2 Nebenläufige Systeme

Motivation

Für den Einsatz von Nebenläufigkeit in Systemen gibt es verschiedene Argumente – auch Echtzeitanforderungen spielen dabei eine Rolle. Nebenläufige Programmverarbeitung ist in vielen unterschiedlichen Inkarnationen in Programmiersprachen vorhanden und hat vor allem für neuere Sprachen wie etwa Java eine große Bedeutung. Sie ist das Grundmodell für Multitasking und Multithreading.

In einem nebenläufigen Programm werden mehrere Kontrollflüsse erzeugt, die potentiell gleichzeitig ausgeführt werden können. Findet die Ausführung tatsächlich gleichzeitig statt, so spricht man von Parallelität (Butenhof, 1997). In deklarativen Programmiersprachen kann die Nebenläufigkeit einzelner Programmteile oftmals abgeleitet werden, z. B. durch die Konfluenzeigenschaft referentiell transparenter funktionaler Sprachen (Peyton Jones, 1989). Bei imperativen Programmiersprachen hingegen fehlt die Möglichkeit einer Nebenläufigkeit, da imperative Programme ein starres Ablaufschema aufweisen.

Zwei Prozesse sind dann nebenläufig, wenn sie unabhängig voneinander arbeiten können und es keine Rolle spielt, welcher der beiden Prozesse zuerst ausgeführt oder beendet wird. Derartige nebenläufige Prozesse können jedoch indirekt voneinander abhängig sein, da sie möglicherweise gemeinsame Ressourcen beanspruchen und untereinander Nachrichten austauschen. Dies macht eine Synchronisation an bestimmten Knotenpunkten in den Prozessen notwendig. Dieser Sachverhalt kann zu Problemen führen, die von schwerwiegenden Fehlern bis hin zu Programmstillstand und Absturz führen können. Eines der wesentlichen Ziele der Nebenläufigkeit ist eine gleichmäßige Ressourcen-Auslastung, wobei diese besonders in objektorientierten Programmiersprachen wünschenswert ist.

Kapitelübersicht

In diesem Kapitel werden wir nach einer Einführung in die Themen Multitasking, Multithreading und Prozesssynchronisiation

bzw. -kommunikation Modelle für Nebenläufigkeit kennenlernen und uns schließlich verteilten eingebetteten Systemen zuwenden.

2.1 Einführung

Einsatz von Nebenläufigkeit

Grundsätzlich werden folgende zwei Argumente für den Einsatz von Nebenläufigkeit angeführt (Butenhof, 1997):

- Viele Probleme lassen sich einfacher modellieren, wenn sie als mehr oder weniger unabhängige Aktivitäten verstanden und durch entsprechende Sprachkonstrukte umgesetzt werden können. Jede Aktivität kann dann isoliert entworfen und implementiert werden. Nebenläufigkeit wird in diesem Kontext also als Abstraktionskonstrukt verstanden, das den Softwareentwicklungsprozess vereinfacht. Ob tatsächlich eine parallele, also gleichzeitige Verarbeitung stattfindet, ist hierbei von untergeordneter Bedeutung.
- Rechner enthalten mehrere ausführende Entitäten, in der Regel einen oder mehrere Prozessoren sowie Ein-Ausgabe-Geräte. Will man diese Ressourcen gleichzeitig nutzen, so muss man ihnen entsprechend mehrere Anweisungsströme zuführen. Diese können dann parallel ausgeführt werden und damit zu einer Reduktion der Ausführungszeit führen. Das Ziel der nebenläufigen Programmdefinitionen ist hier also Parallelität zur schnelleren Programmausführung, indem man das Programm der vorhandenen Hardwarekonfiguration anpasst.

Für beide Modelle existiert eine mehr oder weniger kanonische Implementierung. Sie wirken in unterschiedlichen Systemebenen und weisen daher unterschiedliche Charakteristika bezüglich ihrer Performanz und Skalierbarkeit auf. Eine der entscheidenden Kennzahlen ist hierbei die Zahl der unterstützten nebenläufigen Aktivitäten. Diese ist im ersten Fall durch die Problemgröße definiert, während im zweiten Fall die Hardwarekonfiguration die Grenze zieht.

2.1.1 Multitasking

Multitasking ist die Fähigkeit von Software, beispielsweise Betriebssystemen, mehrere Aufgaben scheinbar gleichzeitig auszuführen. Dabei werden die verschiedenen Prozesse in so kurzen Abständen immer abwechselnd aktiviert, dass für den Beobachter der Eindruck der Gleichzeitigkeit entsteht. Man spricht daher auch oft von „Quasi-Parallelität".

Konzepte

Hierbei gibt es verschiedene Konzepte zur Handhabung des Multitasking (vgl. Kapitel 3). Das am häufigsten angewandte Konzept ist das präemptive Multitasking, bei dem der Betriebssystemkern die Abarbeitung der einzelnen Prozesse steuert und jeden Prozess nach einer bestimmten Abarbeitungszeit zu Gunsten anderer Prozesse anhält. Eine alternative Form des Multitasking ist das u. a. von älteren Windows-Versionen bekannte kooperative Multitasking. Bei dieser Form des Multitasking ist es jedem Prozess selbst überlassen, wann er die Kontrolle an den Kern zurückgibt. Dies birgt den signifikanten Nachteil, dass Programme die nicht kooperieren bzw. Fehler enthalten, das gesamte System zum Absturz bringen können. Bei Echtzeitsystemen ist das Multitasking besonders auf die geforderten Reaktionszeiten hin optimiert (siehe ebenfalls Kapitel 3).

Scheduler

Der Teil des Betriebssystems, der die Prozessumschaltung übernimmt, heißt Scheduler (siehe Abschnitt 3.3.6). Die dem Betriebssystem bekannten, aktiven Programme bestehen aus Prozessen. Ein Prozess setzt sich aus einem Programmcode und den Daten zusammen und besitzt einen eigenen Adressraum. Die Speicherverwaltung des Betriebssystems trennt die Adressräume der einzelnen Prozesse. Daher ist es unmöglich, dass ein Prozess den Speicherraum eines anderen Prozesses sieht. Schließlich gehören zu jedem Prozess noch Ressourcen wie z. B