtig ist dies beispielsweise, wenn zwischen mehreren dicht aufeinanderfolgenden Anfragezyklen die Eingangsdaten *nicht* aktualisiert wurden. Ohne Berücksichtigung der Datenmeßzeit würde dies zu einer wiederholten Bearbeitung der gleichen Eingangsdaten führen. Erkennt ein Funktor oder eine Sequenz jedoch, daß eine offene Anfrage auf die gleiche Datenmeßzeit wie eine bereits beantwortete abzubilden ist, kann die laufende Aktion abgebrochen werden, wobei für die erfolgreiche Fortführung wartender Aktionen, insbesondere von $\mathsf{get}_t^{\bowtie}()$, ein modifizierter $\mathsf{set}()$-Aufruf notwendig ist. Da die Sequenz für die Datenmeßzeit t_{g}, auf die die neue Zykluszeit t_{c} im Nachhinein abgebildet wurde, bereits einen Wert hat, stellt der modifizierte $\mathsf{set}()$-Aufruf für den Sequenzwert nur den neuen, erweiterten Zeitbezug her und weckt die eventuell wartenden Zugriffe $\mathsf{get}_t^{\bowtie}()$ auf. Diese können nun den der Anfragezeit zugeordneten Sequenzwert dem aufrufenden Funktor übergeben.

6.5 Anwendungsmuster für die Ansteuerung von Datenflußgraphen

Nachdem in den vorhergehenden Abschnitten grundsätzliche Konzepte für die Ablaufsteuerung innerhalb eines Modulzyklus vorgestellt wurden, soll an dieser Stelle nun untersucht werden, wie diese Mechanismen in die übergeordnete Gesamtsteuerung einer Applikation eingebettet werden können. Dafür ist insbesondere die Frage zu beantworten, wie ein solches Modul zur dynamischen Sensordatenverarbeitung mit externen Sensoren und Verbrauchern zusammenspielt, wie also die Kopplung an einen externen Prozeß erfolgen kann.

Für andere Programmkomponenten kann das gesamte Modul als ein logischer Sensor betrachtet werden, der kontinuierlich oder durch Datenanfragen getriggert eine Reihe von Sensordatenfolgen bereitstellt. Der interne Aufbau dieses Sensormoduls spielt aus Sicht der Konsumenten der Daten keine Rolle. Der Arbeitsmodus des Sensors läßt sich über die internen Steuerrelationen und ggf. einen zum Modul gehörigen Agenten einstellen.

Hervorzuheben ist hierbei, daß die eigentliche Realisierung des Sensormoduls, d.h. dessen interner Aufbau und Arbeitsmodus, erst in der Konfigurationsphase beim Start des Programms erfolgen muß und nicht schon während der Programmentwicklungsphase. Darüber hinaus kann diese Realisierung jederzeit im laufenden System verändert werden. Betroffen sind davon insbesondere die zu verknüpfenden Elemente des Graphen, die einzubindenden Verfahren sowie der Steuerungsmodus, d.h. die zu setzenden Steuerungsrelationen. Bei der Programmentwicklung sind die als Operatoren verfügbaren Datenverarbeitungsverfahren konkreten Funktorklassen zuzuordnen. Weiterhin sind die von anderen Komponenten nutzbaren Datensequenzen, d.h. die Ausgangsdaten des logischen Sensors zu *benennen*. Die Zuordnung zwischen diesen Namen und konkreten Sequenzobjekten erfolgt jedoch erst beim Programmstart. Dadurch läßt sich in Verbindung mit verschiedenen Konfigurationsmustern der Aufbau des Datenflußgraphen, die Auswahl konkreter Datenverarbeitungsverfahren und der Steuerungsmechanismus leicht ändern und an unterschiedliche Randbedingungen oder Anforderungen anpassen.

In Abhängigkeit von der zu bearbeitenden Aufgabe, der zur Verfügung stehenden Hardware, der Charakteristik der verwendeten Sensoren und aufgrund der unterschiedlichen Arbeitsweise und der Eigenschaften der beiden grundlegenden Steuerungsmechanismen ergeben sich typische Anwendungsmuster für die Konfiguration eines Datenflußgraphen. Die wichtigsten Designentscheidungen, die für die Modulsteuerung zu fällen sind, betreffen dabei

- das modulinterne Steuerungskonzept: daten- oder anfragegetriebene Steuerung (vgl. Abschnitt 6.4.7 bzw. 6.4.8);
- die Zugriffsverfahren auf externe Sensordaten: lesend, wartend, aktiv / insistent (vgl. Abschnitt 6.3.4 Seite 98 *ff* und Abschnitt 6.4.4;
- die Zykluskontrolle, d.h. das Bestimmen einer Instanz, die für das Starten der Zyklen verantwortlich ist: beliebiger externer Prozeß, Eingangssensoren, Verbraucher von Ausgangsdaten, modulinterne Agenten (vgl. Abschnitt 6.4.3);
- den Parallelisierungsgrad: vollständig synchron, partiell asynchron, vollständig asynchron (vgl. Abschnitt 6.4.6).

Jede der alternativen Vorgehensweisen bringt spezielle Vor- und Nachteile mit sich, was sich insbesondere in der mehr oder weniger guten Unterstützung der in Abschnitt 6.4.1 aufgestellten Bewertungskriterien und den jeweiligen Möglichkeiten zum Erkennen von Konflikten, in ihrer Eignung im Zusammenspiel mit bestimmten Sensoren sowie in ihrem Einfluß auf die Zeitcharakteristik (Verarbeitungsrate, Datenverzögerung und Wartezeit) des Datenverarbeitungsmoduls zeigt.

Die folgenden Randbedingungen haben maßgeblichen Einfluß auf die Auswahl konkreter Verfahren:

- Charakteristik der Sensoren: kontinuierliche oder getriggerte Arbeitsweise, Datenverzögerung, Wartezeit und Wiederholzeit, wobei letzteres in engem Zusammenhang mit der Zyklusrate des Moduls zu sehen ist (vgl. Abschnitt 5.3.2);
- Anbindung der Sensoren an das Modul: Anzahl der Sensoren, Anzahl der Zugriffe auf einen Sensor je Zyklus, Art der Einbindung (extern, intern oder gespiegelt, vgl. Abschnitt 6.4.3);

- verfügbare Hardware: Anzahl der Prozessoren und deren Auslastung, Verteilung der Datenverarbeitung in einem Rechnernetz;
- Umfang der in einem Zyklus zu ermittelnden Daten: sollen immer alle Daten bestimmt werden oder sind in Abhängigkeit von der aktuellen Aufgabe nur ausgewählte Ausgangsdaten notwendig.

Im folgenden soll untersucht werden, wie durch die zuvor genannten Steuerungsmechanismen auf die verschiedenen Randbedingungen mehr oder weniger gut reagiert werden kann. Es wird für die einzelnen Entscheidungsklassen gezeigt, welche Verfahren die bestehenden Anforderungen besser erfüllen können, und unter welchen Bedingungen einzelne Ansätze weniger geeignet sind. Daraus ergeben sich in Abhängigkeit von Sensoren, Datenverarbeitungsaufgabe und Hardware typische Anwendungsmuster, die die verschiedenen Mechanismen jeweils in einer geeigneten Weise kombinieren. Da die Mechanismen sich z.T. gegenseitig beeinflussen, wird an dieser Stelle besonders auf die Querbeziehungen zwischen den verschiedenen Entscheidungskategorien hingewiesen. Beispieldatenflußgraphen aus dem RoboCup-Szenario illustrieren die typischen Anwendungsstrukturen.

6.5.1 Parallelisierungsgrad

Der Grad der potentiellen Parallelisierung eines Sensordatenmoduls kann bei dessen Konfiguration durch die Auswahl synchroner oder asynchroner Relationen festgelegt werden. Werden nur synchrone Relationen verwendet, teilt sich innerhalb des Moduls der Kontrollfluß nicht und alle Aktionen werden streng sequentiell bearbeitet, asynchrone Relationen erlauben dagegen, daß der Kontrollfluß sich teilt und einzelne Aktionen parallel ausgeführt werden.

Stehen dem System mehrere Prozessoren zur Verfügung, können diese durch asynchron definierte Kontrollflüsse für die Sensordatenverarbeitung ausgenutzt werden. Teilaufgaben werden dann automatisch nebenläufig ausgeführt, wenn die Datenabhängigkeiten dies zulassen. Für die parallele Bearbeitung mehrerer Zyklen im Pipeliningbetrieb muß das Modul entsprechend oft angestoßen werden. Dies kann beispielsweise an die Bereitstellung der Eingangsdaten gekoppelt werden, dabei ist jedoch — z.B. durch einen Agenten — sicherzustellen, daß im Mittel nicht mehr Zyklen angestoßen werden, als tatsächlich bearbeitet werden können.

Sollte für die Sensordatenverarbeitung allerdings nur ein Prozessor zur Verfügung stehen, lohnt sich der Overhead, der durch das Anlegen und Bearbeiten nebenläufiger Programmzweige entsteht, nicht, und es können von vornherein synchrone Steuerungsrelationen definiert werden. Darüber hinaus hat eine streng sequentielle Bearbeitung den Vorteil, daß die Aufrufreihenfolge deterministisch ist und — bei einer anfragegetriebenen Steuerung — daß nicht deklarierte Rückkopplungszweige nur in diesem Modus erkannt werden können. Damit eignet sie sich besonders für Programmtests während der Entwurfsphase. Die Möglichkeit, bei einer veränderten Hardware, bzw. im realen Betrieb zu asynchronen Kontrollflüssen zu wechseln, bleibt davon unbenommen.

Asynchrone Kontrollflüsse aufgrund mehrerer externer Prozesse: Greifen unabhängig voneinander mehrere nebenläufig arbeitende Komponenten (Eingangssensordaten oder Verbraucher) aktiv auf das Datenverarbeitungsmodul zu, kann das in diesem zu mehreren parallelen Kontrollflüssen führen, selbst wenn alle Relationen synchron definiert sind. Abb. 6.44 demonstriert dies an zwei Beispielen, in Abb. 6.44 a sind die Triggerrelationen $\mathbf{S}_{\mathbf{Img}}\,(\mathsf{trig})\,\mathbf{F}_{\mathbf{Segm}}$

und $\mathbf{S_{USon}}\,(\mathsf{trig})\,\mathbf{F_{OccG}}$ und in Abb. 6.44 b die aktiven Zugriffsrelationen $\mathbf{F_{Plan}}\,(\mathsf{get}_t^{\triangleleft})\,\mathbf{S_{Goal}}$ und $\mathbf{F_{Robo}}\,(\mathsf{get}_t^{\triangleleft})\,\mathbf{S_{Ball}}$ für die parallelen Kontrollflüsse und damit für asynchrone Zugriffe auf Modulobjekte verantwortlich. Verhindern läßt sich dies nur durch einen Wechsel der Modulkontrolle, indem diese exklusiv einem einzigen Objekt übergeben wird. Nur dieses Objekt darf einen neuen Zyklus starten. Dabei kann es sich um *eines* der externen Objekte, aber auch um einen modulinternen Agenten handeln.

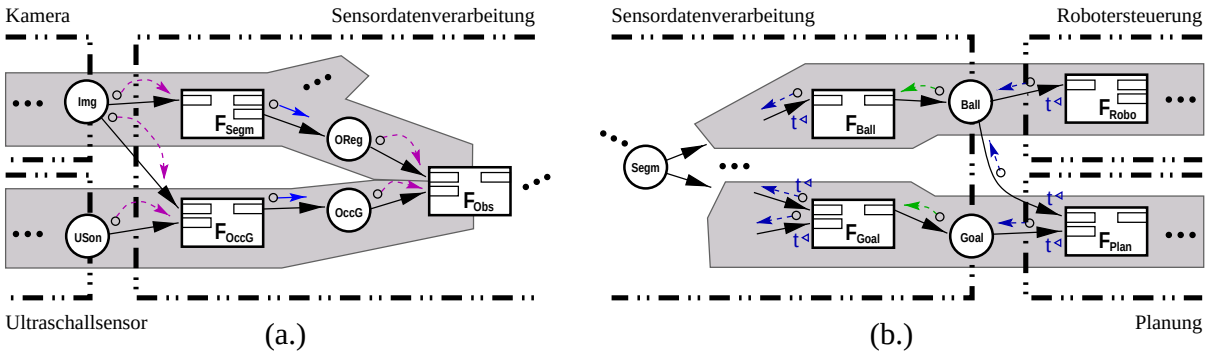


Abbildung 6.44: Asynchrone Objektzugriffe in einem Datenflußgraphen mit ausschließlich synchronen Relationen. Ursache für die parallelen Kontrollflüsse ist die aktive Modulansteuerung durch (a.) unabhängig voneinander aktive Eingangssensoren oder (b.) unabhängige Verbraucher.

6.5.2 Modulkontrolle und interne Steuerungsmechanismen

Damit ein Sensordatenmodul regelmäßig Daten bereitstellt, muß eine Instanz ausgewählte Modulobjekte zyklisch anstoßen und z.B. Funktoren mit $\mathsf{call}()$ aufrufen oder Sequenzwerte mit $\mathsf{get}_t^{\triangleleft}()$ abfragen. Dies muß in Einklang mit den Steuerungsrelationen geschehen, durch die der Datenflußgraph intern verschaltet ist. Ein solcher Anstoß bewirkt dann sukzessive die Aktualisierung aller relevanten Objekte des Datenflußgraphen. Die Kontrollinstanz, die für den Start der neuen Bearbeitungszyklen verantwortlich ist, kann zu einem externen Prozeß gehören, wobei es sich beispielsweise um die Eingangssensoren oder um Konsumenten von Ausgangsdaten handeln kann, oder das Modul enthält eine eigene, interne Kontrolle, z.B. in Form eines Agenten.

Kontrolle durch Eingangssensoren: Als erste Möglichkeit, ein Modul zu kontrollieren, soll hier die Triggerung der Eingangsfunktoren des Datenflußgraphen durch die Eingangssensordaten des Moduls mit $\mathbf{S_{Sensor}}\,(\mathsf{trig})\,\mathbf{F}_{\mathcal{M}}$ untersucht werden. Damit sich von dort die aktuellen Daten im Graphen ausbreiten können, wird bei dieser Form der Modulkontrolle intern eine datengetriebene Steuerung als Basismechanismus vorausgesetzt. Dieser Kontrollmechanismus setzt weiterhin voraus, daß die Eingangssensoren kontinuierlich in einem eigenen (externen) Prozeß arbeiten, und somit eine eigene Kontrollinstanz besitzen.

Ist die Dauer eines Bearbeitungszyklus im Modul $\tau_{\mathsf{c},\mathcal{M}} = f_{\mathsf{c},\mathcal{M}}^{-1}$ größer als die Datenrate der Eingangsdaten $\tau_{\mathsf{cg},\mathbf{S_{In}}} = f_{\mathsf{g},\mathbf{S_{In}}}^{-1}$, muß hier in Verbindung mit asynchronen Kontrollflüssen zusätzlich beachtet werden, daß das Modul nicht überlastet werden sollte: Durch den Start immer neuer Zyklen parallel zu den bereits laufenden, kann bei eingeschränkten Hardwareressourcen Rechenzeit von älteren, noch nicht beendeten Zyklen abgezogen werden, wodurch

deren Bearbeitungszeit weiter steigt. Bei einer zu großen Bearbeitungsdauer im Vergleich zur Eingangsdatenrate empfiehlt sich dieser Kontrollmechanismus daher nicht.

Darüber hinaus ist diese Vorgehensweise nicht sinnvoll, wenn die Ausgangsdaten des Moduls nur relativ selten weiter verarbeitet werden, gleichzeitig jedoch das Modul knappe Ressourcen bindet, die für andere Systemaufgaben fehlen. Besser ist es hier, den Start neuer Zyklen zusammen mit der Kontrolle über die gleichzeitig aktiven Zyklen und die Hardwareauslastung einem Agenten zu übertragen. Für den letzten Fall kommt u.U. auch die Ansteuerung des Moduls über seine Ausgangsdaten in Frage.

Kontrolle durch Zugriffe auf die Ausgangsdaten: Als zweite Möglichkeit können neue Bearbeitungszyklen durch insistente Zugriffe anderer Komponenten auf (prinzipiell beliebige) Moduldaten eingeleitet werden: $\mathbf{F}_{\mathbf{User}}\left(\mathrm{get}_{\mathrm{t}}^{\triangleleft}\right)\mathbf{S}_{\mathcal{M}}$. In Verbindung mit den Steuerungsrelationen für eine anfragegetriebene Modulsteuerung werden sukzessive alle für die Berechnung dieser Daten notwendigen Arbeitsschritte abgearbeitet. In diesem Fall müssen diese Verbraucher-Komponenten in einem externen, unabhängigen Prozeß arbeiten. Dabei kann es sich beispielsweise um ein Planungsmodul oder einen Prozeß für die Robotersteuerung handeln.

Der Vorteil dieser Zugriffsmethode liegt darin, daß hier nur die Daten bestimmt werden, die auch tatsächlich benötigt werden, d.h. ein Zyklus muß nicht den gesamten Graphen betreffen und das Modul muß nicht ununterbrochen arbeiten und so Rechenleistung in Anspruch nehmen. Bei relativ knappen Ressourcen kann das dazu beitragen, die Gesamtperformanz des Systems deutlich zu verbessern. Wird das Modul als logischer Sensor interpretiert, entspricht diese Arbeitsweise dem getriggerten Modus von Sensoren. Dies bedeutet, entsprechend Abschnitt 5.3.2, daß die Wartezeit auf die Daten $\tau_{\mathrm{w},\mathcal{M}}$ relativ groß ist, da mit der Datenverarbeitung erst *nach* dem Zugriff begonnen wird. Sind die Berechnungen im Modul sehr aufwendig, stellt dies u.U. im weiteren Datenverarbeitungsprozeß einen Nachteil dar. Allerdings darf die Wartezeit nicht mit dem Alter der Daten verwechselt werden. Dieses kann hier sogar kleiner sein als beim kontinuierlichen Modus, da die Daten unmittelbar nach ihrer Berechnung verwendet werden.

Enthält das Modul Rückkopplungszweige, also Zugriffe auf Daten des vorhergehenden Zyklus, ist als weiterer Punkt zu beachten, daß beim getriggerten Sensormodus zwischen zwei Datenzugriffen beliebig viel Zeit vergehen und daher das in der Rückkopplung verwendete Datum veraltet sein kann. Schließlich können hier, analog zur Modulkontrolle durch die Eingangsdaten, zahlreiche parallele Datenzugriffe zu einer Überlastung des Moduls führen. Aus diesen Gründen eignen sich diese Kontrollmechanismen nur für relativ einfache Szenarien mit eng umgrenztem Zeit- und Zugriffsverhalten.

Modulkontrolle mit Hilfe eines Agenten Da eine Zykluskontrolle durch externe Prozesse immer nur mit den genannten Einschränkungen möglich ist, soll hier als dritter Weg, der Einsatz von Agenten für diese Aufgabe vorgeschlagen werden. Die Kontrolle über die Datenverarbeitung liegt dabei nicht mehr direkt bei einem externen Prozeß, sondern bleibt innerhalb des Datenverarbeitungsmoduls. Dies erlaubt es, auch an den Modulgrenzen die Programmsteuerung vom Datenfluß zu entkoppeln, wodurch sie deutlich flexibler gestaltet werden kann, als dies bei den beiden vorherigen Verfahren der Fall ist. Insbesondere kann mit Hilfe von Agenten die Zyklusbearbeitung stärker an interne Kriterien, wie die durchschnittliche Modulrechenzeit, die Ressourcenauslastung oder eine eventuell geforderte Mindestdatenrate, gebunden werden.

Die Entkopplung der Programmsteuerung von einem externen Prozeß bedeutet dabei nicht, daß sich beide Prozesse nicht gegenseitig beeinflussen dürfen. Der Agent kann die Koordination mit dem anderen Prozeß übernehmen und die relevanten Ereignisse *kontrolliert* an das Modul weitergeben. So ist es eine Aufgabe des Agenten sicherzustellen, daß ein neuer Zyklus frühestens dann gestartet wird, wenn neue Daten anliegen. Treffen vor dem Ende des aktuellen Zyklus mehrere neue Eingangsdaten ein, kann der Agent entscheiden, ob bis auf die letzten alle anliegenden Daten verworfen werden, oder aber ob z.B. immer n Daten gleichzeitig im Pipeliningverfahren bearbeitet werden. Ein Agent ermöglicht darüber hinaus die Steuerung des Moduls, wenn kein externer Prozeß diese Aufgabe übernehmen könnte, weil z.B. die Eingangssensoren nicht im kontinuierlichen sondern im getriggerten Modus arbeiten.

Um einen Zyklus zu starten und abzuarbeiten, bieten sich die beiden bereits bekannten Basismechanismen an, d.h. entweder triggert der Agent die Eingangsfunktoren des Moduls: $\mathbf{A}\,(\mathsf{trig})\,\mathbf{F}_{\mathcal{M},\mathbf{In}}$ — in Verbindung mit dem datengetriebenen Steuerungskonzept: (trig), (set), oder er fragt alle zu berechnenden Ausgangsdaten ab: $\mathbf{A}\,(\mathsf{get}_{\mathsf{t}}^{\triangleleft})\,\mathbf{S}_{\mathcal{M},\mathbf{Out}}$ — bei anfragegetriebenem Steuerungskonzept: $(\mathsf{get}_{\mathsf{t}}^{\triangleleft})$, (upd). Letztere Methode erlaubt es zudem, sehr einfach den Umfang der Sensordatenverarbeitung frei zu skalieren, um z.B. verschiedene Sensordaten mit unterschiedlichen Datenraten bereitzustellen oder ganze Teilnetze zeitweise von der Bearbeitung auszunehmen. Funktoren, die keine Ausgangsdaten bereitstellen, können mit (trig) an ihre Eingangsdaten gekoppelt und als Seiteneffekt bei deren Aktualisierung bearbeitet werden, oder sie werden ebenfalls durch den Agenten aufgerufen.

An dieser Stelle soll der anfragegetriebenen Modulsteuerung der Vorzug gegeben werden, wofür insbesondere die folgenden Gründe sprechen:

- Gute Skalierbarkeit und Anpassungsmöglichkeiten an sich ändernde Randbedingungen und wechselnde Anforderungen, z.B. hinsichtlich des Datenumfangs — dabei genügt aufgrund der intern definierten Abhängigkeiten die Spezifizierung der Modulaufgabe, d.h. die Benennung der regelmäßig bereitzustellenden Daten, um die internen Operationen korrekt und vollständig durchführen zu können.
- Für den Agenten ist auch im Zusammenhang mit asynchronen Kontrollflüssen leicht erkennbar, wann ein Zyklus beendet wurde, da dies unmittelbar aus der Rückkehr der Datenanfragen folgt. Bei der Vorwärtssteuerung müßten dafür zusätzlich die Ausgangsdaten, etwa anhand der Datenmeßzeiten überwacht werden, was einen erhöhten Verwaltungsaufwand bedeutet.
- Wird das Modul synchron angesteuert, lassen sich potentielle Konflikte lokal und ohne zusätzlichen Aufwand erkennen, wofür der Agent beim Programmstart oder nach einer Rekonfiguration lediglich einen synchronen Testzyklus starten muß.

Realisiert wird dieser Mechanismus, indem der Agent entsprechend Abschnitt 4.4.1 jeweils eine Liste mit den zu aktualisierenden Datensequenzen $\{\mathbf{S}_{\mathbf{dep}}\}^n$ und den aufzurufenden Funktoren $\{\mathbf{F}_{\mathbf{dep}}\}^m$ enthält. Diese Listen können gegebenenfalls durch Anforderungsprofile ergänzt werden, in denen z.B. die minimalen und maximalen Datenraten $\tau_{\mathsf{cg},\mathbf{S_i}}^{\mathsf{min!}}$ und $\tau_{\mathsf{cg},\mathbf{S_i}}^{\mathsf{max!}}$ (bzw. $\tau_{\mathsf{c},\mathbf{F_i}}^{\mathsf{min!}}$, $\tau_{\mathsf{c},\mathbf{F_i}}^{\mathsf{max!}}$) spezifiziert werden. Darüber hinaus kann über ein externes (dominantes) Sensordatum, wie z.B. eine Kamerabildfolge, die Kopplung zu einem externen Prozeß hergestellt werden. Dadurch läßt sich einerseits sicherstellen, daß ein neuer Zyklus nur dann gestartet wird, wenn ein neues Eingangsdatum verfügbar ist. Andererseits ermöglicht dies, schon vor dem Zyklus-

start die aktuelle Datenmeßzeit zu ermitteln und diese für alle Datenzugriffe zu verwenden. Abb. 6.45 illustriert diesen Mechanismus anhand eines Beispielgraphen aus dem RoboCup.

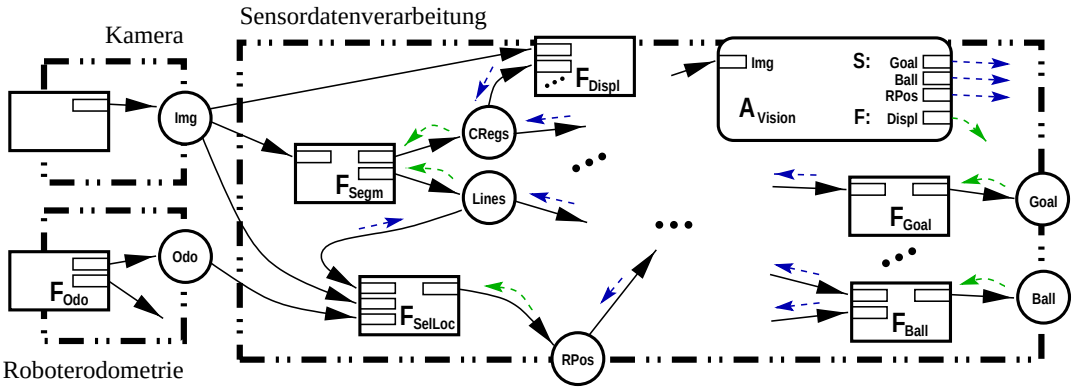


Abbildung 6.45: Modulkontrolle durch einen Agenten, der Listen mit den zu aktualisierenden Datensequenzen und Funktoren enthält. Durch Zugriffe auf die Kamerabildfolge wird sichergestellt, daß neue Zyklen sich an deren Datenmeßzeit orientieren.

6.5.3 Zugriffe auf die Eingangsdaten

An dieser Stelle soll der Zusammenhang zwischen dem Verfahren zur Modulkontrolle, der Sensorcharakteristik und den Zugriffen auf die Eingangsdaten hergestellt werden. Als Zugriffsmethoden kommen prinzipiell alle in Abschnitt 6.3.4 vorgestellten $(\mathtt{get})$-Relationen in Frage. Im einzelnen ergeben sich jedoch die folgenden Einschränkungen, wobei auf die zeitrelevante Bewertung der unterschiedlichen Zugriffsarten in Abschnitt 6.4.4 hingewiesen wird:

Insistente Zugriffe mit $(\mathtt{get}_t^{\triangleleft})$: Diese Zugriffsmethode kommt nur für Sensoren in Frage, die nicht kontinuierlich arbeiten, sondern neue Daten erst durch Zugriffe getriggert bereitstellen. Hat man bei einem Sensor die Wahl, ihn kontinuierlich oder getriggert zu betreiben, muß der Ressourenverbrauch, den der kontinuierliche Modus mit sich bringt, gegen die Wartezeit beim getriggerten Modus abgewogen werden.

Zeitgenaue Zugriffe mit $(\mathtt{get}_t^{?})$: Hierfür muß die zu erwartende Datenzeit bekannt sein, was bei einer Zykluskontrolle durch ein Eingangsdatum oder durch einen Agenten der Fall ist. Letzteres setzt allerdings voraus, daß der Agent die Zyklen mit einem lesenden Zugriff auf den aktuellen Wert dieses Eingangsdatums beginnt. Zeitgenaue Zugriffe können auf andere, sekundäre Eingangsdaten ausgedehnt werden, wenn für diese Daten eine Gültigkeitsdauer τ_v angegeben wird. Diese Vorgehensweise wie auch die folgenden zeittoleranten und wartenden Zugriffe werden ausführlich in Abschnitt 6.4.4 untersucht und einander gegenüber gestellt.

Zeittolerante Zugriffe: Diese Zugriffsmethoden kommen immer dann in Frage, wenn noch keine Datenmeßzeit für den laufenden Zyklus bekannt ist, oder aber wenn die Datenmeßzeit durch einen anderen als den zugegriffenen Sensor vorgegeben wird. Das ist vor allem bei der anfragegetriebenen Modulsteuerung und bei Zugriffen auf sekundäre Sensoren der Fall.

Wartende Zugriffe: Durch die Anwendung von wartenden Zugriffen auf die Eingangsdaten läßt sich aktiv das Zeitverhalten und die Datenrate des Moduls beeinflussen. Wartet das Modul bei jedem Zugriff auf seine Eingangsdaten auf einen neuen Wert, wird dadurch der Beginn eines neuen Zyklus an die Bereitstellung eines neuen Eingangsdatums gekoppelt, d.h. der interne Prozeß wird mit dem externen synchronisiert.

Zum Abschluß dieses Kapitels soll anhand eines Beispiels aus dem RoboCup das Zeitverhalten eines Sensordatenmoduls dargestellt werden, wobei insbesondere der Einfluß von wartenden im Vergleich zu nicht wartenden Zugriffsverfahren auf die Eingangsbilddaten untersucht wird. Interessieren sollen hier vor allem die Verarbeitungsrate des Moduls (bzw. die Ausgangsdatenrate) $f_{c,\mathsf{Vision}} = t^{-1}_{c,\mathsf{Vision}} = f_{g,\mathsf{Ball}} = \tau^{-1}_{cg,\mathsf{Ball}}$ und die Datenverzögerung $\tau_{s,\mathsf{Ball}}$, das ist die Zeitspanne zwischen der Messung des zugrundeliegenden Wertes $t_{g,\mathsf{Img}} = t_{g,\mathsf{Ball}}$ und der Verfügbarkeit des transformierten Wertes $t_{s,\mathsf{Ball}}$ (vgl. dazu die Abschnitte 5.1.1 und 5.3.2).

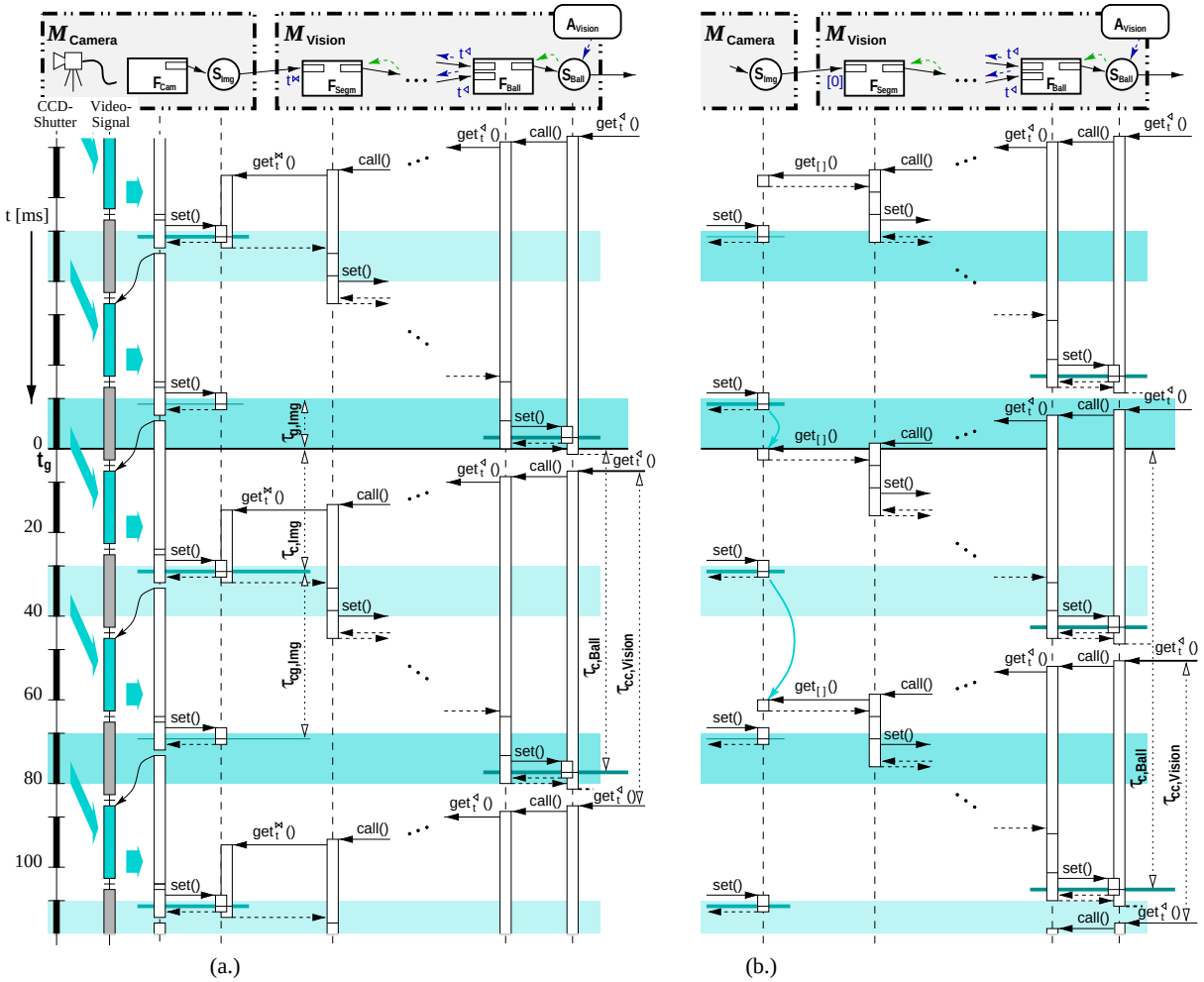


Abbildung 6.46: Vergleich des Zeitverhaltens eines Bildverarbeitungsmoduls, das (a.) wartend und (b.) nicht wartend auf eine Kamerabildfolge zugreift. Die dunklen Balken markieren den Aufnahmezeitraum der Bilder, die durch das Bildverarbeitungsmodul analysiert werden können, die hellen Balken Bilder, die aufgrund der Bearbeitungsdauer $t_{c,\mathsf{Vision}}$ nicht ausgewertet werden.

Abb. 6.46 zeigt das Zusammenspiel von Bildverarbeitungsmodul $\mathcal{M}_{\mathsf{Vision}}$ und Kameraeinbindung $\mathcal{M}_{\mathsf{Camera}}$. Die beiden Datenflußgraphen unterscheiden sich lediglich in der Art des Zugriffs auf die Bildfolge. Im ersten Fall wird beim Zugriff auf das jeweils nächste Datum gewartet: $(\mathsf{get}_{\mathsf{t}}^{\bowtie})$, im zweiten Fall wird sofort mit dem aktuell verfügbaren gerechnet: $(\mathsf{get}_{[]})$.

Die Steuerung der Kameraeinbindung ist direkt an das PAL-Videosignal gekoppelt, der logische Sensor $\mathbf{F}_{\mathbf{Cam}}$ arbeitet kontinuierlich mit einer Datenrate von $\tau_{\mathsf{cg,Img}} = 40\,\mathsf{ms} \mathrel{\hat{=}} f_{\mathsf{g,Img}} = 25\,\mathsf{s}^{-1}$. Dabei wird (wie im RoboCup) immer nur das erste Halbbild verwendet. Die reine Auswertung eines Bildes dauert im Beispiel in etwa $50 \ldots 60\,\mathsf{ms}$, das entspricht den realen Verhältnissen im RoboCup für die Selbstlokalisation sowie die Extraktion und Lokalisation von Ball, Spielfeld, Toren und anderen Robotern. Dort schwankt sie allerdings deutlich stärker — in Abhängigkeit vom aktuellen Bildausschnitt liegt sie in etwa zwischen 40 und $110\,\mathsf{ms}$. Dieser Wert ergibt sich aus der Summe der Rechenzeiten τ_{do} aller Funktoren des Moduls.

In Abhängigkeit von der Art des Zugriffs auf die Eingangsbildfolge ergeben sich unterschiedliche Bearbeitungsraten und Datenverzögerungen. Wie Abb. 6.46 zeigt, ist bei wartenden Zugriffen auf die Bildfolge (a.) die Ausgangsdatenrate $f_{\mathsf{g,Ball}} = \tau_{\mathsf{cg,Ball}}^{-1}$ geringer (bzw. ein Modulzyklus dauert länger) als bei nicht wartenden Zugriffen (b.). Sie beträgt aufgrund der Synchronisation mit den Bilddaten und der angenommenen Bearbeitungsdauer von $\tau_{\mathsf{do,Vision}} = 50\,\mathsf{ms} > 40\,\mathsf{ms}$ exakt $1/(80\,\mathsf{ms}) = 12,5\,\mathsf{s}^{-1}$, d.h. es wird genau jedes zweite Eingangsbild ausgewertet. Schwankungen der reinen Bearbeitungsdauer werden durch die Wartezeit immer auf das nächste ganzzahlige Vielfache der Zyklusdauer der Eingangsdaten aufgerundet.

Im zweiten Fall (b.) wird beim Zugriff jeweils das aktuell verfügbare Bild der Kamerabildfolge eingelesen und ausgewertet. Da die Wartezeit entfällt, wird die Zyklusdauer im wesentlichen durch die Rechenzeit der Bildauswertung bestimmt. In dem dargestellten Beispiel steigt dadurch die Bearbeitungs- bzw. Ausgangsdatenrate auf $f_{\mathsf{c,Vision}} = f_{\mathsf{g,Ball}} \approx 1/(50 \ldots 60\,\mathsf{ms}) = 16 \ldots 20\,\mathsf{s}^{-1}$. Es wird also nicht nur jedes zweite Bild ausgewertet, sondern im Mittel etwa 70% der Bilder.

Betrachtet man allerdings die Datenverzögerung $\tau_{\mathsf{s,Ball}}$, drehen sich die Verhältnisse um. Da bei nicht wartenden Zugriffen die Bereitstellung des im aktuellen Zyklus zu untersuchenden Bildes bis zu $40\,\mathsf{ms}$ zurückliegen kann, sind um genau diesen Betrag die extrahierten Daten im Moment ihrer Verfügbarkeit älter, als bei einem wartenden Zugriff. In beiden Zugriffsmethoden kommt bei der Bestimmung der gesamten Datenverzögerung $\tau_{\mathsf{s,Ball}}$ noch die Verzögerung, mit der das Bild bereitgestellt wurde ($t_{\mathsf{s,Img}} \approx 30 \ldots 40\,\mathsf{ms}$) und die für die Auswertung der Bilddaten nötige Zeit (ca. $50 \ldots 60\,\mathsf{ms}$) hinzu.

Welches Verhalten letztendlich gewünscht ist, muß der Entwickler entscheiden. Hier muß er abwägen, ob der höheren Verarbeitungsrate oder der Zeitnähe der Datenauswertung der Vorzug zu geben ist.

6.6 Zusammenfassung der Eigenschaften der dynamischen funktionalen Programmbeschreibung

Zum Abschluß dieser formalen Einführung in die Konzepte für eine dynamische funktionale Programmbeschreibung sollen im folgenden die wichtigsten Eigenschaften dieses Beschreibungskonzeptes zusammengefaßt werden.

- Grundlage des Konzeptes ist die Beschreibung der zwischen verschiedenen Daten und Funktionen bestehenden Zusammenhänge und Abhängigkeiten mit funktionalen Mitteln. Gleichungen und Datenflußgraphen stellen mögliche Ausdrucksformen der funktionalen Beschreibung dar.
- Datensequenzen und Funktoren, die die Basiselemente der funktionalen Programmbeschreibungen bilden, modellieren nicht nur den aktuellen Zustand eines bestimmten Datums oder einer Aktion, sondern auch dessen zeitlichen Verlauf, d.h. dessen Dynamik. Dabei wird zwischen dynamischen, langlebigen Objekten für die Repräsentation von Zustandsgrößen (S) und Datenverarbeitungsrelationen (F) mit ihren Beziehungen zueinander und der an einen bestimmten Zeitpunkt gebundenen, relativ kurzlebigen Instanziierung dieser Objekte — S für einen konkreten Datenwert und F für eine konkrete Aktion — unterschieden.
- Die funktionale Beschreibung wird um Ausdrucksmittel für eine Einflußnahme auf die Ablaufsteuerung erweitert. Mit Hilfe dieser Steuerungs- und Zugriffsrelationen läßt sich für die bestehenden Datenflußverbindungen spezifizieren, bei welchen Zugriffen ein Objekt die Steuerung abgibt und wo sich der Kontrollfluß aufteilen kann.
- Die dynamische funktionale Beschreibung legt für die Datenflußbeziehungen die Gleichzeitigkeit der miteinander verknüpften Daten zugrunde, geht hierfür also von einem synchronen Datenflußmodell aus. Sie beschreibt damit einen beliebigen synchronen Systemzustand. Dabei ist es nicht vorgesehen, mit speziellen Ausdrucksmitteln, wie Zeitgliedern o.ä., den Übergang zum nächsten synchronen Systemzustand in die Beschreibung zu integrieren, so daß der Graph sich durch den Datenfluß selbst zyklisch (re-) aktivieren könnte.
- Dem Datenverarbeitungsprozeß, der der Bereitstellung der Daten auf einem realen Rechner dient, und der durch die Steuer- und Zugriffsrelationen sowie die dynamischen Elemente repräsentiert wird, wird ein asynchrones Datenflußmodell zugrunde gelegt. Dabei wird das Erreichen eines konsistenten synchronen Systemzustandes von der zyklischen Wiederholung der Datenverarbeitung logisch getrennt.
- Datenwerte und Aktionen sind immer an bestimmte Zeitwerte gebunden. Dabei wird zwischen der Datenmeßzeit und der Bearbeitungszeit unterschieden. Erstere ist entsprechend des synchronen Datenflußmodells für alle gemeinsam verarbeiteten Daten gleich, letztere wird sich i.d.R. aufgrund des asynchronen Datenverarbeitungsmodells für verschiedene Daten und Aktionen unterschieden. Daten mit der gleichen Datenmeßzeit beschreiben damit gemeinsam den Systemzustand zu einem bestimmten Zeitpunkt. Die Zuordnung zwischen Zeitpunkt und Wert ist dabei für jedes Datum eindeutig.
- Auch die Datenzugriffe und Funktoraufrufe sind an bestimmte Zeitwerte gebunden. Diese Zeitwerte können als Zeitpunkte der Datenmessung oder als Zykluszeiten definiert werden. Für einen konkreten Aufrufzeitpunkt ergibt sich damit eine eindeutige Zuordnung von Datenwerten zu der auszuführenden Aktion. Erfolgt bei einer solchen Aktion ein Zugriff auf ein noch unbesetztes Datum, kann das ein hinreichendes Kriterium für dessen Aktualisierung sein.
- Einzelne Sensordaten können durch eine neue Messung unabhängig von den anderen Daten ihren Wert ändern und damit einen neuen Systemzustand (partiell) beschreiben. Dies muß nicht zwangsläufig die Anpassung aller anderen Daten nach sich ziehen. Daten,

für die für einen bestimmten Zeitpunkt (noch) kein neuer Wert bestimmt wurde, sind für diesen Zeitpunkt unbesetzt. Die Elemente des Datenflußgraphen können gleichzeitig alte und neue Systemzustände repräsentieren, wobei nicht jeder Systemzustand vollständig beschrieben sein muß.

- Ältere Daten, d.h. Daten die in vorhergegangen Zyklen bestimmt wurden, können in die Berechnungen ebenfalls mit einfließen. Dies muß beim Zugriff explizit spezifiziert werden.
- Der Datenflußgraph darf keine Scheifen enthalten, in denen ausschließlich Datenwerte eines Zeitpunkts enthalten sind, da dies zu nicht auflösbaren Datenabhängigkeiten führt. Schleifen müssen daher an mindestens einer Stelle aufgebrochen werden und einen Zugriff auf ein altes Datum enthalten.
- Ein Datenzugriff kann auch ohne zeitlichen Bezug erfolgen und vom aktuellen Zustand der Datensequenz ausgehen. Auch die so erhaltenen Daten sind mit Zeitinformationen ausgestattet, so daß deren zeitliche Einordnung im nachhinein möglich ist und für weitere Datenzugriffe verwendet werden kann.
- Datenwerte können über einen bestimmten Zeitraum nach ihrer Messung gültig sein, d.h. alle Datenzugriffe, deren Zugriffszeit innerhalb dieses Gültigkeitsintervalls liegt, werden auf die gleichen Datenwerte abgebildet (vgl. dazu Abschnitt 5.1.4). Dieser Zeitraum ist auch für alle im gleichen Arbeitszyklus abgeleiteten Daten und Funktionsaufrufe gültig.
- Daten können den Zustand „undefiniert“ einnehmen. Dies bedeutet, daß für einen bestimmten Zeitpunkt durch einen Funktoraufruf versucht wurde, einen Wert zu bestimmen, dieser Versuch jedoch fehlgeschlagen ist. Das kann z.B. geschehen, wenn die zu extrahierende Merkmale in einem Bild nicht vorkommen. Dieser Zustand ist nicht mit einem (noch) unbestimmten Wert zu verwechseln. Undefinierte Datenwerte können von nachfolgenden Funktoren wie reguläre Ergebnisse weiterverarbeit werden.

Mit Hilfe der in diesem Kapitel vorgestellten Konzepte lassen sich sowohl einfache Sensoreinbindungen darstellen als auch komplexe Multisensorsysteme mit verteilten und asynchron arbeitenden Sensoren und Rechnern. Dieser Ansatz erlaubt es, den internen Aufbau der Sensordatenmodule abstrakt zu beschreiben und mit allgemeinen Mechanismen zu konfigurieren. Dabei ist die konkrete Realisierung dieser Module und die Ablaufsteuerung in ihnen vollkommen unabhängig von den die Daten weiterverarbeitenden Komponenten. Aus Sicht dieser Komponenten können die kompletten Sensordatenmodule, wie auch einzelne Teile davon, als logische Sensoren betrachtet werden, die unabhängig vom Rest des Systems, auch zur Laufzeit, frei konfigurierbar sind.

Im folgenden Kapitel steht die objektorientierte Modellierung der hier und in den vorangegangenen Kapiteln vorgestellten Darstellungsmittel und Konzepte im Mittelpunkt.

Kapitel 7

Objektorientierte Modelle für die dynamische Sensordatenauswertung

7.1 Entwurfsgrundlagen

Während die vorhergehenden Kapitel die formale Definition einer um dynamische Ausdrucksmittel erweiterten funktionalen Beschreibungsmethode für dynamische Sensordatenprogramme zum Ziel hatten, steht nun die softwaretechnische Umsetzung des zuvor Beschriebenen mit objektorientierten Methoden im Mittelpunkt. Dafür werden in diesem Kapitel die Basissoftwarekomponenten, die den Kern der objektorientierten Bildfolgen- bzw. Sensordatenprogramme bilden, mit den wichtigsten Designentscheidungen im Kontext einer *dynamischen* und *zyklischen* Datenverarbeitung entworfen. Eine objektorientierte Modellierung grundlegender Basisdatentypen für die Bildverarbeitung, wie Bilder, Regionen, Linien usw. soll an dieser Stelle nicht erfolgen. Diese können aus bestehenden Bibliotheken, wie HALCON/C++ importiert werden.

Ausgangspunkt des objektorientierten Klassenentwurfs ist der Wunsch, mit Hilfe von Softwareobjekten eine möglichst direkte Umsetzung des funktionalen Systementwurfs in ein lauffähiges und effizientes Programm zu ermöglichen — einschließlich der vorgestellten Erweiterungen für die Programmsteuerung und der Flexibilität bei Anpassungen des Systems an veränderte Anforderungen und Randbedingungen. Somit bilden die Entwurfsgrundlagen der funktionalen Beschreibung auch die Basis des Softwareentwurfs.

Neben Klassen für die Repräsentation der verschiedenen absoluten und relativen Zeitgrößen sind hier vor allem die in Kapitel 4 vorgestellten Basiselemente der funktionalen Beschreibung zu modellieren. Datenfolgen ($\mathbf{S}$) werden durch eine *Sequenz*-Klasse (Sequence) repräsentiert, deren Instanzen Zeitreihen der einzelnen Sensordaten (S) beherbergen. Diese Elemente werden in Form von *Sequenzwerten* (SequenceValue) modelliert. Dabei handelt es sich um eine Containerklasse, die die eigentlichen Datenwerte (V) — Bilder, aber auch beliebige andere Datentypen ($\mathbf{T}$) — aufnimmt und mit bestimmten Status- und Zeitinformationen assoziiert.

Die Verarbeitung und Auswertung der Daten erfolgt in Funktoren $\mathbf{F}$. Instanzen der entsprechenden Klasse Functor repräsentieren neben der Funktionalität konkreter *Operatoren* auch die *Einbindung* der Verfahren in eine konkrete Ausführungsumgebung mit den Beziehungen zu anderen Objekten, insbesondere den Datensequenzen.

Agenten ($\mathbf{A}$) repräsentieren schließlich aktive Programmkomponenten, d.h. von ihnen können sowohl kontinuierliche als auch spontane Aktionen ausgehen. Darüber hinaus können sie

das System anhand vorgegebener Kriterien überwachen. Ihre Aufgabe ist es, die Arbeitszyklen eines aus Funktoren und Sequenzen bestehenden Moduls anzusteuern und zu kontrollieren, wofür sie regelmäßig neue Werte von den Ausgabesequenzen des Moduls anfordern sowie ausgewählte Funktoren aufrufen.

Wichtige Grundkonzepte der objektorientierten Programmierung wie Klassen, Instanzen, Vererbung, Polymorphie, Komposition, Aggregation und Assoziation werden hier als bekannt vorausgesetzt. Die graphische Darstellung von Objekt- und Klassenbeziehungen erfolgt mit den Ausdrucksmitteln der UML (*Unified Modelling Language*), da diese sich als ein De-facto-Standard etabliert hat. Die folgenden Klassenentwürfe orientieren sich in erster Linie an der konzeptionellen bzw. Spezifikationssicht auf die Klassenbeschreibungen und Funktionalitäten. Auf Implementierungsdetails, die sich z.B. aus den Besonderheiten der verwendeten Programmiersprache C++ ergeben, soll an dieser Stelle weitestgehend verzichtet werden.

Für Objekte, die in diesem Kapitel vorgestellt werden, gelten die folgenden Konventionen bei der Angabe von Attributen und Methoden.

- Auf Attribute und assoziierte Objekte der vorgestellten Klassen kann mit Hilfe von standardisierten Methoden zugegriffen werden, die nicht gesondert angegeben werden. Die Namen dieser Methoden werden nach einem regelmäßigen Schema gebildet, für eine Instanzvariable mit dem Namen und Typ varname: VarType bzw. für eine Liste varnameList: VarType[] ergeben sich die folgenden Methodennamen:

getVarname () : VarType	Auslesen eines Attributwertes bzw. einer Objektreferenz;
setVarname (VarType)	Setzen eines Attributwertes bzw. einer Objektreferenz;
isVarname () : bool	Abfrage des Wahrheitswertes von boolschen Attributen;
addVarname (VarType)	Wert oder Objektreferenz in eine Liste eintragen;
remVarname (VarType)	Wert oder Objektreferenz aus einer Liste entfernen.

- Konvertierungen von Objekten in einen anderen Typ ResultType erfolgen mit Hilfe von Methoden der Form asResultType() und sind i.d.R. explizit anzugeben.
- Basistypen wie string, integer, bool, ... werden durch die zugrundeliegende Programmierumgebung bereitgestellt.
- Für alle Objektklassen werden Konstruktor- und Destruktormethoden definiert.
- Alle hier entwickelten Objektklassen sind von einer gemeinsamen Basisklasse RootObject abgeleitet, die einige wenige grundlegende Methoden, insbesondere für die Objektidentifikation sowie für Kommunikation und externe Repräsentation, jedoch keine Attribute definiert. Eine Auswahl der wichtigsten Methoden wird in Tabelle 7.1 angegeben.
- Im Rahmen dieses Systems werden zahlreiche weitere Klassen und Datentypen verwendet, z.B. Klassen für den Datenaustausch über Netzverbindungen und mit dem Filesystem: Stream, und davon abgeleitet FileStream und NetStream. Das Einlesen von Videobildfolgen erfolgt über eine Framegrabber-Einbindung Framegrabber, Kameraparameter können mit Hilfe der Camera-Klasse repräsentiert werden.

Die vorgestellten Klassen wurden prototypisch in C++ implementiert und sind für die Sensordatenauswertung im Agilo-RoboCuppers-Team der TU München in einer Roboterfußball-Umgebung mit Erfolg eingesetzt worden.

RootObject	
asString () : string	Umwandlung der Objektdaten in einen lesbaren String
getClassName () : string	Ausgabe des Klassennamens.
getName () : string	Ausgabe des Objektnamens, vorausgesetzt die abgeleitete Klasse unterstützt dies und verwaltet den Namen.
getHostname () : string	Liefert den Rechnernamen, auf dem das Objekt beheimatet ist. Wird ein Objekt von einem anderen Rechner gespiegelt, ist dies der Name des Rechners, auf dem das Original liegt.
getIdentString () : string	Ausgabe einer Objektidentifikation; diese besteht aus Objekt-, Klassen- und Rechnernamen, ersteres nur, wenn die konkrete Klasse die Objektbenennung unterstützt.
write (Stream)	Ausgabe der Objektdaten in einen Datenstrom in maschinenlesbarer Form, so daß das Objekt vollständig rekonstruierbar ist.
read (Stream)	Lesen von Objektdaten aus einem Datenstrom, um ein Objekt aus seinen Rohdaten zu rekonstruieren.

Tabelle 7.1: Wichtige Methoden der abstrakten Basisklasse RootObject.

7.2 Modellierung der Zeitgrößen

Entsprechend Kapitel 3 wird bei der Modellierung von Zeitaspekten zwischen absoluten und relativen Zeiten sowie zwischen Zeitpunkten und Intervallen unterschieden. Dies spiegelt sich in der in Abb. 7.1 dargestellten Klassenhierarchie wider; Zeitwerte können durch Instanzen der Klassen TimePointAbs, TimePointRel, TimeIntervalAbs und TimeIntervalRel repräsentiert werden.

Diese Klassen stellen in erster Linie die in Abschnitt 3.3 definierten Vergleichs-, arithmetischen und Zugriffsoperationen sowie Konstruktoren aus anderen Zeitwerten zur Verfügung. Darüber hinaus kann mit Hilfe der Klasse TimePointAbs die aktuelle Uhrzeit bestimmt werden: TimePointAbs :: now(). Werden Absolutzeiten unterschiedlicher Systeme gemeinsam auf einem Rechner bearbeitet, läßt sich die Zeitabweichung zwischen den Systemen $\tau_{\mathrm{T}_{1,2}}$ durch die Instanzvariable offset darstellen.

Die Basisklassen TimeRel und TimeAbs ermöglichen es schließlich, für Attribute oder Methodenparameter (z.B. bei der Datenmeßzeit oder der Zykluszeit) sowohl Intervalle als auch Zeitpunkte zuzulassen. Eine Umwandlung in einen konkreten Typ ist dadurch erst dann notwendig, wenn eine Operation tatsächlich einen bestimmten Typ voraussetzt. Dafür stehen die Methoden asTimePointAbs(), asTimeIntervalAbs(), asTimePointRel() und asTimeIntervalRel() zur Verfügung.

7.3 Modellierung der Sequenzwerte

Für die Auswertung von Sensordaten in einer dynamischen Umgebung ist es notwendig, die einzelnen Datenwerte V mit verschiedenen Attributen, wie Zeitwerten oder anderen dynamischen Daten zu assoziieren. Darüber hinaus verlangt die dynamische Sensordatenverarbeitung nach einer Reihe von Aktionen, die unabhängig von einem konkreten Datentyp definiert werden

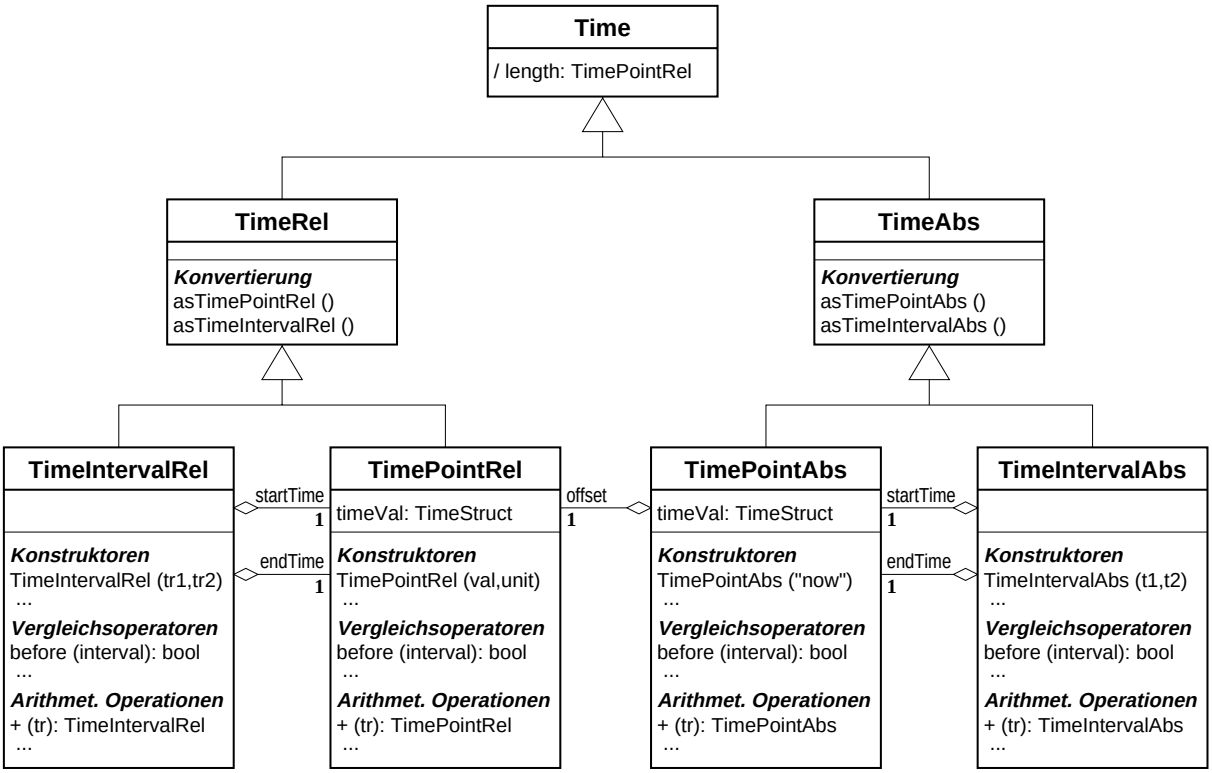


Abbildung 7.1: Klassendiagramm für die Modellierung von Zeitwerten.

können. Daraus läßt sich, entsprechend der in Abschnitt 4.2.1 vorgeschlagenen Modellierung von Sequenzen, ein spezieller Datentyp für die Modellierung dynamischer Datenwerte ableiten. Dieser wird als Sequenzwert $S = (V, \mathbf{T}, \ldots)$ bezeichnet und durch die Klasse SequenceValue repräsentiert. Deren Aufgabe ist es, beliebige Daten um dynamische Aspekte zu erweitern.

7.3.1 Unterscheidung zwischen Zeit und Wert

Die Verbindung von Datenwerten mit Attributen und Methoden kann mit verschiedenen Entwurfsstrategien umgesetzt werden. Ein Ansatz wäre z.B., die in Zeitfolgen zu verarbeitenden Datenklassen bei ihrem Entwurf gleich mit den zusätzlichen Attributen auszustatten. In einer gemeinsamen Basisklasse können alle zeitrelevanten Erweiterungen zusammengefaßt und so einfach an die zu verarbeitenden Daten vererbt werden. Diese Vorgehensweise ist jedoch nur für neu zu definierende Datentypen möglich, nicht aber für Basisdaten des verwendeten Systems oder aus anderen Bibliotheken übernommene Klassen, was die Anwendbarkeit deutlich einschränkt. Außerdem bedeutet dies einen erheblichen Eingriff in die Datenstrukturen, die auch für andere, statische Anwendungen von Bedeutung sind. Da alle Datenstrukturen bei diesem Ansatz die Zeitinformationen *immer* mit sich tragen würden, verschiebt sich deren Semantik und ihre Repräsentation wird künstlich aufgebläht.

Die hier relevanten Attribute stellen eine Art Metawissen über eine konkrete Ausprägung und den Kontext der Daten dar, sie sind jedoch kein fester Bestandteil von diesen und dienen nicht zu deren generellen Beschreibung. Daher spiegelt die Vererbungsrelation die Beziehung zwischen den Daten und deren Modellierung als Element einer Sequenz nicht korrekt wider.

Ein weiterer Modellierungsansatz für das Ausstatten von Daten mit zusätzlichen Attributen ist, die Zielklasse gleichermaßen aus den einfachen, statischen Daten *und* einer allgemeinen dynamischen Objektklasse abzuleiten. Dies setzt jedoch Mehrfachvererbung voraus, ein Konzept das nicht unumstritten ist und daher nicht in allen objektorientierten Programmiersprachen unterstützt wird.

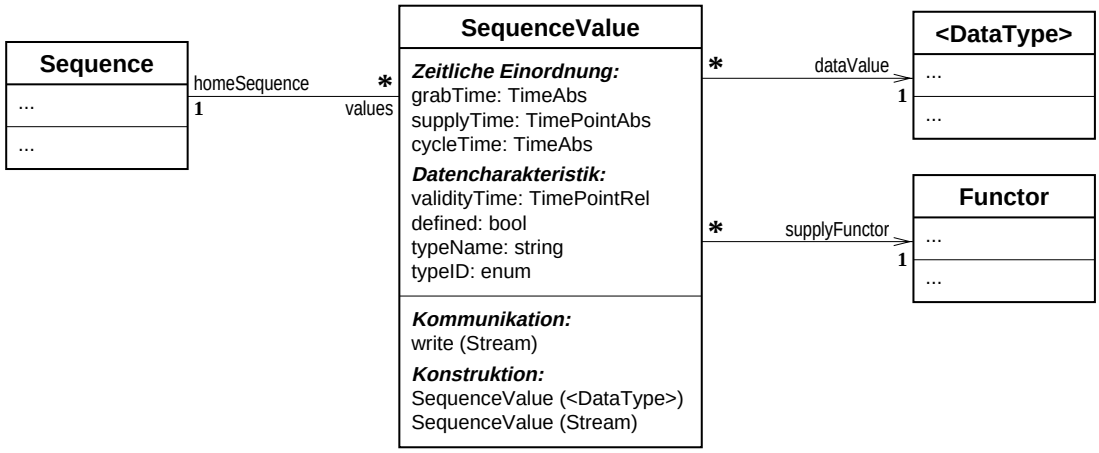


Abbildung 7.2: Klassendiagramm SequenceValue mit den wichtigsten Attributen und Methoden.

Aus diesen Gründen werden, wie Abb. 7.2 zeigt, Daten und Statusinformationen getrennt modelliert und nur für die Repräsentation in einer dynamischen Sensordatenauswertung zusammengeführt. Die *Sequenzwert*-Klasse verbindet die verschiedenen Informationsebenen über eine Assoziationsbeziehung — ein Sequenzwert *verweist auf* ein Datenobjekt und assoziiert es so mit Attributen und anderen Objekten. Durch diesen Mechanismus fungiert die *Sequenzwert*-Klasse für die unterschiedlichsten Datentypen als Containerobjekt. Sie abstrahiert dabei von dem konkreten Datentyp und schafft so ein einheitliches Interface für die dynamischen Aspekte bei der Verarbeitung beliebiger dynamischer Daten mit ihren temporalen Attributen und speziellen Methoden.

Änderungen an den zu kapselnden Datentypen sind dafür nicht notwendig, allerdings müssen in typgebundenen Sprachen, wie C++, die zu verwendenden Datentypen beim Erstellen der Klassenbeschreibung deklariert werden, damit die Klassenhierarchie um spezielle Klassen erweitert werden kann. Diese fungieren in C++ als Brücke zwischen dem typlosen Container SequenceValue und einer konkreten Datenklasse, ggf. auch aller von dieser Klasse abgeleiteten Objektklassen. Realisieren läßt sich dies mit dem *Envelope-Letter-Idiom* [Cop92]. In Abb. 7.3 wird beispielhaft — auch für andere hier zu modellierenden Klassen gezeigt, wie die Umsetzung eines konzeptionellen Entwurfs in C++ mit Hilfe von zusätzlichen Klassen und eines Anwendungsmusters, das deren Design und deren Zusammenspiel vorgibt, erfolgen kann.

Durch dieses Framework wird auch ein anderes Problem gelöst, das in C++ aufgrund der fehlenden automatischen *Garbage Collection* entsteht. In dynamischen Anwendung mit verteilten Aufgabenkompetenzen stellt es ein normales Szenario dar, daß einzelne Objekte nicht fest an andere Instanzen (durch Aggregation oder Komposition) gebunden werden. Sie können, wie die hier betrachteten Sequenzwerte, an einer Stelle im System dynamisch erzeugt und von dort an andere Objekte weitergereicht werden, wobei, aus Konsistenzgründen und um die Identität

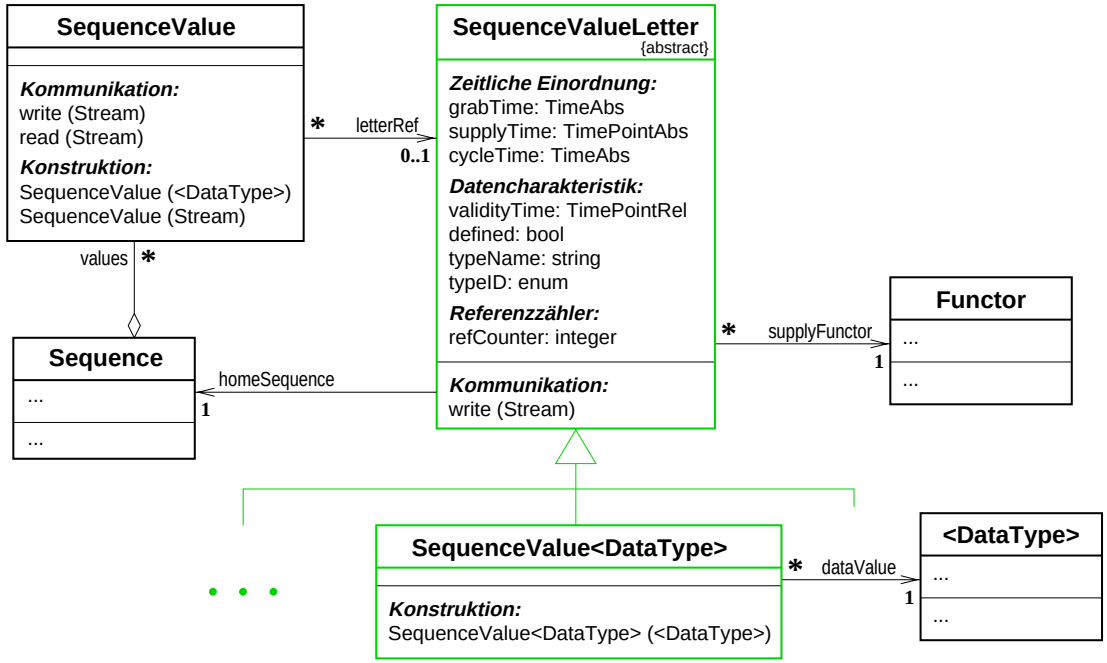


Abbildung 7.3: Klassendiagramm für SequenceValue aus der Implementierungssicht in C++.

der Objekte zu erhalten, ein Kopieren häufig nicht möglich ist. Da diese Objekte veralten und ihr Speicherbereich für neue dynamische Objekte benötigt wird, müssen sie schließlich gelöscht werden, wobei die nicht triviale Frage zu beantworten ist, wann genau dies geschehen kann.

Mit Hilfe des *Envelope-Letter-Idioms* wird dies für die Sequenzwerte folgendermaßen gelöst. Jeder Sequenzwert wird durch genau eine *Letter*-Instanz: SequenceValueLetter repräsentiert, diese ist mit einem Referenzzähler ausgestattet. Objekte, die einen Sequenzwert referenzieren, halten eine lokale SequenceValue-Instanz. Diese nimmt in dem Anwendungsmuster die *Envelope*-Funktion ein. Bei Zuweisungen und Kopien wird der Referenzzähler des entsprechenden *Letter*-Objekts erhöht, überschriebene oder gelöschte Sequenzwertreferenzen vermindern den Zähler des referenzierten *Letter*-Objekts. Wird der Sequenzwert durch kein Objekt mehr referenziert, ist der Zähler Null und das *Letter*-Objekt kann gelöscht werden.

7.3.2 Das Objektmodell

Attribute und Objektreferenzen

Die Einbindung konkreter Daten in eine Sequenzwertinstanz erfolgt über die Instanzvariable dataValue : ⟨DataType⟩. In Abhängigkeit von der Art der Typbehandlung der verwendeten Programmiersprache (Typbindung zur Compile- oder zur Laufzeit, Verfügbarkeit expliziter Typinformationen zur Laufzeit, usw.) kann dies zu unterschiedlichen Implementierungsvarianten führen, worauf hier jedoch nicht näher eingegangen werden soll.

Die von der Sequenzwertklasse verwalteten Attribute dienen in erster Linie der zeitlichen Einordnung der Daten. Für viele Aufgaben ist es wichtig, den Bezug der Daten zur Realzeit zu bewahren und für deren Auswertung zusammen mit den Datenwerten bereitzustellen, so z.B.

SequenceValue: Instanzvariablen	
dataValue : ⟨DataType⟩	Datenobjekt, das als Sequenzwert gekapselt wird.
grabTime : TimeAbs	Datenmeßzeit t_g (oder i_g), zu der ein physikalischer Sensor die zugrundeliegende Szenengröße abgreift.
supplyTime : TimePointAbs	Zeitpunkt der Datenbereitstellung t_s durch den Funktor.
cycleTime : TimeAbs	Zykluszeit t_c, in dem die Datenbereitstellung erfolgte.
validityTime : TimePointRel	Zeitspanne τ_v, die ein Datum nach seiner Erzeugung gültig bleibt. Dieser Wert wird von der die Sequenzwerte verwaltenden Sequenz standardmäßig auf Null gesetzt.
supplyFunctor : Functor	Funktor, von dem der Sequenzwert erzeugt wurde.
defined : bool	Das Flag gibt an, ob der Sequenzwert ein gültiges Datum enthält oder nicht definiert ist.
typeName : string typeID : integer	Kennzeichnung des gekapselten Datentyps, z.B. für die zur Laufzeit notwendige Typkontrolle oder Konvertierung.

Tabelle 7.2: Wichtige Attribute und Objektreferenzen der Sequenzwertklasse SequenceValue.

bei der Analyse von Objektbewegungen, der Bewegungsplanung oder der Planung der Vergabe von Rechenressourcen.

Die Datenmeßzeit grabTime (t_g) stellt dafür den Bezug zwischen einem Datenwert und der Szene her. Über sie kann das Alter eines Datums oder die Zeit zwischen zwei Messungen ermittelt werden, z.B. um den Fortschritt einer realen Aktion bis zum aktuellen Zeitpunkt zu prädizieren oder die Geschwindigkeit extrahierter Objekte zu bestimmen. Dieser Zeitwert wird von den Eingangsdaten eines Funktors an dessen Ausgangsdaten weiterpropagiert (vgl. Abschnitt 5.1.2). Durch die gemeinsame Verarbeitung unterschiedlich alter Daten oder um die Dauer der Aufnahme für jeden Meßwert explizit darzustellen, kann diese Zeit auf ein Intervall i_g ausgedehnt werden. Bei der Bestimmung des Datenalters oder der Gültigkeit der Sequenzwerte wird in diesem Fall vom Ende des Meßintervalls ausgegangen.

Die in einem realen Computersystem durch die Verarbeitung und Auswertung der Daten auftretenden Verzögerung bis zur Verfügbarkeit eines Sequenzwertes wird durch den Zeitpunkt der Bereitstellung des Datenwerts supplyTime (t_s) modelliert. Die Zykluszeit des Bearbeitungszyklus cycleTime (t_c) ergänzt die temporale Datenbeschreibung. Sie hilft, die Rechenzeit, die zwischen dem Start einer Datenanfrage und der Verfügbarkeit einer Antwort vergeht, abzuschätzen. Durch diesen Zeitpunkt wird darüber hinaus die Zuordnung zwischen einem Bearbeitungszyklus und dem Zeitpunkt der Datenmessung in der Szene hergestellt.

Mit Hilfe dieser Zeitwerte lassen sich Aussagen über den Ressourcenverbrauch, insbesondere die Rechenzeit bestimmter Teilaufgaben, z.B. die Extraktion eines bestimmten Merkmals aus den Kamerabildern, machen. Eine Planungskomponente kann diese Informationen nutzen, um die vorhandenen Ressourcen neu zu verteilen, unwichtigere Teilaufgaben zurückzustellen oder zeitaufwendige Verfahren gegen weniger aufwendige Methoden auszutauschen.

Durch zusätzliche Attribute lassen sich weitere Informationen, wie Datentyp und Status (gültiger Wert oder „undefiniert“), Verweise auf die Meßumgebung (Aufnahmesensor, zugrundeliegende Szenendaten und Berechnungsfunktor) oder die Einbindung des Wertes in eine Zeitfolge gleichartiger Daten (Datensequenz, Vorgänger, Nachfolger) explizit ausdrücken. Die

wichtigsten Attribute der Klasse SequenceValue sowie deren Referenzen zu anderen Objekten werden in Tabelle 7.2 zusammengefaßt.

Konstruktoren und Instanzmethoden

Erzeugen läßt sich eine neue Sequenzwertinstanz direkt aus dem zu kapselnden Datum, oder aber aus einem Datenstrom, den ein Funktor aus einer Datei oder einer Netzwerkverbindung zu einem anderen Rechner liest. Der SequenceValue-Konstruktor, dem dieses Stream-Objekt übergeben wird, liest aus ihm die Typkennung, die Rohdaten und die Zeitcharakteristik, so daß er den Ausgangswert vollständig *re*-konstruieren kann.

Zusammen mit der write()-Methode steht damit ein Basismechanismus zur Verfügung, um Datensequenzen abzuspeichern oder in einem Rechnernetz zu verteilen, damit sie später — für eine Simulation — oder sofort — in einem anderen Prozeßraum — rekonstruiert werden können. Im RoboCup wird dies genutzt, um zwischen den verschiedenen Robotern sowie zwischen den Robotern und externen Rechnern, die die Visualisierung der Daten oder übergeordnete Planungsaufgaben durchführen, wichtige Daten, wie die extrahierten Hindernisse, den Ball und die Roboterpositionen auszutauschen. Diese Daten stehen dadurch auf jedem Rechner wie lokale Sensordaten zur Verfügung und können in das eigene Szenenmodell eingearbeitet werden.

SequenceValue: Methoden	
write(Stream)	Schreibt Datenwert, Typkennung und Zeitcharakteristik in einen Datenstrom, um das Objekt abzuspeichern oder über eine Netzwerkverbindung auf anderen Rechnern zu spiegeln.
SequenceValue(Stream)	Klassenmethode zum Anlegen eines neuen Sequenzwertes aus den Daten einer Datei oder dem über eine Datenverbindung von einem anderen Rechner gesendeten Datenstrom.
SequenceValue(⟨DataType⟩)	Anlegen eines Sequenzwertes mit dem übergebenen Datum.

Tabelle 7.3: Wichtige Methoden der Sequenzwertklasse SequenceValue.

Zugriffe auf den Datenwert können zum einen als Lesen der Instanzvariable dataValue oder aber als Typkonvertierung SequenceValue in ⟨DataType⟩ interpretiert werden. Ersteres entspräche einer Methode getDataValue : ⟨DataType⟩(). Diese Möglichkeit scheidet jedoch aus, da es in C++ nicht möglich ist, die getDataValue()-Methode für die Klasse SequenceValue mit unterschiedlichen Rückgabetypen zu definieren. Aus diesem Grund erfolgt der Datenzugriff mit einer Methode as ⟨DataType⟩() : ⟨DataType⟩. Dabei wird automatisch eine Typüberprüfung durchgeführt, die sicherstellt, daß nachfolgende Operationen nur Daten des richtigen Typs erhalten.

7.4 Modellierung von Datensequenzen

Datenfolgen werden explizit als *ein* Sequenzobjekt **S** modelliert. Die Sequence-Klasse faßt dafür die Eigenschaften und Methoden dynamischer Datenfolgen zusammen und abstrahiert

dabei vollständig von dem konkreten Datentyp der einzelnen Folgenelemente, d.h. es werden von der Sequenzklasse keine spezielleren Sequenzen für Bild- oder Regionenfolgen abgeleitet.

Die Modellierung orientiert sich dabei weitestgehend an den Entwurfskriterien für Datensequenzen, die in den Abschnitten 4.2 (Attribute und Aktionen), 5.1 (Zeitcharakteristik) und 6.3 (funktionale Beschreibung und Programmsteuerung) zusammengetragen wurden. Eine Instanz der Sequenzklasse repräsentiert dabei ein diskretes Sensorsignal. Dieses wird im wesentlichen charakterisiert durch die Gesamtheit der aufeinanderfolgenden Datenwerte sowie durch Datenflußbeziehungen und Programmsteuerungsrelationen zu anderen Objekten, welche sich aus der Einbindung der Datenfolge in einen Datenflußgraphen ergeben.

7.4.1 Das Objektmodell

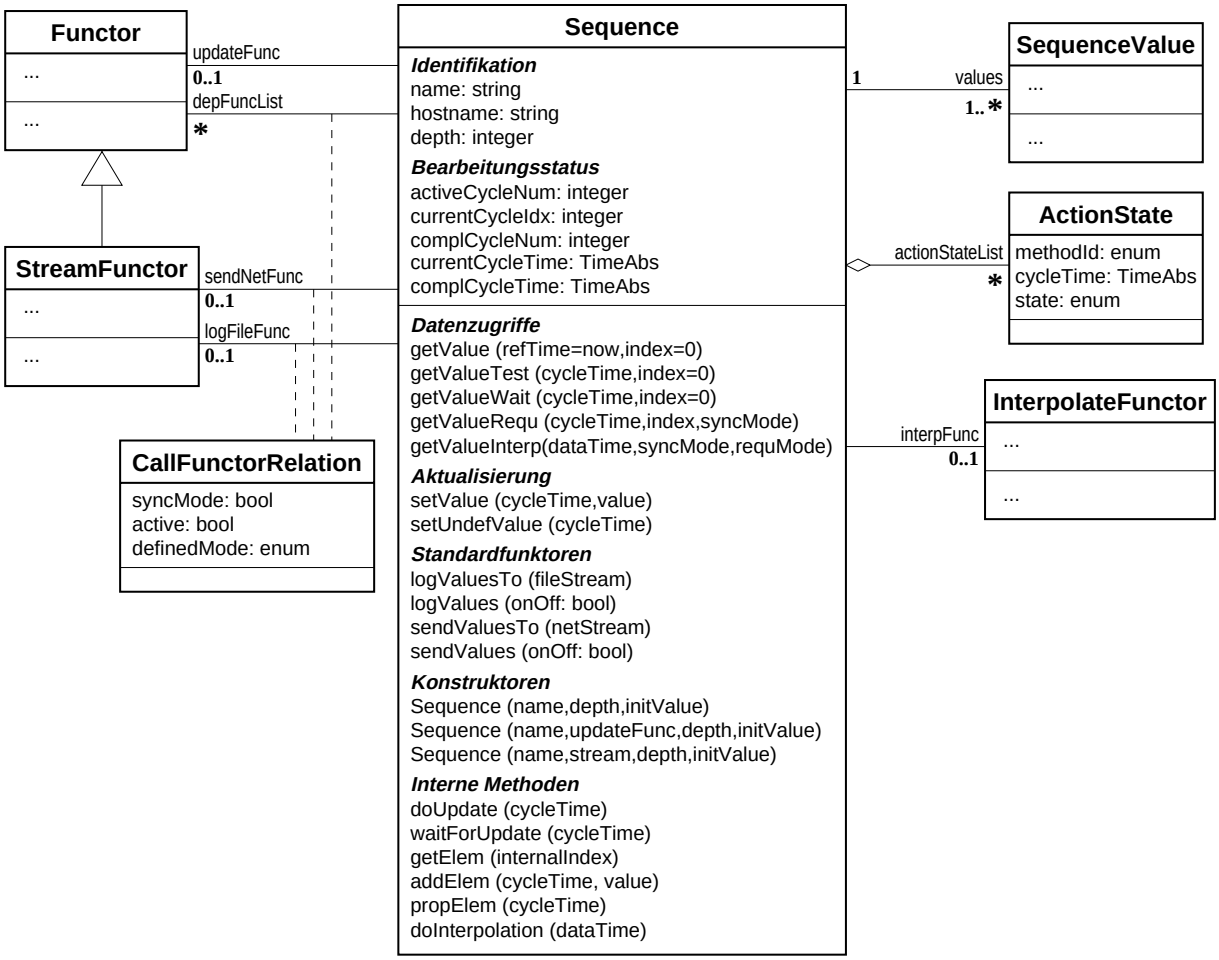


Abbildung 7.4: Sequence-Klassendiagramm mit den wichtigsten Attributen und Methoden.

Attribute und Objektreferenzen

Ein diskretes Sensorsignal wird in erster Linie durch seine Werte, die nacheinander eintreffen, charakterisiert. Das in Abschnitt 4.2 definierte Sequenzmodell stellt die Grundlage für die ob-

Sequence: Instanzvariablen (1)	
values : SequenceValue[] (SequenceValueArray)	Sequenzwertliste; die Realisierung erfolgt hier als Ringarray, gekapselt in einer eigenen Klasse SequenceValueArray.
depth : integer	Anzahl gleichzeitig zu speichernder Sequenzwerte — entspricht der Größe der als Ringpuffer realisierten Werteliste.
activeCycleNum : integer	Anzahl der aktuell aktiven Zyklen.
complCycleNum : integer	Anzahl der abgeschlossenen Zyklen.
currentCycleIdx : integer	Index I des aktuellen Bearbeitungszyklus.
complCycleTime : TimeAbs	Zykluszeit des zuletzt gesetzten, (aktuellsten) Sequenzwertes.
currentCycleTime : TimeAbs	Zykluszeit des aktuellen Bearbeitungszyklus.
actionStateList : ActionState[] (ActionStateList)	Bearbeitungsstand der Sequenz; Liste mit den aktuell laufenden Aktionen, deren Status und der Zykluszeit.
interpValue : SequenceValue	Zuletzt interpolierter Sequenzwert.
name : string	Die Semantik der Datenfolge bezeichnender Name.
hostname : string	Rechner, auf dem sich das Original der Sequenz befindet.

Tabelle 7.4: Attribute der Sequence-Klasse für die Verwaltung der Sequenzwerte. Für die hier und in den folgenden Tabellen in Klammern gesetzten Klassennamen gilt, daß sie sich aus der Spezifizierungs- bzw. Implementierungsebene ergeben. Insbesondere bei Datenlisten oder Arrays ist gegenüber der konzeptionellen Entwurfsebene i.d.R. die Einführung zusätzlicher Klassen erforderlich.

jektorientierte Modellierung dar, es geht jedoch von der idealisierten Annahme aus, daß eine Sequenz *alle* Daten eines diskreten Signals repräsentiert. In einem realen System ist dies aufgrund der nur begrenzt verfügbaren Ressourcen natürlich nicht möglich. Daher beschränkt sich das Softwaremodell auf die Verwaltung der letzten depth Sequenzwerte. Diese werden in dem realisierten System zyklisch in ein Ringarray (values : SequenceValueArray) geschrieben. Da die Größe des Arrays frei wählbar ist und auch zur Laufzeit verändert werden kann, schränkt dies die praktische Anwendbarkeit des Ansatzes nicht ein. Alternativ dazu könnten die Sequenzwerte auch in einer verketteten Liste verwaltet werden, hinsichtlich der Laufzeit stellt dies jedoch die ungünstigere Variante dar.

Für die Programmsteuerung ist es notwendig den aktuellen Zustand der Sequenz bestimmen zu können. Dieser wird durch den letzten abgeschlossenen Zyklus sowie durch die aktuell laufenden Aktionen beschrieben. Ein aktiver Bearbeitungszyklus kann durch seinen absoluten Index I: currentCycleIdx und die Zykluszeit t_c: currentCycleTime charakterisiert werden. Sind aufgrund von Pipelining-Verarbeitung mehrere Zyklen gleichzeitig aktiv, bezeichnen currentCycleIdx und currentCycleTime den zuerst gestarteten und noch aktiven Zyklus, die Anzahl der *aktiven* Zyklen steht in activeCycleNum. Dabei ist ein Bearbeitungszyklus vom Beginn einer insistenten Datenanfrage bis zur Datenbereitstellung aktiv. Wird die Sequenz durch den Funktor aktualisiert, ohne daß dem ein Zugriff auf diese Sequenz vorausgegangen ist, ist sie nur während des Setzens des Wertes mit der setValue()-Methode aktiv.

Die Instanzvariable complCycleNum enthält die Anzahl aller Sequenzwerte seit Beginn der Messung. Nachdem eine Aktualisierung abgeschlossen wurde, wird dieser Wert angepaßt und erhält den Index des beendeten Zyklus: complCycleNum := currentCycleIdx. Die entsprechende

Zykluszeit wird in complCycleTime := currentCycleTime übertragen. Sind weitere Zyklen aktiv, wird currentCycleIdx inkrementiert, ansonsten behält es seinen Wert bis ein neuer Zyklus gestartet wird.

Der Status laufender Aktionen wird in einer Aktionsliste actionStateList : ActionState[] verwaltet. Diese enthält eine Aufzählung der Aktionen, die entweder aktiv die Aktualisierung der Sequenz beeinflussen (setValue(), doUpdate(), ...) oder deren erfolgreiche Ausführung im Rahmen der Konflikterkennung (vgl. Abschnitt 6.4.5) überwacht werden soll (z.B. getValueWait()). Jeder Listeneintrag enthält eine Aktionskennung (methodId), die Zykluszeit (cycleTime) und den Status der Aktion (state : enum{started, running, canceled, finished, error}). Nach erfolgreicher Beendigung einer Aktion oder deren Restart wird der Eintrag wieder aus der Liste entfernt.

Für die eindeutige Identifizierung einer Sequenz im Programm und in einem Rechnerverbund ist deren Kennzeichnung mit einem eindeutigen Bezeichner wichtig. Dieser setzt sich aus einem Namen, der die Semantik der Datenfolge bezeichnet: name, und dem Rechnernamen der Originalsequenz: hostname zusammen. Eine Übersicht über die wichtigsten, für die Verwaltung der Sequenzwerte notwendigen Attribute und Instanzvariablen wird in Tabelle 7.4 gegeben.

Tabelle 7.5 gibt einen Überblick über die Zuordnung von Funktorobjekten zu einer Sequenz. Das Klassendiagramm in Abb. 7.4 zeigt, daß keine dieser Relationen verbindlich die Zuweisung eines Funktors erfordert. Die für die Steuerung der Sequenz relevanten Sequenz-Funktor-Relationen (trig) und (upd) (vgl. Abschnitt 6.3.3) werden gesetzt, indem die entsprechenden Funktorreferenzen in die Instanzvariablen depFuncList ($\{\mathbf{F_{dep}}\}$) und updateFunc ($\mathbf{F_{upd}}$) eingetragen werden. Fehlt dieser Eintrag, ist die Relation nicht gesetzt.

Sequence: Instanzvariablen (2)	
updateFunc : Functor	Aktualisierungsfunktor $\mathbf{F_{upd}}$ zur Bereitstellung neuer Sequenzwerte.
depFuncList : Functor[] (CallFunctorRelationList)	Menge der nach einer Sequenzaktualisierung zu triggernden Folgefunktoren $\{\mathbf{F_{dep}}\}$; realisiert als Relationen-Liste.
logFileFunc : StreamFunctor (CallFunctorRelation)	Folgefunktor ($\mathbf{F_{Log}} \in \{\mathbf{F_{dep}}\}$) zum Protokollieren der Sequenzwerte in eine Datei.
sendNetFunc: StreamFunctor (CallFunctorRelation)	Folgefunktor ($\mathbf{F_{Send}} \in \{\mathbf{F_{dep}}\}$) zum Versenden der Sequenzwerte über eine Netzverbindung an andere Rechner.
interpFunc : InterpolateFunctor	Interpolationsfunktor $\mathbf{F_{int}}$; kapselt Funktion zum Interpolieren der in der Sequenz verwalteteten Datenwerte.

Tabelle 7.5: Funktorassoziationen der Sequence-Klasse für die Programmsteuerung.

Instanzen der Relationsklasse CallFunctorRelation erlauben es, jede Relation einzeln zu parametrisieren, beispielsweise um den Funktor synchron und asynchron aufzurufen, um das Verhalten bei undefinierten Datenwerten zu bestimmen oder aber um eine Relation zu deaktivieren, ohne den Funktor aus der Liste entfernen zu müssen. Für den Aktualisierungsfunktor updateFunc ist eine solche parametrisierte Assoziation nicht notwendig, da dessen Aufruf immer synchron erfolgen soll und an keine besonderen Bedingungen geknüpft ist.

Um eine Datenfolge vollständig aufzuzeichnen, kann sie mit Hilfe eines Folgefunktors $\mathbf{F_{Log}}$ in eine Datei protokolliert werden. Soll die Sequenz auch auf anderen Rechnern im Netz verfügbar sein, ist ein Versenden der neuen Datenwerte an einzelne oder alle vernetzte Rechner

möglich, wofür ebenfalls ein Folgefunktor $\mathbf{F_{Send}}$ verwendet werden kann. Diese beiden Folgefunktoren könnten mit den anderen, entsprechend Abschnitt 6.4.7 zu triggernden Funktoren in der Liste depFuncList verwaltet werden: $\mathbf{F_{Log}}, \mathbf{F_{Send}} \in \{\mathbf{F_{dep}}\}$. Aufgrund ihrer besonderen Stellung sollen sie hier jedoch explizit benannt und über spezielle Instanzvariablen logFileFunc und sendNetFunc separat behandelt werden. Dies erlaubt es, die beiden Funktionalitäten Datenprotokollierung und Datenspiegelung zum einen explizit mit den Mitteln der funktionalen Datenflußbeschreibung darzustellen (Abb. 7.5 a) und sie andererseits als integrale Eigenschaft der Sequenzklasse aufzufassen (Abb. 7.5 b).

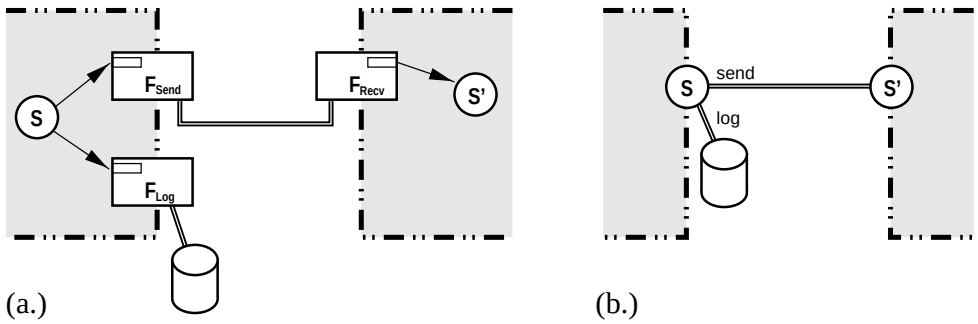


Abbildung 7.5: Darstellung der für Datensequenzen relevanten Funktionalitäten „Protokollierung“ (log) und „Spiegelung“ (send) als Elemente des Datenflußgraphen (a.) und als Eigenschaft der Klasse Sequence (b.).

Wird durch die Verwendung der entsprechenden Zugriffsmethoden die Interpolation von Sequenzwerten gefordert, muß die Sequenz mit einem geeigneten Interpolationsfunktor assoziiert werden. Auf diesen kann mit Hilfe der interpFunc-Variable verwiesen werden.

Methoden

Die Methoden der Sequenzklasse lassen sich verschiedenen Aufgabenbereichen zuordnen. Die wichtigsten sind:

Konstruktion: Erzeugen von Sequenzen, ggf. in einem bestimmten Bearbeitungskontext.

Konfiguration: Setzen oder Modifizieren des Aufrufkontexts durch Spezifizieren von Datenflußbeziehungen und Steuerungsrelationen.

Datenfluß: Lesende und schreibende Sequenzwertzugriffe im Rahmen der zyklischen Datenverarbeitung. Hierzu gehören in erster Linie die in Abschnitt 4.2.2 modellierten Zugriffsmethoden einschließlich deren Aufteilung in die internen Teilaktionen.

Erzeugt wird eine neue Sequenzinstanz mit Hilfe von Konstruktormethoden, die wichtigsten sind in Tabelle 7.6 aufgelistet. Der Standardkonstruktor erwartet einen Namen für die Sequenz, die Größe der Sequenzwertliste und einen Initialisierungswert. Ausgehend von diesem Standardkonstruktor werden weitere Konstruktoren bereitgestellt, die die Programmentwicklung vereinfachen und mehrere Schritte zusammenfassen.

Durch die Übergabe des Aktualisierungsfunktors kann so z.B. bereits die Einbindung der Sequenz in den Datenflußgraphen erfolgen. Dies setzt voraus, daß der Funktor vor der Sequenz

Sequence: Konstruktor-Methoden
Sequence (name : string, depth : integer, initValue : SequenceValue) Standardkonstruktor; erzeugt eine Sequenzinstanz mit dem Namen name. Die depth Sequenzwerte, die die Sequenz gleichzeitig in values verwalten kann, werden mit initValue initialisiert. Ein Aktualisierungsfunktor wird nicht gesetzt.
Sequence (name : string, updateFunc : Functor, depth : integer, initValue : SequenceValue) Erzeugt eine Sequenzinstanz und setzt den Aktualisierungsfunktor.
Sequence (name : string, stream : Stream, depth : integer, initValue : SequenceValue) Sequence (name : string, fg : Framegrabber, depth : integer, initValue : SequenceValue) Spezielle Sequenzkonstruktoren für häufig auftretende Standardanwendungen. Der Aktualisierungsfunktor updateFunc wird zusammen mit der Sequenz erzeugt und ist Instanz einer speziellen, vordefinierten Funktorklasse. Die Sequenzwerte werden von ihm entweder aus einem Dateistrom (aus Dateien oder vom Netz) empfangen: StreamFunctor (stream : Stream) oder von einem Framegrabber eingelesen: FgFunctor (fg : Framegrabber).

Tabelle 7.6: Methoden der Sequence-Klasse zum Erzeugen neuer Sequence-Instanzen.

erzeugt wurde. Ist die neu erzeugte Sequenz die einzige Ausgangssequenz des Funktors, kann auch der Funktor sofort mit ihr assoziiert, d.h. die (einzige) Funktor-Ausgangssequenz-Relation des Funktors gesetzt werden. Anderenfalls muß, wie beim Standardkonstruktor, die Assoziation zwischen einem Ausgangstor des Funktors und der Sequenz später in einem eigenen Schritt erfolgen.

In vielen Anwendungen werden einzelne Sequenzen in einem immer gleichen Standardkontext betrieben. Die häufigsten sind die Rekonstruktion einer Sequenz aus einer Datei, das Anlegen des Spiegelbilds einer Sequenz aus einem anderen Prozeßraum und in der Bildverarbeitung das Bereitstellen von Kamerabildern in einer Bildfolge. Für diese Anwendungsmuster bietet es sich an, spezielle Konstruktoren bereitzustellen, die automatisch einen geeigneten, von der Klasse Functor abgeleiteten Aktualisierungsfunktor erzeugen. Diese übernehmen dann die Aufgabe, die Sequenzdaten aus einer Datei zu lesen, über eine Netzverbindung zu empfangen oder Kamerabilder direkt von einem Framegrabber entgegenzunehmen. Da diese Funktoren i.d.R. nur eine Ausgangsdatensequenz besitzen, kann hier die Funktor-Ausgangssequenz-Relation automatisch in den neu erzeugten Funktor eingetragen werden.

Soll der Datenflußgraph beim Programmstart automatisch aus einer formalen, textuellen Beschreibung, die beispielsweise in einer Konfigurationsdatei vorliegt, aufgebaut werden, kann dies am einfachsten mit Hilfe der Standardkonstruktoren erfolgen. In einem ersten Schritt werden alle Sequenzen und Funktoren unabhängig voneinander erzeugt. Daran anschließend werden sie in einem zweiten Schritt durch Setzen der entsprechenden Datenfluß- und Steuerungsbeziehungen miteinander verknüpft. Andere Programmkomponenten können auf die Sequenzen über deren Namen und eine global verankerte Liste, in die alle Sequenzen bei ihrer Erzeugung eingetragen werden, zugreifen. In C++ bietet es sich an, diese Liste in der Sequenz-*Klasse* zu verankern, der Zugriff darauf ist dann über Sequence-Klassenmethoden möglich.

Eine zweite Gruppe von Sequenzmethoden dient der Konfiguration der Daten- und Kontrollflußbeziehungen. Setzen und ändern lassen sich die Sequenz-Funktor-Relationen dadurch, daß die zu assoziierenden Funktoren über die entsprechenden Variablen updateFunc und depFuncList

referenziert werden, wofür verschiedene Standardmethoden entsprechend Abschnitt 7.1 zur Verfügung stehen. Die Eigenschaften der Sequenz-Folgefunktor-Relationen können in den CallFunctorRelation-Instanzen, die Folgefunktoren referenzieren, gesetzt werden. Diese bestimmen, ob ein Folgefunktor synchron oder asynchron getriggert wird. Weiterhin legen sie fest, ob der Funktor nur bei definierten, nur bei undefinierten oder aber bei allen Ausgabewerten aufgerufen werden soll. Auf diese Weise können leicht Ausnahmebehandlungen definiert werden, die z.B. greifen, wenn ein Merkmal nicht bestimmt werden konnte.

Sequence: Kontext-Methoden
logValuesTo (fileStream : Stream) Erzeugt einen Funktor für die Datenprotokollierung und trägt ihn in logFileFunc ein.
logValues (onOff : bool) Ein- oder Ausschalten der Datenprotokollierung.
sendValuesTo (netStream : Stream) Erzeugt einen Funktor für die Datenübertragung und trägt ihn in sendNetFunc ein.
sendValues (onOff : bool) Ein- oder Ausschalten der Datenübertragung.

Tabelle 7.7: Methoden der Sequence-Klasse für die Modifikation des Kontexts der Sequenz.

Neben den Standardmethoden zum Setzen von allgemeinen Folgefunktoren können die beiden speziellen Folgefunktoren logFileFunc und sendNetFunc zum Protokollieren bzw. Versenden der Daten (vgl. Abb. 7.5) mit den Methoden logValuesTo (Stream) und sendValuesTo (Stream) separat erzeugt und mit logValues (bool) bzw. sendValues (bool) gestartet bzw. beendet werden. In Tabelle 7.7 werden die entsprechenden Methoden zusammengefaßt.

Der dritte Aufgabenbereich umfaßt Methoden, mit denen in den regulären Datenverarbeitungszyklen in Abhängigkeit vom festgelegten Daten- und Kontrollfluß lesend oder schreibend auf die Werte der Sequenz zugegriffen werden kann. In Tabelle 7.8 werden die Methoden mit den entsprechenden Parametern beschrieben. Hierzu gehören lesende Datenzugriffe mit den verschiedenen getValue$*$()-Methoden und das Eintragen neuer Sequenzwerte mit setValue (). Diese entsprechen direkt den in Abschnitt 4.2.2 vorgestellten set()- und get()-Methoden. Konnte ein Datenwert in einem Zyklus nicht bestimmt werden und ist undefiniert, kann dies der Sequenz mit setValueUndef () angezeigt werden. Der Aufruf erzeugt einen Sequenzwert mit undefiniertem Datum und trägt ihn in die Liste ein.

Interne Methoden und Semaphore

Am Anfang jeder getValue$*$()-Zugriffsmethode testet eine Statusanalyse, ob ein Wert mit der internen getElem()-Methode zurückgeliefert werden kann und welcher dies ist. Steht der geforderte Wert nicht zur Verfügung, wird — abhängig von der konkreten Methode und ggf. von bereits laufenden Aktionen — bestimmt, wie der Aufruf fortzusetzen ist: doUpdate(), waitForUpdate(), doInterpolation() oder Abbruch.

Im Statustest der setValue()-Methode wird überprüft, ob mit der internen addElem()-Methode ein neuer Wert in die Liste eingetragen werden kann. Dies schlägt fehl, wenn für die angegebene Zeit bereits ein Wert gesetzt wurde. Im aktuellen System wird auch das Einfügen

Sequence: Zugriffs-Methoden
getValue (index : integer = 0) : SequenceValue getValue (refTime : TimePointAbs, index : integer = 0) : SequenceValue Einfache Sequenzwertabfrage $get_{[]}()$ — geht vom aktuellen Zustand der Sequenz beim Methodenaufruf bzw. zur angegebenen Referenzzeit aus. Alle getValue$*$()-Aktionen greifen mit der internen getElem()-Methode auf die Datenliste zu. getValueTest (cycleTime : TimeAbs, index : integer = 0) : SequenceValue Zeitgebundener Datenzugriff $get_t^?()$ — Abbruch bei fehlendem Wert. getValueWait (cycleTime : TimeAbs, index : integer = 0) : SequenceValue Zeitgebundener, wartender Datenzugriff $get_t^{\bowtie}()$ — fehlt der Wert, wartet die Methode, bis er gesetzt wird. getValueRequ (cycleTime : TimeAbs, index : integer = 0, syncMode : bool) : SequenceValue Zeitgebundener, insistenter Datenzugriff $get_t^{\lhd}()$ — fehlt der Wert, leitet die Sequenz mit der internen doUpdate()-Methode die Datenaktualisierung ein. getValueInterp (dataTime:TimePontAbs, syncMode: bool, requMode: enum): SequenceValue Datenzugriff mit Interpolation $get_t^{\sim}$, $get_t^{\widetilde{\bowtie}}$, $get_t^{\widetilde{\lhd}}$. Der Parameter requMode: enum {predict, wait, request} wählt das konkrete Zugriffsverfahren aus und spezifiziert so das Verhalten falls dataTime $>$ values[0].grabTime ist. setValue (cycleTime : TimeAbs, value : SequenceValue) Setzen eines neuen Sequenzwertes — dabei wird mit der internen addElem()-Methode zuerst der neue Wert in die Datenliste eingefügt und anschließend mit der propElem() an die Folgefunktoren weiter propagiert. setValueUndef (cycleTime : TimeAbs) Setzen eines undefinierten Sequenzwertes für die angegebene Zeit.

Tabelle 7.8: Methoden der Sequence-Klasse für Zugriffe auf die Sequenzwertliste.

Sequence: interne Methoden
doUpdate (cycleTime : TimeAbs) Aktualisierung der Sequenz durch Aufruf des Aktualisierungsfunktors. getElem (cycleTime : TimeAbs, index : integer) : SequenceValue Lesender Zugriff auf die interne Sequenzwertliste values. addElem (cycleTime : TimeAbs, value : SequenceValue) Hinzufügen eines neuen Sequenzwertes in die interne Liste values. Ein neuer Wert überschreibt dabei immer den ältesten in der Liste. propElem (cycleTime : TimeAbs) Propagieren neuer Sequenzwerte durch Triggern der Folgefunktoren logFileFunc, sendNetFunc und depFuncList. doInterpolation (dataTime : TimePointAbs) : SequenceValue Aufruf des Interpolationsfunktors. Der interpolierte Rückgabewert wird in interpValue für erneute Zugriffe mit derselben Datenzeit zwischengespeichert.

Tabelle 7.9: Interne Methoden der Sequence-Klasse.

alter Werte in die Liste nicht unterstützt. Zum Abschluß des Statustests wird die ausgewählte Aktion, falls sie für andere Methoden oder für eine Konfliktanalyse wichtig ist, in die Aktionsliste actionStateList eingetragen. Dies betrifft vor allem die doUpdate()-, doInterpolation()- und addElem()-Aufrufe, da sie den Status der Sequenz verändern und so das Verhalten anderer Aktionen beeinflussen können. Darüber hinaus ist das Eintragen von waitForUpdate()-Aufrufen und des Abbruchs der getValueTest()-Methode sinnvoll, um Konflikte, die den regulären Abschluß von Bearbeitungszyklen verhindern, erkennen zu können.

Die Sequenzaktualisierung wird mit der internen propElem()-Methode abgeschlossen. Deren Aufgabe ist es, neue Sequenzwerte im Datenflußgraphen weiterzupropagieren. Falls die entsprechenden Funktoren gesetzt wurden, wird zuerst der logFileFunc-Funktor gestartet, der den neuen Wert in eine Datei schreibt. Daraufhin sendet der sendNetFunc-Funktor die Daten über eine Netzverbindung an andere Rechner, und schließlich werden nacheinander die in der depFuncList-Liste eingetragenen Folgefunktoren getriggert. Wurde in den entsprechenden Funktoraufruf-Relationen der asynchrone Aufrufmodus gesetzt, wird jede dieser Aktionen in einem eigenen *Thread* gestartet, so daß die Funktoren parallel aufgerufen werden.

Die Statustests am Methodenbeginn erfolgen in einem durch ein sequenzweit gültiges Semaphor[1] semaState geschützten Bereich, so daß in einer Sequenz immer nur eine Methode gleichzeitig den Statustest durchführen kann. Zugriffe auf die interne Sequenzliste mit addElem() und getElem() erfolgen, um Inkonsistenzen in der Datenliste zu vermeiden, ebenfalls exklusiv, was mit Hilfe eines weiteren sequenzeigenen Datensemaphors semaData sichergestellt wird. Die Interpolationsmethode doInterpolation() wird durch semaInterp geschützt, damit parallel gestartete Interpolationszugriffe auf das Ende des ersten Aufrufs warten und dessen Ergebnis direkt verwenden können.

Sequence: Semaphore	
semaState : Semaphore	Statussemaphor $\zeta_{\mathbf{S}}^{State}$; zum Absichern der exklusiven Zustandsanalyse beim Aufruf der Zugriffsmethoden.
semaData : Semaphore	Datensemaphor $\zeta_{\mathbf{S}}^{\mathsf{values}}$; zum Absichern exklusiver Zugriffe auf die interne Sequenzwertliste.
semaInterp : Semaphore	Interpolationssemaphor $\zeta_{\mathbf{S}}^{\mathsf{int}}$; um sicherzustellen, daß immer nur ein Interpolationsaufruf aktiv ist.

Tabelle 7.10: Semaphore zum Schutz von Programmbereichen in der Sequence-Klasse.

7.5 Modellierung von Operatoren durch Funktoren

Funktoren $\mathbf{F}$ haben die Aufgabe, beliebige Operatoren oder Funktionen F in einen Datenflußgraphen einzubinden. Sie repräsentieren neben einer gewissen Funktionalität auch den Kontext der Funktionen, d.h. die oft über einen längeren Zeitraum relativ konstanten Beziehungen zu anderen Objekten des Datenflußgraphen, insbesondere zu den benachbarten Eingangs- und Ausgangsdaten $\{\mathbf{S_{In}}\}$, $\{\mathbf{S_{Out}}\}$. Weiterhin wird durch die explizite Modellierung von temporalen

[1] In der aktuellen C++-Implementierung werden Semaphore durch *Mutex*-Variablen nachgebildet.

Attributen die Grundlage für eine Analyse des Zeitverhaltens der Datenverarbeitungsprozesse und der Systemperformanz gelegt. Darauf aufbauend kann beispielsweise die Vergabe von Rechnerressourcen oder die Auswahl konkreter Datenverarbeitungsverfahren geplant werden.

Intern fassen Funktoren alle für die Realisierung einer bestimmten Teilaufgabe notwendigen Objektassoziationen zusammen und repräsentieren nach außen nur die geforderte Funktionalität. Dadurch erlauben sie es, von der konkreten Umsetzung eines Verfahrens mit Hilfe verschiedener Objekte und Operatoren zu abstrahieren und eine einheitliche Schnittstelle für die Einbindung und den Aufruf beliebiger Funktionalitäten bereitzustellen. Mit den Funktoren können so leicht einzelne Verfahren komplett ausgetauscht und durch andere Lösungen ersetzt werden.

Die objektorientierte Modellierung von Funktoren mit den für eine dynamische Sensordatenanalyse notwendigen Methoden basiert im wesentlichen auf dem im Abschnitt 4.3.4 vorgestellten Funktormodell, den in Abschnitt 5.2 untersuchten Zeitattributen sowie den für die Ablaufsteuerung relevanten Funktor-Sequenz-Relationen (vgl. Abschnitt 6.3.3).

7.5.1 Das Objektmodell

Ausgangspunkt der Funktormodellierung ist die Basisklasse Functor. Sie repräsentiert alle Attribute, Objektreferenzen und Methoden, die für die dynamische Sensordatenverarbeitung notwendig sind und die Einbindung des Funktors in einen durch den Datenflußgraphen vorgegebenen Kontext beschreiben. Dazu zählen Assoziationen mit den im Datenflußgraphen benachbarten Datensequenzen sowie Attribute, die den Status des Funktors und dessen Zeitcharakteristik ausdrücken.

Des weiteren stellt die Functor-Klasse einen Anwendungsrahmen für zyklische Funktionsaufrufe in einem Datenflußgraphen zur Verfügung. Dieser stellt sich nach außen in Form einer einheitlichen Aufrufschnittstelle — der call()-Methode — dar, die intern mit einem festen Bearbeitungsschema verbunden ist. Es besteht entsprechend Abschnitt 4.3.5 aus der Aufrufvorbereitung pre(), dem eigentlichen Funktionsaufruf do() und der abschließenden Datenaufbereitung post(). Für eine anschließende Auswertung des Zeitverhaltens des Funktors wird vor und nach den einzelnen Methoden jeweils die aktuelle Zeit bestimmt und aufgezeichnet: t_{call}, t_{do}, t_{post} und t_{leave}.

Jeder einzelne Funktoraufruf wird von einer eigenen Bearbeitungsinstanz (FunctorCall) begleitet. Während der Funktor selbst den relativ langlebigen Kontext der Funktionseinbindung in ein System darstellt, repräsentieren FunctorCall-Objekte den kurzlebigen Kontext eines konkreten Funktionsaufrufs. Dazu werden sowohl die beim Aufruf verwendeten Eingangs- und die durch ihn erzeugten Ausgangssequenz*werte* verwaltet als auch zusätzliche Attribute, die eine zeitliche Einordnung des Aufrufs und die anschließende Auswertung des Zeitverhaltens ermöglichen sowie den aktuellen Bearbeitungsstand — die aktive Methode und deren Status — anzeigen. Dadurch wird die parallele Bearbeitung mehrerer Funktoraufrufe unterstützt — eine wichtige Voraussetzung um bei der anfragegetriebenen Programmsteuerung die Funktoren im Pipeliningbetrieb bearbeiten zu können. Analog zur Verwaltung der Datenwerte in einer Sequenz verwaltet der Funktor stets die letzten callDepth Funktionsinstanzen in einem Ringarray.

Die Klasse Functor verwaltet ausschließlich Attribute und Methoden, die sich aus dem allgemeinen Funktormodell ergeben und für alle Funktoren gleichermaßen von Bedeutung sind. Das schließt eine Aufrufschnittstelle für die zu kapselnden Verfahren ein, die Funktionalität sel-

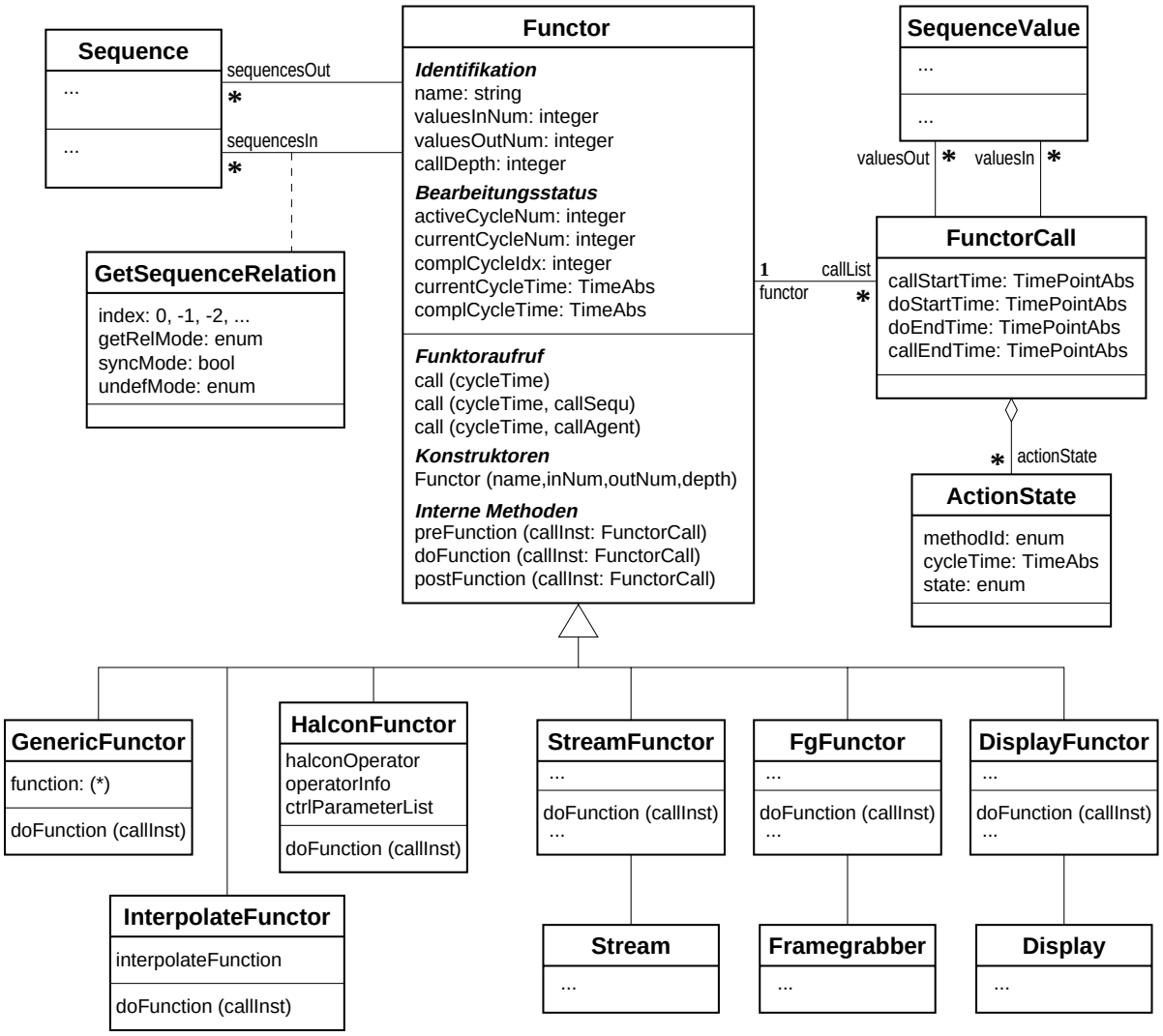


Abbildung 7.6: Funktor-Klassendiagramm. Die Basisklasse Functor definiert wesentliche Objektassoziationen, Attribute und Methoden, die abgeleiteten Klassen stellen die Grundlage zum Einbinden verschiedener Funktionalitäten dar.

ber wird jedoch nicht direkt in der Functor-Klasse verwaltet. Diese Aufgabe wird von eigenen, von Functor abzuleitenden Klassen übernommen, wobei sich zwei Vorgehensweisen anbieten: (1.) die Ableitung einer speziellen Funktorklasse (oder ganzen Klassenhierarchien) für jede einzelne Aufgabe und (2.) die Definition von generischen Funktorklassen, deren Instanzen mit den zu kapselnden Funktionen bzw. Operatoren assoziiert werden.

Der erste Weg erlaubt es, beliebige Operationen zusammenzufassen, wofür die interne doFunction()-Methode in den abgeleiteten Klassen zu überschreiben ist. Darüber hinaus können zusätzliche Objektassoziationen, Attribute, Konfigurationsmethoden und angepaßte Konstruktoren frei in die neu zu definierenden Klassen eingebunden werden, d.h. der Funktor verwaltet seine Parameter selbst, sorgt für deren Initialisierung und kann geschützt auf sie zugreifen. Das Klassendiagramm in Abb. 7.6 zeigt verschiedene Beispielklassen, die diesen Ansatz verwenden. So ermöglicht FgFunctor, Kamerabilder direkt von einer Framegrabbereinbindung entgegenzu-

nehmen, wofür eine entsprechende Framegrabber-Instanz referenziert wird. StreamFunctor-Objekte verwalten eine Stream-Verbindung, mit deren Hilfe die Daten vom Dateisystem oder einer Netzverbindung gelesen oder in diese geschrieben werden können. Und DisplayFunctor-Instanzen sind für die Visualisierung ikonischer oder textueller Daten in einem Fenster verantwortlich.

Der zweite Weg geht von generischen Funktorklassen, wie GenFunctor aus, die ebenfalls von der Functor-Basisklasse abgeleitet sind. GenFunctor ist — wie die zuvor genannten Klassen — Bestandteil der Klassenhierarchie, verwaltet jedoch kein spezielles Verfahren, sondern eine Referenz auf eine Funktion, die vom Entwickler definiert und dem Funktor bei dessen Erzeugung übergeben wird. Der Entwickler muß damit — um einen neuen Funktor mit einer bestimmten Funktionalität zu erzeugen — keine neue Klasse definieren, sondern nur eine Funktion, die die erforderlichen Operationen zusammenfaßt. Für Funktionen, die keinen besonderen Kontext erfordern, kann dieser Weg die Einbindung vereinfachen und so zu einer Beschleunigung der Programmentwicklung führen. Die Funktion muß eine bestimmte Aufrufschnittstelle besitzen und wird durch die in GenFunctor definierte doFunctor()-Methode gestartet. Die meisten Interpolationsverfahren können auf die gleiche Art in einen Funktor integriert werden, wobei die InterpolateFunctor-Klasse eine modifizierte Aufrufvorbereitung preFunction() und Datenaufbereitung postFunction() durchführt.

Klassen, wie HalconFunctor vereinen schließlich Aspekte der beiden zuvor genannten Ansätze und nehmen so eine gewisse Zwischenstellung ein. Die Aufgabe der HalconFunctor-Klasse ist es, eine Schnittstelle für beliebige Operatoren der Bildverarbeitungsbibliothek HALCON bereitzustellen. Damit stellt sie, wie GenFunctor, eine generische Klasse für eine Gruppe von Funktionen mit einer genau definierten Aufrufschnittstelle dar. Hier müssen die Funktionen allerdings nicht vom Entwickler bereitgestellt werden, sondern stammen aus einer vordefinierten Bibliothek. Aufgrund einer Beschreibung, die für jeden HALCON-Operator zur Laufzeit verfügbar ist, kann allein aus dem Operatornamen und einer Zuordnung zwischen den Operatorparametern und Sequenzen bzw. einer Steuerparameterliste ein gültiger doFunction()-Aufruf generiert werden. Damit kann der Funktor aber auch als ein Spezialfunktor für ein genau abgestecktes Aufgabengebiet aufgefaßt werden, der durch einige Parameter (wie den Operatornamen) lediglich konfiguriert wird.

Attribute und Objektreferenzen

Funktoren werden wie Sequenzen im Datenflußgraphen über ihren Namen name identifiziert. Der Rechnername ist in diesem Fall nicht von Bedeutung, da Funktoren nur lokal bekannt sind. Darüber hinaus wird ein Funktor durch seine Kardinalität $m \times n$ — Anzahl der Eingangs- (valueInNum) und Ausgangsdatenwerte (valueOutNum) — charakterisiert. Da diese Größen für einen konkreten Funktor konstant sind, werden die Referenzen auf die Eingangs- und Ausgangsdatensequenzen in Arrays sequencesIn[valueInNum] und sequencesOut[valueOutNum] verwaltet, die zusammen mit dem Funktor erzeugt werden.

Während für die Ausgangsdaten lediglich die Sequenzreferenzen anzugeben sind, werden die Eingangssequenzen in einem GetSequenceRelation-Array verwaltet. Diese stellen für jeden der m Eingangsdatenwerte den Bezug zu einer konkreten Sequenz $\mathbf{S}$ her. Darüber hinaus enthalten sie den Relativindex index und die (get) -Relation getRelMode : enum {getIdx, getRefTime, getTest, getWait, getRequ, getInterp, getInterpWait, getInterpRequ}.

Functor: Instanzvariablen	
name : string	Funktorname.
valuesInNum : integer valuesOutNum : integer	Kardinalität $m \times n$ des Funktors: Anzahl der Eingangs- und Ausgangs*werte*.
sequencesIn : Sequence[] (GetSequenceRelation[])	Array von Eingangssequenz-Referenzen mit den entsprechenden Zugriffsrelationen.
sequencesOut : Sequence[]	Array für die valuesOutNum Ausgangssequenzen.
callList : FunctorCall[] (FunctorCallArray)	Die Funktoraufrufe begleitende Aufrufinstanzen zur Verwaltung der Eingangs- und Ausgangssequenzwerte, des aktuellen Aktionsstatus und charakteristischer Zeitwerte.
callDepth : integer	Maximalzahl gleichzeitig aktiver Bearbeitungszyklen — entspricht der Größe der als Ringpuffer realisierten Aufrufliste.
activeCycleNum : integer	Anzahl der gleichzeitig aktiven Bearbeitungszyklen.
complCycleNum : integer	Anzahl der abgeschlossenen Zyklen.
currentCycleIdx : integer	Index I des aktuellen Bearbeitungszyklus.
complCycleTime : Time	Zykluszeit des zuletzt abgeschlossen Bearbeitungszyklus.
currentCycleTime : Time	Zykluszeit des aktuellen Bearbeitungszyklus.

Tabelle 7.11: Attribute und Objektreferenzen der Funktor-Basisklasse.

Bei insistenten oder interpolierenden Zugriffsverfahren wird mit syncMode : bool zwischen synchronem und asynchronem Zugriffsmodus unterschieden, und schließlich bestimmt die Variable undefMode : enum wie mit undefinierten (nicht zu verwechseln, mit nicht verfügbaren) Eingangsdaten umzugehen ist. Die folgenden Möglichkeiten stehen zur Verfügung:

undefMode = undefOut : Ein undefiniertes Eingangsdatum führt zwangsläufig zu undefinierten Ausgangsdaten. Konnte beispielsweise im RoboCup der Ball nicht aus dem Bild extrahiert werden, kann ein nachfolgender Funktor auch nicht dessen Position bestimmen. Dieses stellt die Standardeinstellung dar.

undefMode = takeUndef : Der undefinierte Sequenzwert kann von den Operatoren verwendet werden und fließt regulär in die Berechnungen mit ein. Für eine initiale Selbstlokalisation beim RoboCup wird z.B., falls ein Tor im Bild sichtbar ist, diese Information verwendet. Nicht sichtbare Tore dürfen jedoch nicht zum Abbruch der Lokalisation führen, denn zum einen kann für einen Roboter höchstens eines der beiden Tore sichtbar sein, und zum anderen stellt auch der Umstand, kein Tor zu sehen, eine nützliche Information dar.

undefMode = takeLastDef : Statt des aktuellen, undefinierten Datums wird der letzte definierte Wert der Liste verwendet. Wird ein Objekt im Bild verdeckt, kann so z.B. noch eine zeitlang mit der zuletzt ermittelten Objektposition gerechnet werden. Hierbei stellt das Alter dieses Datums eine bedeutende Information dar, die in der doFunction()-Methode unbedingt auszuwerten ist.

Der Zustand eines Funktors läßt sich, wie der von Sequenzen, durch die aktiven und die zuletzt abgeschlossenen Bearbeitungszyklen sowie durch die aktuell laufenden Aktionen beschreiben. Ersteres erfolgt analog zu der für Sequenzen in Abschnitt 7.4.1 beschriebenen Vorgehensweise mit den Attributen complCycleNum, currentCycleIdx, complCycleTime, currentCycleTime

und activeCycleNum. Darüber hinaus werden alle Funktoraufrufe durch eine Aufrufinstanz FunctorCall begleitet. Diese verbindet drei Aufgaben:

1. Weitergabe der Eingangsdaten valuesIn von der Vorbereitungsmethode preFunction() an die doFunction()-Methode und von dort der Ausgangsdaten valuesOut an die abschließende Datenaufbereitung.
2. Verwalten der zwischen den internen Aktionen gemessenen Zeitpunkte: Funktorstart: callStartTime ($t_{\text{call}}, t_{\text{pre}}$), Start der eigentlichen Sensordatenverarbeitung: doStartTime ($t_{\text{do}}, t_{\text{pre.end}}$), Ende der Datenverarbeitung: doEndTime ($t_{\text{post}}, t_{\text{do.end}}$) und Ende des Funktoraufrufs: callEndTime ($t_{\text{leave}}, t_{\text{post.end}}, t_{\text{call.end}}$). Diese Werte stellen die Grundlage für die Auswertung des Zeitverhaltens des Funktors dar.
3. Repräsentation des aktuellen Bearbeitungszustandes actionState : ActionState, bestehend aus der Zykluszeit cycleTime, der aktiven Methode methodId : enum {pre, do, post} und dessen Ausfürungsstatus state : enum {started, running, canceled, finished, error}.

Die letzten callDepth Funktoraufrufe sind in einem Ringarray verfügbar, so daß sie von anderen Komponenten auch nach Abschluß eines Zyklus noch ausgewertet werden können. Neben den wichtigsten, hier untersuchten und in Tabelle 7.11 zusammengefaßten Attributen können Funktoren mit weiteren Attributen, z.B. den mittleren Bearbeitungszeiten oder CPU-Zeiten ausgestattet werden.

Konstruktoren und Instanzmethoden

Functor: Methoden
Functor (name : string, inNum : integer, outNum : integer, depth : integer) Standardkonstruktor zum Erzeugen erzeugt eine Funktorinstanz mit dem Namen name, der Kardinalität inNum × outNum und der maximalen Aufruftiefe depth.
call (cycleTime : TimeAbs) call (cycleTime : TimeAbs, callSequ : Sequence) call (cycleTime : TimeAbs, callAgent : Agent) Funktoraufruf; indiziert durch die Zykluszeit. Die aufrufende Instanz (Sequenz oder Agent) kann als Parameter übergeben werden.
preFunction (callInst : FunctorCall) : ErrorState Interne Methode für die Aufrufvorbereitung; deren wichtigste Aufgabe ist die Bereitstellung der Eingangsdaten für die nachfolgende Datenverarbeitung.
doFunction (callInst : FunctorCall) : ErrorState Interne Datenverarbeitungsmethode; abgeleitete Funktorklassen überschreiben diese. In ihr werden die zu kapselnden Datenverarbeitungsmethoden aufgerufen.
postFunction (callInst : FunctorCall) : ErrorState Interne Methode für die Datenaufbereitung; hier werden die berechneten Ergebnisse in die jeweiligen Ausgangssequenzen eingetragen.

Tabelle 7.12: Elementare Methoden der Functor-Klasse.

Erzeugt wird ein Funktor durch Angabe eines Namens, der Kardinalität und ggf. der Anzahl der gleichzeitig zu verwaltenden Funktoraufrufe. Zusätzlich erwarten die von der Basisklasse abgeleiteten, speziellen Funktoren i.d.R. weitere Objektreferenzen oder Attribute als Parameter, so z.B. der generische Funktor GenericFunctor die zu kapselnde Funktion oder der Framegrabberfunktor FgFunctor eine Framegrabber-Instanz. Auf der anderen Seite besitzen spezielle Funktoren häufig eine feste Kardinalität, so daß diese nicht mehr beim Konstruktoraufruf explizit angegeben werden muß — der Framegrabberfunktor etwa hat die Kardinalität 0×1.

Um den Funktor in einen Datenflußgraphen zu integrieren, müssen Eingangssequenzen mit den entsprechenden Zugriffs- und Steuerungsrelationen sowie die Ausgangssequenzen gesetzt werden. Die Zugriffsmethoden auf die Instanzvariablen werden regulär, wie in Abschnitt 7.1 beschrieben, gebildet. Defaultrelationen können dem Entwickler beim Programmentwurf das Setzen der Zugriffsrelationen abnehmen. Darüber hinaus sollen die Funktoren auch automatisch aus dem in einer Konfigurationsdatei funktional beschriebenen Datenfluß erzeugt und die notwendigen Relationen gesetzt werden können.

Aufgerufen wird ein Funktor ausschließlich über seine call()-Methode, wobei die Zykluszeit den Aufruf indiziert. Erfolgt der Aufruf durch eine Sequenz oder einen Agenten, übergibt diese Instanz zusätzlich eine Referenz auf sich selbst, so daß der Funktor den Initiator des Aufrufs erkennen kann. Auch wenn die bisher beschriebenen Standardfunktoren diese Information nicht verwenden, kann sie in speziellen, für eine bestimmte Anwendung abgeleiteten Funktoren sehr nützlich sein. So erlaubt dies, verschiedene Ursachen des Funktoraufrufs — unabhängig voneinander aktualisierte Eingangsdaten etwa — zu unterscheiden. Im RoboCup wird dieser Mechanismus verwendet, um auf den Robotern verschiedene, asynchron eintreffende Lokalisationsdaten — Odometrie, Bildmerkmale und auf anderen Robotern ermittelte Objektpositionen — mit Hilfe eines Selbstlokalisationsfunktors zu einem gemeinsamen Positionsmodell zu fusionieren.

Die Bearbeitung eines Funktoraufrufs erfolgt in den Phasen Aufrufvorbereitung, Datenverarbeitung und Datenaufbereitung. Realisiert wird dies mit Hilfe der internen Methoden preFunction(), doFunction() und postFunction().

Vor dem Aufruf der preFunction()-Methode wird der Status des Funktors überprüft. Sollte bereits ein Aufruf mit der gleichen Zykluszeit laufen, wird die call()-Methode abgebrochen. Anderenfalls wird eine neue FunctorCall-Instanz erzeugt und in ihr der Aktionsstatus (methodId=pre, state=started) sowie die Funktorstartzeit callStartTime eingetragen. Anschließend wird preFunction() mit der FunctorCall-Instanz als Parameter aufgerufen. Dieser Programmbereich wird mit einem funktorweit gültigen Statussemaphor semaState geschützt.

Die preFunction()-Methode liest von den im sequencesIn[]-Array bezeichneten Sequenzen die Eingangsdaten für die Funktoroperationen und trägt diese in die als Parameter übergebene FunctorCall-Instanz ein. Mit welchen Zugriffsmethoden die Sequenzwerte abgefragt werden, folgt aus den für jeden sequenceIn[]-Eintrag spezifizierten Attributen getRelMode, syncMode und index der Sequenzrelation GetSequenceRelation. Der genaue Ablauf der Aufrufvorbereitung ergibt sich weitestgehend aus den in den Abschnitten 6.4.7 und 6.4.8 beschriebenen Steuerungsmechanismen.

Der Rückgabewert der Methode ErrorState=enum {ok, cancel, undef, error} kennzeichnet den Erfolg der Aufrufvorbereitung und beeinflußt, wie die call()-Methode fortgesetzt wird. Mit cancel wird signalisiert, daß ein Sequenzwert, auf den mit getValueTest() zugegriffen wurde, nicht verfügbar war, woraufhin die call()-Methode abgebrochen wird. Ein Aufrufkonflikt wird

mit der Rückgabe von error angezeigt und führt ebenfalls zum Abbruch der call()-Methode. Als ein weiterer Spezialfall können undefinierte Eingangsdaten bei einer entsprechenden Spezifizierung der Sequenzrelation (GetSequenceRelation : undefMode = undefOut) zur Rückgabe von undef führen. In diesem Fall kann auf den Aufruf der doFunction()-Methode verzichtet und für alle Ausgangssequenzen direkt setUndefValue() aufgerufen werden.

Im Regelfall liefert preFunction() ok zurück, und der call()-Aufruf wird mit doFunction() fortgesetzt. Zuvor wird die aktuelle Uhrzeit ermittelt und in doStartTime der callInst-Instanz eingetragen. Am Beginn der doFunction()-Methode steht das Auslesen der Datenwerte V aus den mit der Aufrufinstanz übergebenen Sequenzwerten S: valuesIn[]. Zusammen mit den Datenwerten, stehen die Zeit- und Typinformationen der Daten zur Verfügung. Stimmen die verfügbaren Datentypen nicht mit den erwarteten überein, führt das zum Abbruch der Methode mit error als Rückgabewert. Sind die Eingangsdaten dagegen zu alt — z.B. weil die aktuellen undefiniert sind — kann doFunction() mit undef abgebrochen werden. Stehen die Eingangsdaten bereit, werden nun die eigentlichen Datenverarbeitungs- oder internen Sensoroperatoren aufgerufen, mit deren Hilfe die Ausgabedaten bestimmt werden. Nach Abschluß dieser Operationen und nachdem die so erzeugten Daten in Sequenzwerte gekapselt und in das valuesOut[]-Array der den Aufruf begleitenden FunctionCall-Instanz eingetragen wurden, wird doFunction() beendet, und die Kontrolle kehrt zur call()-Methode zurück.

Dort wird erneut die Zeit bestimmt und als doEndTime in die callInst-Instanz eingetragen. Die postFunction()-Methode trägt daraufhin in die mit sequencesOut[] referenzierten Ausgangssequenzen die korrespondierenden Sequenzwerte ein. Zum Abschluß wird erneut die Zeit gemessen (callEndTime) und der Funktoraufruf beendet.

Von der Basisklasse Functor abgeleitete Klassen überschreiben i.d.R. nur die doFunction()-Methode. Erfordert der Funktor eine besondere Aufrufvorbereitung oder abschließende Datenaufbereitung, können aber auch die anderen Methoden neu definiert werden.

7.6 Agenten als aktive Komponenten in dynamischen Sensordatensystemen

Auch wenn es in komplexen, dynamischen Sensordatensystemen zahlreiche Aufgabenbereiche gibt, die von agentenbasierten Verfahren profitieren können, würde die Konzeption und (objektorientierte) Modellierung eines umfassenden und allgemeingültigen Agentensystems, das allen in diesem Zusammenhang relevanten Anforderungen gerecht wird, den Rahmen dieser Arbeit sprengen. Daher beschränkt sich das hier vorgestellte Agentenmodell auf die grundlegenden Mechanismen, die für die Steuerung von Datenflußgraphen entsprechend der in den vorangegangenen Kapiteln vorgestellten Konzepte notwendig sind. Die Agenten werden als Sequenzagenten (SequenceAgent) bezeichnet. Von den in [Lüc00] aufgeführten grundlegenden Agenteneigenschaften sind für sie vor allem die folgenden von Bedeutung:

- Autonomie und Aktivität,
- Steuerung durch einen Agentenzyklus,
- Kommunikationsfähigkeit und Konfigurierbarkeit,
- Verschiedene Arbeitszustände: aktiv, schlafend, deaktiviert und terminiert,

- reaktive Verhaltensmuster.

Das Agentenmodell soll dabei so angelegt sein, daß es relativ einfach um zusätzliche Eigenschaften erweitert werden kann, z.B.

- eine explizite Wissensrepräsentation,
- Lern- und Adaptionsfähigkeit sowie
- die Fähigkeit, verschiedene Rollen annehmen zu können.

Diese Eigenschaften sind notwendig, wenn Agenten zusätzliche Aufgaben übernehmen sollen, wie z.B. die Optimierung von Performanz und Ressourcenauslastung. In diesem Fall muß der Agent Ressourcen zuteilen und deren Vergabe planen können, wofür er Informationen über die verfügbaren Mechanismen zur Parallelisierung und Konfiguration des Datenflußgraphen, sowie Wissen über mögliche Auswirkungen dieser Schritte benötigt.

7.6.1 Das Objektmodell

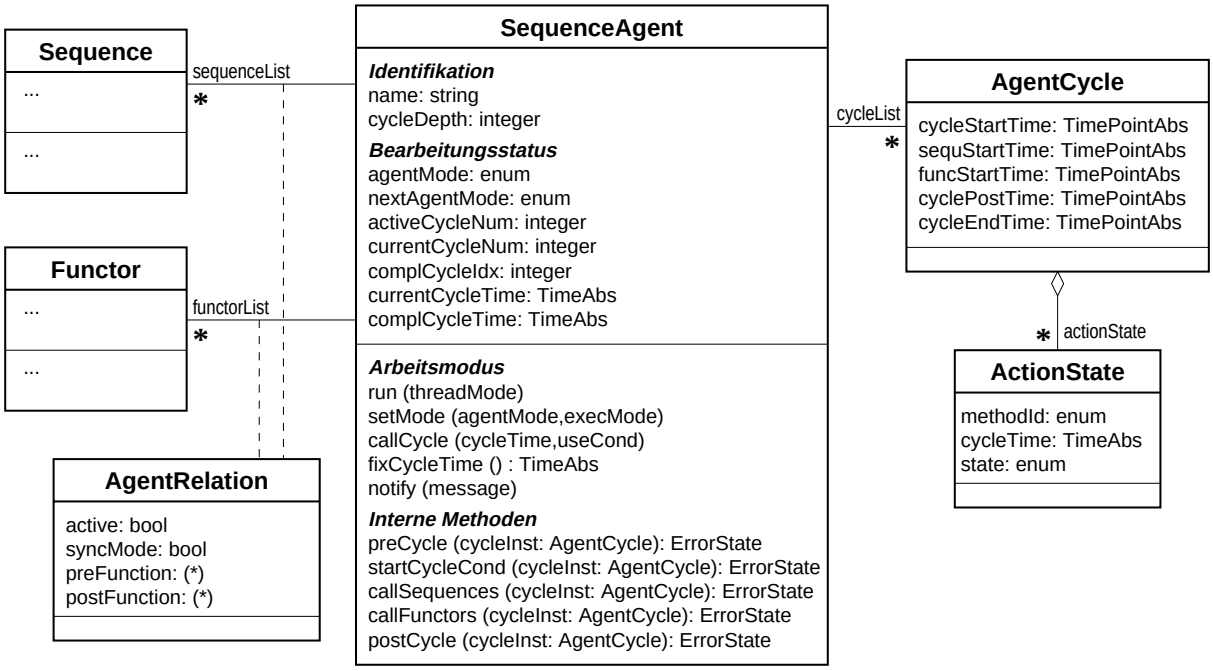


Abbildung 7.7: Klassendiagramm für die Agentenklasse `SequenceAgent`. In ihr werden die wesentlichen Objektassoziationen und Verhaltensmuster von Agenten für die dynamische Sensordatenauswertung definiert.

Der Objektentwurf stützt sich im wesentlichen auf das in Abschnitt 4.4 vorgestellte Agentenmodell. Ein Sequenzagent verwaltet eine Liste mit Sequenzen $\{\mathbf{S}_{\mathbf{dep}}\}$ und eine mit Funktoren $\{\mathbf{F}_{\mathbf{dep}}\}$, die zyklisch zu aktualisieren bzw. zu bearbeiten sind. Davor und danach können zusätzliche Aktionen für die Überwachung und Konfiguration des Systems installiert werden. Im laufenden System können diesen Listen beliebige Sequenzen bzw. Funktoren hinzugefügt und

wieder aus ihnen entfernt werden. Insbesondere mit der Funktorliste steht damit ein mächtiger Mechanismus bereit, dem agentengesteuerten System *beliebige* Funktionalitäten hinzuzufügen und diese frei zu konfigurieren.

Für Agenten werden verschiedene Arbeitsmodi definiert. So können sie inaktiv (noch nicht gestartet bzw. beendet) — oder aktiv sein, letzteres bedeutet im Regelfall, daß sie kontinuierlich arbeiten. Im kontinuierlichen Arbeitsmodus wird ein festes Aufrufschema zyklisch abgearbeitet. Dieser Agentenzyklus läßt sich sowohl durch innere als auch durch äußere Methodenaufrufe unterbrechen oder beenden. Ein schlafender oder deaktivierter Agent kann durch andere Komponenten und externe Ereignisse geweckt werden. Der Einzelschrittmodus erlaubt schließlich, jeden einzelnen Zyklus explizit von außen anzustoßen.

Für die assoziierten Sequenzen und Funktoren lassen sich verschiedene Attribute angeben. Diese legen für jede Agent-Objekt-Relation fest, ob sie aktiv ist und ob sie synchron oder asynchron bearbeitet werden soll. Darüber hinaus erlauben sie die Verwaltung zweier Funktionen — eine die vor dem assoziierten Objekt, und eine, die danach aufzurufen ist. Die Rückgabewerte dieser Funktionen beeinflussen den weiteren Zyklusverlauf.

Die Flexibilität des Agentenansatzes ergibt sich somit vor allem aus den vielfältigen Möglichkeiten, das Arbeitsschema des Agentenzyklus mit verschiedenen Verfahren auszufüllen: Neben dem Hinzufügen neuer Funktoren können die in der SequenceAgent-Klasse definierten Methoden mit eigenen Funktionen oder Objekten assoziiert oder in abgeleiteten Agentenklassen überschrieben und so an die eigenen Bedürfnisse angepaßt werden.

Attribute und Objektreferenzen

Wie Sequenzen und Funktoren besitzt jeder Agent einen Namen name, der bei dessen Erzeugung festgelegt wird, und über den er eindeutig identifiziert werden kann. Die Maximalzahl gleichzeitig aktiver Agentenzyklen wird durch cycleDepth bestimmt, d.h. in einem Sequenzagenten können auch mehrere Zyklen parallel bearbeitet werden. Der aktuelle Zustand wird wie in den Sequenz- und Funktorklassen durch die aktuelle bzw. letzte Zykluszeit t_c, den Index I und die laufenden Aktionen $\{M\}$ gekennzeichnet, wofür die Attribute complCycleNum, currentCycleIdx, complCycleTime, currentCycleTime und activeCycleNum zur Verfügung stehen.

Der Bearbeitungsstand wird — analog zu Funktoren — in einer für jeden Zyklus exklusiven Aufrufinstanz cycleList : AgentCycle[] verwaltet. Um eine Auswertung des Zeitverhaltens der Bearbeitungszyklen zu ermöglichen, werden in die AgentCycle-Instanzen charakteristische Zykluszeiten eingetragen: der Zyklusbeginn — cycleStartTime, Start der Sequenzaktualisierungen — sequStartTime, Beginn der Funktoraufrufe — funcStartTime, Ende der Funktoraufrufe — cyclePostTime und das Zyklusende — cycleEndTime. Der aktuelle Aktionsstatus wird in actionState : ActionState repräsentiert. Aus den Zeiten lassen sich minimale, maximale und mittlere Bearbeitungszeiten ableiten und im Agenten verwalten.

Agenten durchlaufen nicht nur die sich aus ihrer Aktionsabfolgen ergebenden Objekt- und Aktionszustände, sondern sie können auch auf einer grundlegenderen Ebene unterschiedliche Arbeitsmodi annehmen: aktiv, schlafend, deaktiviert und terminiert. Den aktuellen Modus zeigt agentMode an. Ein Moduswechsel kann sofort oder verzögert zum Zyklusende erfolgen. Im verzögerten Fall zeigt nextAgentMode den neuen Zustand an, in den am Ende des laufenden Zyklus gewechselt werden soll.

Die Sequenzen, die zyklisch vom Agenten zu aktualisieren sind, werden in der verketteten Liste sequenceList abgelegt. Eine weitere verkettete Liste functorList verwaltet die in jedem Zyklus aufzurufenden Funktoren. Jedem dieser Listeneinträge können über Instanzen der Assoziationsklasse AgentRelation Attribute zugeordnet werden: syncMode : bool unterscheidet zwischen synchroner und asynchroner Bearbeitung, mit active: bool=false kann eine Assoziation deaktiviert werden, ohne daß das Objekt aus der Liste entfernt werden muß, preFunction verweist auf eine Funktion, die vor der Objektaktualisierung, und postFunction auf eine, die danach aufgerufen wird. Der Rückgabewert dieser Funktionen ErrorState = enum {ok, cancel, abort, error} wird vom Agenten ausgewertet und legt fest, wie der Zyklus fortgesetzt wird. Liefert die Vorbereitungsroutine (∗preFunction) () als Status cancel, wird das dazugehörige Objekt ignoriert und mit dem nächsten in der Liste fortgefahren. Bei abort wird der aktuelle Zyklus abgebrochen und error führt zu einer Ausnahmebehandlung durch den Agenten.

Methoden

Die in der SequenceAgent-Klasse definierten Methoden haben in erster Linie die Aufgabe, ein Bearbeitungsschema für die Agentenzyklen zu definieren und dadurch die Arbeitsweise der Sequenzagenten festzulegen. Es besteht aus den folgenden Schritten:

Zyklusvorbereitung — preCycle()**:** Abfrage von Ausführungs- und Abbruchbedingungen mit der Methode startCycleCond () : ErrorState — diese ermöglicht es dem Entwickler, Vorbedingungen für den Start eines neuen Zyklus und für den Abbruch der Agentzyklen zu definieren.

Sequenzaktualisierung — callSequences()**:** Entsprechend ihrer Reihenfolge in sequenceList werden die Sequenzen, für die das active-Flag der AgentRelation-Instanz gesetzt ist, aufgerufen, d.h. auf sie wird mit der getValueRequ()-Methode zugegriffen. Sind die entsprechenden Funktionen in der AgentRelation-Instanz gesetzt, wird vor diesem Zugriff (∗preFunction)() und nach dem Zugriff (∗postFunction)() aufgerufen.

Funktoraufrufe — callFunctors()**:** Entsprechend ihrer Reihenfolge in functorList werden die Funktoren, für die das active-Flag der AgentRelation-Instanz gesetzt ist, mit deren call()-Methode aufgerufen. Sind die entsprechenden Funktionen in der AgentRelation-Instanz gesetzt, wird vor diesem Zugriff (∗preFunction)() und nach dem Zugriff (∗postFunction)() aufgerufen.

Zyklusauswertung — postCycle()**:** Diese Methode erlaubt eine Auswertung der Ergebnisse und des Zeitverhaltens des vorangegangenen Zyklus. Sie ist im bisherigen System leer, stellt jedoch die Schnittstelle für die Einbindung von Analyse- und Rekonfigurationsverfahren dar.

Zwischen den einzelnen Schritten wird die Zeit gemessen und in die Aufrufinstanz des Zyklus eingetragen. Diese Zeiten können für Laufzeitanalysen ausgegeben oder mitprotokolliert werden.

Weitere Methoden ermöglichen es, den Agenten zu konfigurieren (z.B. durch Setzen von Sequenzen und Funktoren sowie der dazugehörigen Assoziationen) oder den Arbeitsmodus des Agenten zu ändern, d.h. dessen Arbeit anzuhalten, zu unterbrechen, zu beenden oder ihn (wieder) zu aktivieren. Mit setMode (agentMode, execMode) kann direkt der neue Modus angegeben

werden: agentMode : enum {active, sleeping, deactive, terminated}. Der Parameter execMode legt fest, ob der Moduswechsel sofort, nach Ende der laufenden internen Aktion oder am Ende des Zyklus erfolgen soll.

Starten lassen sich die Agenten auch mit der run (threadMode)-Methode. Der Parameter threadMode : bool legt dabei fest, ob der Agent in einem eigenen Thread gestartet wird, was mehrere aktive Komponenten in einem Programm erlaubt, oder ob das Programm die Kontrolle exklusiv an den Agenten übergibt. Im zweiten Fall ist zu beachten, daß, wenn der Agent terminiert, das Programm an der Position nach dem Agentenaufruf fortgesetzt wird.

Statt den Agenten mit run() im kontinuierlichen Modus zu betreiben, kann mit callCycle (cycleTime, useCond) auch jeder Zyklus einzeln von außen angestoßen und so die Kontrolle von einer externen Instanz ausgeübt werden. Der Parameter useCond : bool legt fest, ob die interne Aufrufbedingung abzutesten ist, oder ob der Zyklus *un-bedingt* ausgeführt werden soll. In diesem Fall wird die Zykluszeit von außen vorgegeben und als Parameter übergeben. Arbeitet der Agent im kontinuierlichen Modus, bestimmt die fixCycleTime()-Methode die Zykluszeit.

Die Methode notify() schließlich ermöglicht es anderen Objekten, den Agenten direkt über bestimmte Ereignisse zu informieren und z.B. interne Statusvariablen, die beim Aufruftest abgefragt werden, zu modifizieren.

Kapitel 8

Realisierung, Zusammenfassung und Ausblick

8.1 Umfang der Implementierung und Bewertung

Ausgangspunkt dieser Arbeit war der Wunsch nach einem Softwaresystem, das den Entwurf, die Implementierung und die Konfiguration von komplexen dynamischen Sensordatenapplikationen, insbesondere im Bereich der Bildfolgenanalyse, unterstützt und durch geeignete Modelle deutlich erleichtert. Der RoboCup als neues attraktives Forschungsgebiet und sich etablierendes Standardproblem für die KI, Robotik und Sensordatenverarbeitung bildete dabei den Anwendungsrahmen, wofür ein eigenes Forschungsprojekt in der Forschungsgruppe Bildverstehen (FG BV) der Technischen Universität, München ins Leben gerufen wurde. Die mobilen Roboter stellen in diesem Szenario zusammen mit der Umwelt, in der sie agieren, ein komplexes, dynamisches, verteiltes und inhomogenes Multisensorsystem dar. Dieses Szenario besitzt grundlegende Eigenschaften und Anforderungen realer Anwendungen.

Die Praxistauglichkeit wurde damit von Anfang an zu einem der wichtigsten Bewertungskriterien der vorgestellten Konzepte. Daher stellt die Implementierung der entwickelten Modelle und deren Einsatz in einem realen System eine zentrale Aufgabe des Promotionsprojektes dar. Sie bildete die Voraussetzung für eine Überprüfung der Anwendbarkeit der Konzepte. Anhand welcher Kriterien diese Validierung zu erfolgen hat, folgt direkt aus dem Profil und den Anforderungen der Anwendungen: Elementare Voraussetzungen für die Handhabbarkeit und effiziente Umsetzung der Applikationen sind die leichte Skalierbarkeit einzelner Programmkomponenten, flexible Konfigurationsmöglichkeiten, ein geringer Overhead und — insbesondere für die Sensordatenauswertung — Mechanismen, die eine Auswahl und Parametrisierung der einzusetzenden Verfahren — auch im laufenden System —- unterstützen. Dies wiederum erfordert Hilfsmittel für die Visualisierung der Sensordaten, der extrahierten Merkmale sowie des Ressourcenverbrauchs der Operatoren, wodurch ein Vergleich der verschiedenen Verfahren unter realen Anwendungsbedingungen möglich wird.

Anhand des Systementwurfs im RoboCup soll in diesem Abschnitt illustriert werden, wie die entwickelten Modelle diesen Forderungen gerecht werden. Dabei wird auf die verschiedenen, in diesem Rahmen realisierten Steuerungsmechanismen eingegangen. Verzichtet wurde an dieser Stelle auf detaillierte quantitative Vergleiche der unterschiedlichen Mechanismen. Für den Einsatz im RoboCup sollte es genügen, mit Laufzeitmessungen zu belegen, daß der

Overhead durch die bereitgestellten Modelle vernachlässigbar gering ist. Dieser liegt bei unter 80 μs für einen insistenten Sequenzzugriff einschließlich aller Aktionen für die Aktualisierung der Sequenz.[1] Für den geringen Overhead der Modelle spricht auch die hohe Datenrate von $10-15$ Farbbildern (bei einer Auflösung von 384×172 Pixel), die pro Sekunde verarbeitet und ausgewertet werden können. Weiterführende quantitative Untersuchungen hätten den Umfang dieser Arbeit gesprengt und sollen Gegenstand zukünftiger Forschungsarbeiten sein.

Die zentralen Komponenten des vorgestellten Systems wurden in der objektorientierten Programmiersprache C++ prototypisch implementiert und im RoboCup-Projekt eingesetzt. Zu den entwickelten Komponenten zählen in erster Linie die Klassen für die Repräsentation von Zeit, Datenfolgen und Sequenzwerten, Funktionsobjekten und Agenten. Für die Umsetzung der in Kapitel 7 vorgestellten Objektmodelle waren in C++ zahlreiche zusätzliche Klassen erforderlich, da Implementierungsaspekte, wie die Speicherplatzbereitstellung und -freigabe sowie Typbindungsmechanismen im konzeptionellen Entwurf unberücksichtigt bleiben, in C++ jedoch die Anwendung spezieller Entwurfsmuster fordern. In unmittelbarem Zusammenhang mit diesen Modellen für die Repräsentation dynamischer Sensordatenfolgen wurden zahlreiche weitere Hilfsklassen entwickelt, so z.B. Klassen, die die Protokollierung der Datensequenzen in Dateien oder den transparenten Zugriff auf sie über Prozeß- und Rechnergrenzen hinweg erlauben.

Des weiteren wurde im Rahmen dieser Arbeit ein aus mehreren Bibliotheken bestehendes Basissystem mit grundlegenden Datentypen und Objekten für die Sensordatenverarbeitung in autonomen Robotern entwickelt. Zu den hierin bereitgestellten Funktionalitäten zählen beispielsweise die Einbindungen von Framegrabbergeräten und Kameras, die Roboterrepräsentationen sowie Datenmodelle für die verschiedenen Sensordatentypen und Objektmerkmale. Grundlegende ikonische Sensordatentypen wurden dabei vom HALCON-Bildverarbeitungssystem übernommen.

Basierend auf diesem Basissystem und den Klassen für die Verarbeitung von Sensordatenfolgen wurden schließlich die verschiedenen, für den RoboCup sowie andere Multiagenten-Multiroboter-Systeme erforderlichen Programmkomponenten und Subsysteme entwickelt und implementiert. Im einzelnen handelt es sich dabei um Softwareklassen für die folgenden Aufgabengebiete:

- Erweiterung der Roboteransteuerung um Basisfähigkeiten, die das Navigieren der Roboter auf dem Spielfeld mit und ohne Ball erlauben.
- Zahlreiche konkrete Verfahren für die Bildfolgenauswertung, mit denen beispielsweise die Farbbilder klassifiziert, die relevanten Objektmerkmale aus den Bildern extrahiert und die entsprechenden Objekte in Relation zum Roboter und zum Spielfeld lokalisiert werden.
- Verfahren die sowohl eine initiale als auch die schritthaltende, bild- und odometriebasierte Selbstlokalisation der Roboter erlauben.
- Verfahren sowie Daten- und Kommunikationsstrukturen für die Multisensorfusion, d.h. für das Zusammenführen der verschiedenen lokal ermittelten Sensordaten von Kamera, Roboterodometrie und im begrenzten Umfang auch Ultraschallsensoren, sowie der von den anderen Robotern ermittelten und übertragenen Objektmerkmale.

[1] Der Overhead wurde auf einem im RoboCup verwendeten Pentium MMX 200 MHz System gemessen.

- Die lokale Planung von Aktionen und Bewegungen (einschließlich einer Hindernisvermeidung) für die einzelnen Roboter auf der Grundlage der lokalen Szeneninformationen.
- Die Koordination verschiedener Roboteraktionen sowie die Kooperation von mehreren Robotern und die Planung gemeinsamer Aktionen, jeweils unter Berücksichtigung der aktuellen Strategie und Taktik.
- Mechanismen für eine flexible und frei skalierbare Visualisierung der parallelen Roboteraktionen und den relevanten Objektmerkmalen.
- Eine Benutzerschnittstelle, die es dem Entwickler — auf der Suche nach geeigneten Verfahren und Parametern oder bei der Fehlersuche — erlaubt, das laufende System frei zu konfigurieren sowie ausgewählte Informationen auszugeben.

Dank der in dieser Arbeit beschriebenen Konzepte und deren objektorientierten Umsetzung sind die verschiedenen Programmteile gut separierbar und konnten relativ unabhängig voneinander entwickelt werden. In einer vergleichsweise kurzen Entwicklungszeit wurden diese so zu einem funktionstüchtigen Gesamtsystem zusammengefügt, was die Praxistauglichkeit der vorgeschlagenen Konzepte unterstreicht. Die Funktionsfähigkeit des Systems wurde durch die guten Ergebnisse, die das Münchner RoboCup-Team bei den Wettbewerben 1999 in Stockholm und Stuttgart erzielen konnte, unter Beweis gestellt. Die meisten der genannten Aufgabengebiete sind weiterhin Gegenstand aktueller Forschungen und werden unter dem Aspekt der Optimierung des Gesamtsystems weiterentwickelt.

8.2 Überblick über das im RoboCup entwickelte Sensordatenmodul

In dem Gesamtsystem nimmt das Modul für die Aufbereitung der Sensordaten eine zentrale Rolle ein. Diesen Aufgabenbereich hier vollständig behandeln zu wollen, würde den Rahmen dieser Arbeit sprengen. Weiterführende Informationen finden sich in [BHKS99, BKHS99, KLZ+99]. An dieser Stelle sollen lediglich einige Realisierungsaspekte, die aus Sicht der vorgestellten Konzepte interessant sind, beleuchtet werden.

Den anderen Programmkomponenten gegenüber präsentiert sich die Sensordatenverarbeitung als ein logischer Sensor, dessen Aufgabe es ist, u.a. die folgenden, für die Planung und die Roboteraktionen relevanten Szenenmerkmale in Form von Datensequenzen bereitzustellen:

Ballposition: Entfernung, Richtung und Bewegung des Balles relativ zum Roboter ($\mathbf{S_{Ball:PosRel}}$ = BallPosRelSequ) und in absoluten Spielfeldkoordinaten ($\mathbf{S_{Ball:PosAbs}}$ = BallPosAbsSequ);

Positionen anderer Roboter: Entfernung und Richtung der im Bild sichtbaren Roboter relativ zur eigenen Position ($\mathbf{S_{Robots:PosRel[]}}$ = RobotsPosRelSequ), ggf. mit der Kennzeichnung, ob der Roboter als eigener oder fremder erkannt wurde;

Sichtbare Torbereiche: Entfernung und Richtung des größten sichtbaren Torbereichs relativ zur eigenen Roboterposition ($\mathbf{S_{OwnGoal:PosRel}}$ = OwnGoalPosRelSequ und $\mathbf{S_{OppGoal:PosRel}}$ = OppGoalPosRelSequ), wobei die Roboter konstruktionsbedingt immer nur maximal eines der beiden Tore sehen können;

Freie Spielfeldbereiche: *Free Motion Space, FMS*, freie Segmente auf dem Spielfeld relativ zum Roboter ($\mathbf{S_{FMS:PosRel[]}}$ = FmsPosRelSequ);

Eigene Position: geschätzte Position und Orientierung des eigenen Roboters auf dem Spielfeld ($\mathbf{S_{Robot:PosAbs}}$ = RobotPosAbsSequ).

Diese Datensequenzen werden auf jedem Roboter lokal bereitgestellt. Ihre aktuellen Werte können jederzeit systemweit abgefragt werden. Mit den in dieser Arbeit beschriebenen Mechanismen werden die Daten auf alle anderen Roboter und die externen Hilfsrechner für die Visualisierung (*Monitor*) und übergeordnete Planung (*Coach*) gespiegelt, so daß auch dort transparent auf sie zugegriffen werden kann. Damit ist jeder Roboter stets über die Positionen seiner Mitspieler informiert. Die auf den verschiedenen Robotern ermittelten Ballpositionen können in die lokale Ball- und Selbstlokalisation einfließen, so daß z.B. auch die Roboter, die den Ball nicht sehen, dessen Position abschätzen können. Das externe *Coach*-Programm hat die Aufgabe, ein konsistentes Weltmodell zu erstellen, den Spielern situationsabhängig Rollen zuzuweisen und übergeordnete Planungsaufgaben vorzugeben. Das *Monitor*-Programm dient der Visualisierung der extrahierten Sensordaten und Roboterpositionen, was eine essentielle Aufgabe bei die Entwicklung und dem Einsatz einer derart komplexen Applikation darstellt.

Als Eingangsdaten fließen in das Sensordatenmodul die von der Kamera bereitgestellte Videobildfolge, die Odometriedaten des Roboters und die extrahierten Szenenmerkmale der anderen Roboter. Dabei werden zum einen die unterschiedlichen Meßzeiten der verschiedenen Sensoren und zum anderen der Zeitfehler zwischen den einzelnen Robotersystemen ermittelt und modelliert.

Intern wird das Modul durch einen relativ grobgranularen Datenflußgraphen repräsentiert. Die in ihm enthaltenen internen Datensequenzen und Funktoren orientieren sich an abstrakt formulierbaren Teilaufgaben, mit denen die oben genannten Szenenmerkmale bestimmt werden können. Im einzelnen werden Funktoren für die folgenden Aufgaben bereitgestellt:

Framegrabbereinbindung ($\mathbf{F_{FgRobo}}$ = FgRoboFunctor): Mit Hilfe dieses Funktors werden die Videobilder als Bildsequenz bereitgestellt. Dabei stellt der im Funktor gekapselte Operator ein vorklassifiziertes Farbbild ($\mathbf{S_{Color:Reg}}$ = ColorRegsSequ) und ein Schwarzweißbild ($\mathbf{S_{Camera:Img}}$ = CameraImgSequ) zur Verfügung:

$$\{\mathbf{S_{Color:Reg}}, \mathbf{S_{Camera:Img}}\} = \mathbf{F_{FgRobo}}\,(\,)$$

Spielfeldsegmentation ($\mathbf{F_{FieldSegm}}$ = FieldSegmFunctor): In dem klassifizierten Farbbild werden zusammenhängende Farbregionen, die zum Spielfeld gehören, gesucht und entsprechend des Spielfeldmodells klassifiziert. Dabei wird zwischen dem grünen Spielfeld ($\mathbf{S_{Field:Reg}}$ = FieldRegSequ), den weißen Spielfeldlinien ($\mathbf{S_{Lines:Reg}}$ = LinesRegSequ), der weißen Wand ($\mathbf{S_{Wall:Reg}}$ = WallRegSequ) und den grünen Markierungen der Spielfeldecken ($\mathbf{S_{Corner:Reg}}$ = CornerRegSequ) als separate Ausgangssequenzen des Funktors unterschieden:

$$\{\mathbf{S_{Field:Reg}}, \mathbf{S_{Lines:Reg}}, \mathbf{S_{Wall:Reg}}, \mathbf{S_{Corner:Reg}}\} = \mathbf{F_{FieldSegm}}\,(\,\mathbf{S_{Color:Reg}}\,)$$

Ballsegmentaton ($\mathbf{F_{BallSegm}}$ = BallSegmFunctor): Im klassifizierten Farbbild wird nach einer roten Region, die dem Ball entsprechen kann, ($\mathbf{S_{Ball:Reg}}$ = BallRegSequ) gesucht. Unter Berücksichtigung entsprechender Konsistenzchecks wird die Position des Balles relativ zum Roboter ermittelt:

$$\{\mathbf{S_{Ball:Reg}}, \mathbf{S_{Ball:PosRel}}\} = \mathbf{F_{BallSegm}}\,(\,\mathbf{S_{Color:Reg}}\,)$$

Bestimmen der Absolutposition des Balles ($\mathbf{F_{BallLocal}}$ = BallLocalFunctor): Ausgehend von der relativen Ballposition, der alten absoluten Ballposition, den von den anderen Robotern ermittelten absoluten Ballpositionen und der eigenen Roboterposition wird die absolute Ballposition in Spielfeldkoordinaten bestimmt:

$$\mathbf{S_{Ball:PosAbs}} = \mathbf{F_{BallLocal}}\left(\mathbf{S_{Ball:PosRel}}, \mathbf{S^{(-1)}_{Ball:PosAbs}}, \mathbf{S^{(0)}_{Ball[rob]:PosAbs}}, \mathbf{S_{Robot:PosAbs}}\right)$$

Robotersegmentation ($\mathbf{F_{RobotsSegm}}$ = RobotsSegmFunctor): Im klassifizierten Farbbild wird nach anderen Robotern gesucht. Für diese wird die Position relativ zum Roboter ermittelt. Konnten die Roboter anhand ihrer Farbmarkierung der eigenen oder der gegnerischen Mannschaft zugeordnet werden, wird dies als Attribut mit abgespeichert:

$$\{\mathbf{S_{Robots:Reg}}, \mathbf{S_{Robots:PosRel[]}}\} = \mathbf{F_{RobotsSegm}}\left(\mathbf{S_{Color:Reg}}\right)$$

Toresegmentation ($\mathbf{F_{GoalsSegm}}$ = GoalsSegmFunctor): Im vorklassifizierten Farbbild wird nach den Toren gesucht und deren Position relativ zum Roboter bestimmt, wobei insbesondere der freie Teil des Tores interessiert. Dabei wird zwischen dem eigenen und dem gegnerischen Tor unterschieden:

$$\{\mathbf{S_{OwnGoal:Reg}}, \mathbf{S_{OwnGoal:PosRel}}, \mathbf{S_{OppGoal:Reg}}, \mathbf{S_{OppGoal:PosRel}}\} = \mathbf{F_{RobotsSegm}}\left(\mathbf{S_{Color:Reg}}\right)$$

Bestimmung des Free Motion Space ($\mathbf{F_{FmsSegm}}$ = FmsSegmFunctor): Aus den Spielfeld- und Roboterregionen und unter Berücksichtigung der Ballregion wird der freie Raum vor dem Roboter ermittelt:

$$\mathbf{S_{FMS:PosRel[]}} = \mathbf{F_{FmsSegm}}\left(\mathbf{S_{Color:Reg}}\right)$$

Selbstlokalisation ($\mathbf{F_{SelfLocal}}$ = SelfLocalFunctor): Der Selbstlokalisationsfunktor kapselt zwei verschiedene Lokalisationsverfahren, ein relatives und ein absolutes. Das relative Verfahren versucht anhand der Roboterodometrie bzw. der Farbübergänge im vorklassifizierten Farbbild die Roboterposition zu *korrigieren*. Dieses Verfahren ist relativ schnell, benötigt jedoch eine halbwegs genaue Schätzung der Position, wofür i.d.R. die im vergangenen Zyklus ermittelte Position ausreicht. War diese Schätzung nicht gut genug, kann das anhand eines Fehlermaßes und durch einen Konsistenzcheck, in den die extrahierten Torpositionen einfließen, erkannt werden. In diesem Fall wird ein absolutes, allerdings auch deutlich langsameres Lokalisationsverfahren verwendet. Dieses kann initial, d.h. ohne Vorwissen die eigene Position auf dem Spielfeld bestimmen. Dies geschieht auf der Basis von Grauwertkanten sowie von extrahierten Spielfeldmerkmalen, wie Linien und Toren:

$$\mathbf{S_{Robot:PosAbs}} = \mathbf{F_{SelfLocal}}\left(\mathbf{S_{Color:Reg}}, \mathbf{S_{Odometry:Pos}}, \mathbf{S_{Camera:Img}}, \mathbf{S_{OwnGoal:PosRel}}, \mathbf{S_{OppGoal:PosRel}}, \mathbf{S_{Lines:Reg}}\right)$$

Diese Aufzählung zeigt bereits die zwischen den Funktoren bestehenden Abhängigkeiten. Diese werden durch die Art der Programmbeschreibung, die auf der funktionalen Datenflußbeschreibung basiert, automatisch aufgelöst. Konflikte durch Abhängigkeiten, die sich nicht automatish auflösen lassen, werden vom System angezeigt.

Die Ablaufsteuerung in diesem Sensor ist wie auch dessen interner Aufbau vollkommen losgelöst von den anderen Programmkomponenten und damit unabhängig vom Rest des Systems konfigurierbar. In der realisierten RoboCup-Applikation werden verschiedene der in Kapitel 6

vorgestellten Konzepte angewendet. Als Basissteuerungsmechanismus des Sensordatenmoduls wird die *anfragegetriebene Steuerung* verwendet. Die Anfragen an die interessierenden Ausgangsdatensequenzen werden kontinuierlich von einem Agenten generiert, was zu einer optimalen Ressourcenauslastung führt.

Daneben werden verschiedene Aktionen *datengetrieben* ausgelöst, so z.B. die Selbstlokalisation, die sowohl durch neue Roboterodometridaten als auch durch Bilddaten oder durch von anderen Robotern übermittelte Positionsdaten getriggert wird.

Pipelining oder Aufgabenparallelität innerhalb eines Sensordatenmoduls spielte bei der Implementierung bisher noch keine Rolle, da es sich bei den im RoboCup verwendeten Rechnern um Einprozessormaschinen handelt. Allerdings wird die asynchrone Arbeit verschiedener Programmkomponenten durch entsprechende Parallelisierungsmittel unterstützt. So können das Sensordatenmodul und die Planungskomponente mit unterschiedlichen Zyklusraten arbeiten. Diese unterscheiden sich von der Videorate und sind im allgemeinen nicht konstant. Die Robotersteuerung besitzt wiederum ihre eigenen Bearbeitungszyklen, die durch die Roboterhardware vorgegeben werden. Ein weiterer Parallelisierungsaspekt ergibt sich durch die gleichzeitig aktiven und miteinander agierenden Roboter- und Hilfsrechner. Neben dem asynchronen Datenaustausch zwischen den einzelnen Prozessen sind hierbei zusätzlich die verschiedenen Zeitbasen der Rechner aneinander anzugleichen.

In das System wurden verschiedene Mechanismen für die Interaktion zwischen dem Programm und dem Entwickler bzw. Anwender integriert. So lassen sich über eine Textkonsole zahlreiche Statusinformationen, extrahierte oder berechnete Bild- bzw. Objektmerkmale und Programmaktivitäten ausgeben. In Grafikfenstern können ikonische Daten dargestellt werden, beispielsweise um die Klassifikations- und Segmentationsergebnisse visuell überprüfen und Parameter einstellen oder ggf. nachjustieren zu können. Diese Ausgabefunktionen lassen sich jederzeit, also auch in der laufenden Echtzeitanwendung, flexibel aktivieren oder deaktivieren, wodurch die Grenzen zwischen Entwicklungs- und Laufzeitversion verschwimmen. Dabei sind die Laufzeitverluste bei deaktivierten Ausgaben, wie die erreichten Bearbeitungsraten zeigen, minimal.

Die vorgestellte Sensordatenbibliothek stellt Mechanismen für die Rekonfiguration des Datenflußgraphen bereit. In der bisherigen Applikation nutzen Agenten diese Möglichkeit noch nicht aus. Allerdings ist ein Austausch der Funktoren und ein Neusetzen der Daten- und Steuerflußbeziehungen auch über die Benutzerschnittstelle möglich, so daß der Entwickler die Rekonfiguration durchführen kann, etwa um verschiedene Verfahren unter realen Bedingungen zu testen und miteinander zu vergleichen. Die Benutzerschnittstelle erlaubt auch, Funktoren, die erst nach dem Start der Applikation entwickelt und kompiliert wurden, dynamisch nachzuladen und in den Datenflußgraphen zu integrieren. Die bisher integrierten rein textuellen Mechanismus sind dafür allerdings noch relativ umständlich, Mittel und Wege um dies zu vereinfachen, wurden in [Hof99] vorgestellt.

8.3 Zusammenfassung

In der vorliegenden Arbeit wurden neuartige Konzepte für die Modellierung von Bildfolgen und dynamischen Datensequenzen sowie für die Einbindung von kontinuierlich aufzurufenden Da-

tenverarbeitungsverfahren entwickelt und implementiert. Dabei wurde ein Anwendungsrahmen geschaffen, der die Entwicklung von Programmkomponenten für die dynamische Sensordatenverarbeitung unterstützt, indem Mechanismen und eine objektorientierte Klassenstruktur bereitgestellt werden, die die direkte Übertragung funktionaler Datenflußbeschreibungen in eine Implementierung ermöglichen. Das Konzept des logischen Sensors spielt dabei eine besondere Rolle, da in diesem Ansatz dynamische, durch beliebige Operatoren bereitgestellte Datenfolgen in den Applikationen in der gleichen Weise zugreifbar sind wie Sensordaten, die von einem physikalischen Sensor geliefert werden. Mit diesem Konzept lassen sich physikalische Sensoren, spezielle, in Hardware realisierte Datenverarbeitungseinheiten, Bildverarbeitungskarten, Operatoren sowie komplette Sensordatenmodule nach einem einheitlich Prinzip — als logische Sensoren — in eine Applikation einbinden und über eine einheitliche, anwendungsunabhängige Schnittstelle steuern und zugreifen. Der interne Aufbau eines solchen logischen Sensors ist dabei aus Sicht der Datenkonsumenten vollkommen unerheblich.

Die Modellierung von Zeit und quantitativen zeitlichen Ausdrücken stellt ein zentrales Konzept dieses Ansatzes dar. In Kapitel 3 wurden dafür die Grundlagen gelegt und eine konsistente Notation für die Darstellung von Zeitpunkten und Intervallen vorgestellt. Den verschiedenen semantischen Ausprägungen zeitlicher Ausdrücke wird in dem Modell durch die Unterscheidung von Absolut- und Relativzeiten gerecht. Darüber hinaus wird durch die feste Einbeziehung der physikalischen Zeiteinheit Sekunde in das Zeitmodell der Bezug zur Realität gewahrt, was bei der Modellierung realer physikalischer Sensoren unverzichtbar ist. Für die Interpretation von Zeitausdrücken in einem realen Computersystem stellt die explizite Unterscheidung zwischen der kontinuierlich ablaufenden Realzeit, der Rechenzeit von Operatoren im Computer und der Datenmeßzeit der von physikalischen Sensoren gelieferten Szenendaten eine wichtige Voraussetzung für die Modellierung dynamischer Prozesse dar. Zentrale Begriffe, die in diesem Zusammenhang modelliert werden, sind das Alter der Daten, die Verzögerung zwischen einer Datenanfrage und deren Verfügbarkeit und die Gültigkeitesdauer von Daten, ferner die Zyklusdauer und die Operatorlaufzeit.

Erstmals werden Bildfolgen in einem Bildverarbeitungs- bzw. Bildanalysesystem nicht nur als *variables Bild* oder lose assoziierte Gruppe von Bildinstanzen modelliert, sondern als *ein langlebiges Sequenzobjekt*, welches die nacheinander von einer Kamera aufgenommenen (oder durch ein Bildverarbeitungsoperator gelieferten) Bilder zusammenfaßt und diese zusammen mit dem (relativ) konstanten Verarbeitungskontext der Daten verwaltet. Neu ist an dem vorgestellten Ansatz ferner, daß die dynamischen Aspekte von Sensordaten durch ein *allgemeines* Modell dargestellt werden und nicht auf Bildfolgen und damit auf einen bestimmten Datentyp beschränkt sind. Damit wird ein einheitliches Interface für den Umgang mit den verschiedensten dynamischen Daten, unabhängig vom Datentyp und der die Daten erzeugenden Quelle bereitgestellt. Zu den wichtigsten allgemeinen Merkmalen von Datenfolgen, die durch die Sequenzklasse modelliert werden, zählen:

- der gemeinsame Verarbeitungskontext gleichartiger Daten, der insbesondere durch die Funktionsobjekte geprägt wird, die die Daten erzeugen (als logischer Sensor), weiterverarbeiten (bei der Einbindung der Datenfolge in einen übergeordneten logischen Sensor) oder auswerten (bei der Triggerung bestimmter Aktionen durch die Datenfolge);
- die zeitliche Einordnung der Daten in die realen Szenenabläufe sowie in die Datenverarbeitungsprozesse;

- zeitabhängige Zugriffsverfahren auf die Daten, mit denen das Verhalten des logischen Sensors, der die Daten bereitstellt, gesteuert werden kann;
- die Verwaltung des Datenverlaufs, einschließlich von Mechanismen und Regeln für den Zugriff auf alte Daten und die Objektfreigabe;
- die Verwaltung von Interpolationsverfahren und der einheitliche Zugriff auf interpolierte oder predizierte Datenwerte;
- die Protokollierung der Daten in eine Datei und deren spätere Rekonstruktion;
- der transparente Datenaustausch in einem Rechnernetz;
- sowie die Dateninitialisierung.

Bei der Repräsentation von Operatoren für die Datenverarbeitung findet man in der Literatur bereits Arbeiten, die zwischen allgemeinen sowie lang- und kurzlebigen Funktionsaspekten unterscheiden. Neu ist hier die konsequente explizite Modellierung der Datenflußbeziehungen, in die die Operatoren eingebettet sind, die Repräsentation von dynamischen Aspekten zur Beschreibung des Laufzeitverhaltens von Funktionsaufrufen sowie die einheitliche, sich aus der Art der Einbindung in einen Datenflußgraphen ergebende Aufrufschnittstelle für die unterschiedlichsten Verfahren. Mit der *Funktorklasse* wird eine eigene abstrakte Operatorebene bereitgestellt. In ihr wird von der konkreten Funktionalität der einzubindenden Verfahren vollkommen abstrahiert. Sie legt damit die Grundlage für die Definition allgemeiner Methoden und Mechanismen, die

- die Auswertung des Zeitverhaltens von Operatoren und Datenverarbeitungsverfahren und deren Vergleich erlauben;
- darauf aufbauend den einfachen Austausch der einzusetzenden Verfahren — auch im laufenden System — ermöglichen;
- die Etablierung allgemeiner Programmsteuerungs- und Parallelisierungsstrategien unterstützen.

Durch die Kopplung der im Rahmen der Sensordatenauswertung durchzuführenden Aktionen an einen bestimmten Zeitwert lassen sie sich trotz ihrer sequentiellen bzw. asynchronen Ausführung einander und einem konkreten (synchronen) Arbeitszyklus zuordnen, wodurch ein eindeutiger Bezug zwischen einzelnen Datenwerten und Funktionsaufrufen hergestellt werden kann. Dadurch werden Mehrfachberechnungen in einem Verarbeitungszyklus vermieden, und Datenabhängigkeiten können eindeutig aufgelöst werden.

Mit Hilfe der Funktoren und Datensequenzen wird neben der Datenflußebene eine eigene Ebene für die Spezifikation der Programmsteuerung und des Parallelisierungsgrades bereitgestellt. Ausgehend von der funktionalen, implizit parallelen, Beschreibung der internen Datenflußbeziehungen eines logischen Sensors kann die Aufteilung der Programmsteuerung in parallele Kontrollflüsse automatisch erfolgen und — eine planende Komponente vorausgesetzt — entsprechend der vorhandenen Ressourcen im laufenden System optimiert werden.

Komplette Datenverarbeitungsmodule, die als logische Sensoren betrachtet werden, können sowohl durch ihre Eingangsdaten als auch durch Anfragen an ihre Ausgangsdaten angestoßen werden. Darüber hinaus können Softwareagenten die Operatoren und Datensequenzen eines logischen Sensors gezielt ansteuern, konfigurieren und kontinuierlich triggern, wofür das in dieser

Arbeit vorgestellte Agentenmodell die Grundlagen schafft. Die agentenbasierte Steuerung der logischen Sensoren hat gegenüber einer rein daten- oder anfragegetriebenen Sensorarbeitsweise (kontinuierlicher oder getriggerter Modus) den Vorteil, daß mit Hilfe der Agenten die Datenrate jedes einzelnen Ausgangsdatums flexibel an den aktuellen Bedarf und die zur Verfügung stehenden Ressourcen angepaßt werden kann.

Schließlich ist es eine wichtige Aufgabe des vorgestellten Ansatzes, verschiedene Programmebenen sowohl logisch als auch programmtechnisch voneinander zu trennen, um so die Komplexität derartig großer dynamischer und verteilter Programmsysteme besser beherrschbar zu machen. Die wichtigsten dieser Ebenen sind

Datenverarbeitungsebene: in ihr werden die konkreten, applikationsabhängigen Datentypen und Operatoren definiert;

Datenflußebene: sie beschreibt den Datenfluß zwischen Daten und Operatoren, wobei von den konkreten Typen weitestgehend abstrahiert wird;

Programmsteuerungsebene: in ihr erfolgt der Aufruf der Operatoren und die Zuweisung der extrahierten Datenobjekte;

Parallelisierungsebene: in ihr lassen sich Aufgabenparallelitäten und Pipelining spezifizieren, ohne daß explizit der Kontrollfluß sequentialisiert werden muß;

Datenkommunikationsebene: sie definiert Mechanismen für den Datenaustausch zwischen verschiedenen Prozessen sowie zwischen Prozessen und dem Dateisystem;

Benutzerinteraktionsebene: sie bietet Möglichkeiten für die Visualierung der Daten und für die Einflußnahme auf das Programm durch den Benutzer bzw. Entwickler.

Dieser Ansatz erlaubt es beispielsweise, einzelne Berechnungsverfahren durch schnellere oder genauere zu ersetzen, ohne daß davon der Datenfluß, die Datenzugriffe oder die Programmsteuerung betroffen wären. Weiterhin ermöglicht er einen transparenten Zugriff auf Daten, die von einem anderen Rechner bereitgestellt werden, was die Entwicklung verteilter Systeme unterstützt. Die Verfahren für den Datenaustausch mit dem Dateisystem oder anderen Rechnern können unabhängig vom Rest des Systems ausgetauscht werden, genauso wie z.B. die für die Visualisierung der verschiedenen Daten und Objektaktivitäten zuständigen Klassen und Methoden.

Die beschriebenen Objektmodelle wurden prototypisch als C++-Klassenbibliothek implementiert. Mit ihr wurden wesentliche der in dieser Arbeit untersuchten Konzepte realisiert. Basierend auf diese Bibliothek wurde an der TU München die Software für eine Mannschaft fußballspielender Roboter entwickelt, implementiert und in verschiedenen internationalen RoboCup-Wettbewerben mit Erfolg eingesetzt. Die durch die Softwarebibliothek bereitgestellten Datenmodelle, die realisierte Trennung der verschiedenen Programmebenen sowie die integrierten Visualisierungs- und Interaktionsmöglichkeiten trugen wesentlich dazu bei, die Programmentwicklung zu beschleunigen und zu modularisieren.

8.4 Ausblick

Mit den in dieser Arbeit vorgestellten Konzepten und Objektmodellen für die Repräsentation dynamischer Datenfolgen wurde weitestgehend wissenschaftliches Neuland betreten, weswe-

gen eine Reihe von Punkten noch offen geblieben ist oder nur gestreift werden konnte. Für eine Fortführung der Arbeit ergeben sich damit verschiedene interessante Anknüpfungspunkte. Mit der prototypisch implementierten Datenfolgenbibliothek wurden wesentliche Konzepte der Arbeit realisiert, in einigen Bereichen ist die Implementierung jedoch noch nicht vollständig. Dies betrifft z.B. die vollständige Unterstützung der Pipelining-Parallelisierung sowie das Einlesen von Datenflußbeschreibungen aus einer Konfigurationsdatei.

Die im Rahmen dieser Arbeit durchgeführten Tests standen in erster Linie unter dem Blickwinkel der Beherrschbarkeit der großen Komplexität der Multisensor-Multirobotersysteme im RoboCup-Scenario, was sich anhand von flexiblen Konfigurationsmöglichkeiten, der guten Skalierbarkeit der Entwicklungen sowie des leichten Austauschs und der Verifikation der verschiedensten Datenverarbeitungsverfahren beweisen mußte. Die Performanzanalysen beschränkten sich bisher im wesentlichen auf die Messung des Overheads, der durch die Anwendung der vorgestellten Modelle entsteht. Dabei sollte es zunächst genügen zu zeigen, daß dieser vernachlässigbar klein ist und keine wesentlichen Auswirkungen auf die zu erzielende Datenrate besitzt. Was fehlt sind detaillierte Messungen, z.B. hinsichtlich des Laufzeitgewinns bei verschiedenen Parallesierungsstrategien in Mehrprozessorsystemen.

Die implementierte Bibliothek stellt Mechanismen bereit, mit denen zur Laufzeit Datenflußgraphen aufgebaut, modifiziert und konfiguriert werden können. Darüber hinaus ist es möglich, in laufende Applikationen nachträglich neue Operatoren hinzuzubinden und diese in Funktoren gekapselt in einen bestehenden Datenflußgraphen zu integrieren. Und schließlich stehen verschiedene Hilfsmittel und Benutzerschnittstellen zur Verfügung, mit denen zur Laufzeit die unterschiedlichsten Sensordaten und Statusinformationen ausgegeben werden können, wobei die Visualisierungwerkzeuge flexibel ein- und ausschaltbar sind. Damit sind die grundlegenden Voraussetzungen gelegt für ein graphisches oder textuelles Programmier- bzw. Rapid-Prototypingsystem, mit dem interaktiv beliebig komplexe Bildfolgenprogramme erstellt und — unter realen Laufzeitbedingungen und direkt in der Zielapplikation — sofort getestet werden können. Durch ein solches System ließen sich die jeweiligen Vorteile von graphischen und textuellen sowie von interpretierenden und compilierenden Entwicklungswerkzeugen verbinden.

Das Agentenmodell orientiert sich bisher im wesentlichen an den wichtigsten, für die Steuerung der logischen Sensoren erforderlichen Eigenschaften. Hier wären, um die Möglichkeiten der vorgestellten Konzepte voll ausschöpfen zu können, Erweiterungen um Verfahren notwendig, die basierend auf einer Analyse der Sensorzugriffe, der Laufzeit der einzelnen Funktoren und der Ressourcenauslastung eine (Re-) Konfiguration der Datenflußgraphen durchführen können.

Für eine weitergehende Auswertung der Dynamik der Anwendungsdomäne wird es in Zukunft notwendig sein, Unsicherheiten bei der Zeitmodellierung zu berücksichtigen. Dies betrifft zum einen die Messung der Zeitpunkte bestimmter realer Aktionen, wie z.B. der Sensordatenerfassung. In vielen Fällen ist eine exakte Zeitmessung nicht möglich, so daß die zeitliche Einordnung einer Aktion durch eine Wahrscheinlichkeitsfunktion oft besser beschrieben wird als durch einen einfachen Zeitpunkt. Zum anderen betrifft dies die Formulierung von zeitlichen Restriktionen, da viele reale Anforderungen durch einen kontinuierlichen Übergang besser modelliert werden als durch harte Zeitschranken.

Weitere untersuchenswerte Ansatzpunkte ergeben sich schließlich aus einer möglichen Aufweichung des in dieser Arbeit postulierten Paradigmas der Eindeutigkeit von Wertzuweisungen für die betrachteten Meßzeitpunkte. Als interessanteste Erweiterung soll hier die Möglichkeit

des *Backtrackings* auf der Zeitachse genannt werden. Ergeben sich z.B. bei der Sensordatenauswertung für bestimmte Merkmale verschiedene Interpretationsmöglichkeiten, kann so, statt alle Varianten parallel zu verfolgen und dadurch vielleicht unnötig Rechenzeit zu verschwenden, nur der erfolgversprechenste Zweig verfolgt werden. Erweist sich dieser innerhalb einer durch die Tiefe der Datensequenzen vorgegebenen Maximalzeit als falsch, können Teile der Sensordatenverarbeitung zum Ausgangszyklus zurückkehren und von dort aus mit einer alternativen Interpretation die Datenverarbeitung vergangener Zyklen — u.U. mit einer reduzierten Datenrate — erneut durchlaufen.

Literaturverzeichnis

[Act99] ActivMedia Robotics, *www.activrobots.com*, 1999.

[AK83] James F. Allen, Johannes A. Koomen, *Planning Using a Temporal World Model*, In *Eigth International Joint Conference on Artificial Intelligence (IJCAI-83)*, Karlsruhe, 1983, pp. 741–747.

[All83] James F. Allen, *Maintaining Knowledge about Temporal Intervals*, Communications of the ACM **26** (1983), Nr. 11, pp. 832–843.

[All84] James F. Allen, *Toward the General Theory of Action and Time*, Artifical Intelligence **23** (1984), pp. 123–154.

[Ame92] Amerinex Artificial Intelligence, Inc., *General Support Tools for Image Understanding*, Technical Report, Amerinex Artificial Intelligence, Inc., Amherst, MA, 1992.

[AO94] H. Aust, M. Oerder, *Generierung einer Datenbankanfrage aus einem gesprochenen Satz mit einer stochastisch attributierten Grammatik*, In W. G. Kropatsch, H. Bischof (Hrsg.), *Mustererkennung 1994*, Wien, 16. Symposium der Deutschen Arbeitsgemeinschaft für Mustererkennung (DAGM) und 18. Workshop der ÖAGM, Universität Wien, Springer-Verlag, 1994, pp. 230–237.

[BBG94] Martin Brunzema, Horst Burmeister, Dimitris Gerogiannis, *Parallelization of an Image Aanlysis Application: Problems, Results and a Solution Framework*, In *12th International Conference on Pattern Recognition (ICPR '94)*, Vol. III, 1994, pp. 406–411.

[BD89] Mark Boddy, Thomas Dean, *Solving time-dependent planning problems*, In *Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI, 1989, pp. 979–984.

[BFRR95] Thomas Bräunl, Stefan Feyrer, Wolfgang Rapf, Michael Reinhardt, *Parallele Bildverarbeitung*, Addison-Wesley Publishing Company, Bonn, Paris, 1995.

[BHKS99] Thorsten Bandlow, Robert Hanek, Michael Klupsch, Thorsten Schmitt, *Agilo RoboCuppers: RoboCup Team Description*, In *RoboCup-99 Workshop*, Stockholm, August 1999.

[Bis97] R. Bischoff, *A Humanoid Mobile Manipulator fpr Service Tasks*, In *International Conference on Fields ans Service Robotics*, Canberra, 1997, pp. 508–515.

[BJR98] Grady Booch, Ivar Jacobson, James Rumbaugh, *The Unified Modeling Language User Guide*, Addison-Wesley Object Technology Series, Addison-Wesley Publishing Company, Reading, MA, 1998.

[BKHS99] Thorsten Bandlow, Michael Klupsch, Robert Hanek, Thorsten Schmitt, *Fast Image Segmentation, Object Recognition and Localization in a RoboCup Scenario*, In *Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, Stockholm, August 1999.

[Boo94] Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2nd Auflage, Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[Brz94] Ch. Brzoska, *Temporallogisches Programmieren*, Dissertation, Fakultät für Informatik der Universität Karlsruhe (TH), 1994.

[Coa97] Peter Coad, *Object Models: Strategies, Patterns, and Applications*, 2nd Auflage, Yourdon Press, New Jersey, 1997.

[Cop92] James O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley Publishing Company, Reading, MA, 1992.

[DM99] E.D. Dickmanns, M. Maurer, *Eine Systemarchitektur für sehende autonome Fahrzeuge*, Automatisierungstechnik **47** (1999), Nr. 2, pp. 70–79.

[DMP91] Rina Dechter, Itay Meiri, Judea Pearl, *Temporal Constraint Networks*, Artifical Intelligence **49** (1991), pp. 61–95.

[Dor89] Jürgen Dorn, *Wissensbasierte Echtzeitplanung*, Künstliche Intelligenz, Vieweg, Braunschweig/Wiesbaden, 1989.

[ES96] Wolfgang Eckstein, Carsten Steger, *Interactive Data Inspection and Program Development for Computer Vision*, In Georges G. Grinstein, Robert F. Erbacher (Hrsg.), *Visual Data Exploration and Analysis III*, Proc. SPIE 2656, 1996, pp. 296–309.

[ES97] Wolfgang Eckstein, Carsten Steger, *Architecture for computer vision application development within the HORUS system*, Journal of Electronic Imaging **6** (1997), Nr. 2, pp. 244–261.

[FS97] Martin Fowler, Kendall Scott, *UML Destilled. Applying the Standard Object Modeling Language*, Addison Wesley Longman, Inc, 1997.

[FU94] Afonso Ferreira, Stephane Ubeda, *Ultra-Fast Contour Tracking with Applications to Thinning*, Pattern Recognition **27** (1994), Nr. 7, pp. 867–878.

[Gam96] Johann Gamper, *A Temporal Reasoning and Abstraction Framework for Model-Based Diagnosis Systems*, DISKI 136, Infix, Sankt Augustin, 1996.

[GB98] V. Graefe, R. Bischoff, *Vision-Guided Intelligent Robots*, In *International Conference on Mechatronics and Machine Vision in Practice*, Nanking, 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Longman, Inc, Reading, MA, 1995.

[Hai82] Brent T. Hailpern, *Verifying Concurrent Processes Using Temporal Logic*, Springer-Verlag, Berlin, 1982.

[Hil95] Bernd Hildebrandt, *Struktur und Bedeutung temporaler Konstituenten in einem sprachverstehenden Dialogsystem*, DISKI 104, Infix, Sankt Augustin, 1995.

[Hof99] Markus Hofbauer, *Implementation eines graphischen Benutzerinterfaces für die Interaktion mit komplexen Bildfolgenprogrammen*, Diplomarbeit, Forschungs- und Lehreinheit Informatik IX, Technische Universität München, 1999.

[HS93] Robert M. Haralick, Linda G. Shapiro, *Computer and Robot Vision*, Vol. II, Addison-Wesley Publishing Company, Reading, MA, 1993.

[HTI90] M. Hirakawa, M. Tanaka, T. Ichikawa, *An Iconic Programming System: HI-VISUAL*, IEEE Transactions on Software Engineering **16** (1990), Nr. 10, pp. 1178–1184.

[HUKW86] L. Hitzenberger, R. Ulbrand, H. Kritzenberger, P. Wenzel, *FACID - Fachsprachlicher Corpus informationsabfragender Dialoge*, Techn. Bericht, Universität Regensburg, Regensburg, 1986.

[HVD+99] R. Herpers, G. Verghese, K. Derpanis, R. McCready, J. MacLean, A. Levin, D. Topalovic, A. Jepson, J.K. Tsotsos, *Detection and Tracking of Faces in Real Environments*, In *IEEE Int. Workshop on Recognition, Analysis, and Tracking of Faces and Gestures in Real-Time Systems (RATFG-RTS'99)*, Corfu, Greece, IEEE Computer Society Press, September 1999, pp. 96–104.

[Ill95] Klaus Illgner, *Hierarchical Joint Estimation of Motion and Segmentation*, In *17. Symposium der Deutschen Arbeitsgemeinschaft für Mustererkennung (DAGM)*, Bielefeld, 1995.

[IW94] A. Iwansky, W. Wilhelmi (Hrsg.), *Lexikon der Computergrafik und Bildverarbeitung*, Vieweg, Braunschweig/Wiesbaden, 1994.

[Jäh97] Bernd Jähne, *Digitale Bildverarbeitung*, 4. Auflage, Springer-Verlag, Berlin, 1997.

[JBR99] Ivar Jacobson, Grady Booch, James Rumbaugh, *Unified Software Development Process*, Addison-Wesley Object Technology Series, Addison-Wesley Publishing Company, Reading, MA, 1999.

[Jos99] Jürgen Jost, *Entwurf und Aufbau eines Systems für die kamerabasierte Analyse der Dynamik mobiler Roboter*, Systementwicklungsprojekt, Fakultät für Informatik, Technische Universität München, 1999.

[K+97] Hiroaki Kitano, et al., *RoboCup Synthetic Agent Challenge 97*, In *Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, 1997.

[KAK+97a] Hiroaki Kitano, Minoru Asada, Yasou Kuniyoshi, Itsuki Noda, Eiichi Osawa, *RoboCup: The Robot World Cup Initiative*, In *Proceedings of the First International Conference on Autonomous Agent (Agent-97)*, ACM Press, 1997.

[KAK+97b] Hiroaki Kitano, Minoru Asada, Yasou Kuniyoshi, Itsuki Noda, Eiichi Osawa, Hitoshi Matsubara, *RoboCup: A Challenge Problem for AI*, AI Magazin (1997), pp. 73–85.

[KANM98] Hiroaki Kitano, Minoru Asada, Itsuki Noda, Hitoshi Matsubara, *RoboCup: Robot World Cup*, IEEE Robotics and Automation Magazin (1998), pp. 30–36.

[KE96] Michael Klupsch, Wolfgang Eckstein, *Object-Oriented Image Processing in Smalltalk: Using Complex Operator Objects*, In Nagib C. Callaos (Hrsg.), *International Conference on Information Systems, Analysis and Synthesis — ISAS '96*, Orlando, FL, International Institute of Informatics and Systemics (IIIS), 1996, pp. 833–839.

[KHS99] Volker Krüger, Alexander Happe, Gerald Sommer, *Affine Real-Time Face Tracking using Wavelets*, In *IEEE Int. Workshop on Recognition, Analysis, and Tracking of Faces and Gestures in Real-Time Systems (RATFG-RTS'99)*, Corfu, Greece, IEEE Computer Society Press, September 1999.

[KLZ+99] Michael Klupsch, Maximilian Lückenhaus, Christoph Zierl, Ivan Laptev, Thorsten Bandlow, Marc Grimme, Ignaz Kellerer, Fabian Schwarzer, *Agilo RoboCuppers: RoboCup Team Description*, In Minoru Asada, Hiroaki Kitano (Hrsg.), *RoboCup-98: Robot Soccer World Cup II*, Springer-Verlag, Berlin, 1999, pp. 446–451.

[Kon89] Kontron Bildanalyse, Eching, *IMCO 1000 Product Information*, 1989.

[KR94] Konstantinos Konstantinides, John R. Rasure, *The Khoros Software Development Environment for Image and Signal Processing*, IEEE Transactions on Image Processing **3** (1994), Nr. 3, pp. 243–252.

[Krö87] Fred Kröger, *Temporal Logic of Programs*, EATCS Monographs on Theoretical Computer Science, Vol. 8, Springer-Verlag, Berlin, 1987.

[Lam83] Leslie Lamport, *Specifying Concurrent Program Modules*, In *ACM Transactions on Programming Languages and Systems, Vol. 5, No. 2*, 1983, pp. 190–222.

[Lüc00] Maximilian Lückenhaus, *Agentenbasierte, automatische Parallelisierung in der Bildanalyse*, Dissertation, Fakultät für Informatik, Technische Universität München, 2000.

[Mac93] Alan Mackworth, *On Seeing Robots*, Computer Vision: Systems, Theory, and Applications (1993), pp. 1–13.

[Mat93] Matrox Electronic Systems Ltd., Dorval, QU, Canada, *Image Series*, 1993.

[McD82] Drew McDermott, *A Temporal Logic for Reasoning About Processes and Plans*, Cognitive Science **6** (1982), pp. 101–155.

[MD97] M. Maurer, E.D. Dickmanns, *A System Architecture for Autonomous Visual Road Vehicles Guidance*, In *Conf. on Intelligent Transportation Systems (ITSC '97*, Boston, MA, IEEE, November 1997.

[Mel95] Wolfgang Melchert, *Ein Ansatz zur Formalisierung der Echtzeit-Bildauswertung und seine Nutzung zur automatischen Erzeugung lauffähiger Maschinenprogramme*, DISKI 87, Infix, Sankt Augustin, 1995.

[MG95] Uwe Meyer-Gruhl, *Konzepte für eine wissensbasierte visuelle Sprache zur Bildanalyse*, Dissertation, Fakultät für Informatik, Technische Universität München, November 1995.

[MKB+95] J. L. Mundy, C. Kohl, T. Binford, D. Lawton, T. Boult, T. O'Donnell, M. C. Chiang, S. Fenster, D. Morgan, A. Hanson, R. Beveridge, K. Prince, R. Haralick, V. Ramesh, T. Strat, *IUE Overview Document. Image Understanding Environment Program*, Tech. Report, DARPA, June 1995.

[Mun93] J. L. Mundy, *The Image Understanding Environment: Overview*, In *Image Understanding Workshop*, San Francisco, CA, Morgan Kaufmann Publishers, 1993.

[NJ95] Sateesha G. Nadabar, Anil K. Jain, *Fusion of Range and Intensity Images on a Connection Maschine (CM-2)*, Pattern Recognition **28** (1995), Nr. 1, pp. 11–26.

[OA94] M. Oerder, H. Aust, *A Realtime Prototype of an Automatic Inquery System*, In *International Conference on Spoken Language Processing (ICSLP-94)*, Yokohama, Japan, Acoustical Society Japan, 1994, pp. 703–706.

[Ost95] Jonathan S. Ostroff, *Abstraction and Composition of Discrete Real-Time Systems*, Tech. Report, Department of Computer Science, York University, North York Ontario, Canada, 1995.

[Pau92] Dietrich W. R. Paulus, *Objektorientierte und wissensbasierte Bildverarbeitung*, Vieweg, Braunschweig/Wiesbaden, 1992.

[PBC+97] B. Pell, D. Bernard, S. Chien, E. Gat, N. Muscettola, P. Nayak, M. Wagner, B. Williams, *An Autonomous Spacecraft Agent Prototype*, In *Proceedings of the First International Conference on Autonomous Agent (Agent-97)*, ACM Press, 1997.

[PH95] Dietrich W. R. Paulus, Joachim Hornegger, *Pattern Recognition and Image Processing in C++*, Vieweg, Braunschweig/Wiesbaden, 1995.

[PJK98] Andrew Price, Andrew Jennings, John Kneen, *RoboCup97: An Omnidirectional Perspective*, In Minoru Asada, Hiroaki Kitano (Hrsg.), *RoboCup-97: Robot Soccer World Cup I*, Springer-Verlag, Berlin, 1998, pp. 320–332.

[PL94] Arthur R. Pope, David G. Lowe, *Vista: A Software Environment for Computer Vision Research*, In *Computer Vision and Pattern Recognition*, 1994, pp. 768–772.

[Pre94] Wolfgang Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley Publishing Company, 1994.

[RBP+91] James Rumbaugh, M. Blah, W. Premerlanie, F. Eddy, W. Lorensen, *Object-Oriented Modelling and Design*, Prentice-Hall International, Inc., Eaglewood Cliffs, NJ, 1991.

[Rit86] Jean-Francois Rit, *Propagating Temporal Constraints for Scheduling*, In *Proceedings of the 1986 National Conference on AI (AAAI-86)*, Philadelphia, PA, 1986, pp. 383–388.

[RK94] John Rasure, Steven Kubica, *The Khoros Application Development Environment*, In Henrik I. Christensen, James L. Crowley (Hrsg.), *Experimental Environments for Computer Vision and Image Processing*, Singapore, World Scientific Publishing Co. Pte. Ltd., 1994, pp. 1–32.

[Sah93] Michael K. Sahota, *Real-time Intelligent Behaviour in Dynamic Environments: Soccer-playing Robots*, Master's thesis, Iniversity of British Columbia, 1993.

[Sch94] Steffen Schäfer, *Objektorientierte Entwurfsmethoden: Verfahren zum objektorientierten Softwareentwurf im Überblick*, Addison-Wesley Publishing Company, Bonn, Paris, 1994.

[Sch96] Karl H. Schäfer, *Unscharfe zeitlogische Modellierung von Situationen und Handlungen in Bildfolgenauswertung und Robotik*, DISKI 135, Infix, Sankt Augustin, 1996.

[Ste87] Stemmer PC-Systeme GmbH, Puchheim, *Schneller als Echtzeit: Das Hochleistungs-System für die Bildverarbeitung Serie 150/151 von Imaging Technology*, 1987, Stemmer News Letter.

[Wil90] T. D. Williams, *Image Understanding Tools*, In *10th International Conference on Pattern Recognition*, IEEE Computer Society Press, 1990, pp. 606–610.

[Win94] Andreas Winklhofer, *Zeitrepräsentation und merkmalsgesteuerte Suche zur Terminplanung*, DISKI 58, Infix, Sankt Augustin, 1994.

[YMH+86] I. Yoshimoto, N. Monden, M. Hirakawa, M. Tanaka, T. Ichikawa, *Interactive Iconic Programming Facility in HI-VISUAL*, In *IEEE Workshop on Visual Languages*, Dallas, TX, 1986, pp. 34–41.

[YOM+98] K. Yokota, K. Ozaki, A. Matsumoto, K. Kawabata, H. Kaet, *Omni-directional Autonomous Robots Cooperating for Team Play*, In Minoru Asada, Hiroaki Kitano (Hrsg.), *RoboCup-97: Robot Soccer World Cup I*, Springer-Verlag, Berlin, 1998, pp. 333–347.

[ZB96] W. D. Zimmer, E. Bonz, *Objektorientierte Bildverarbeitung – Datenstrukturen und Anwendungen in C++*, Carl Hanser Verlag, München, Wien, 1996.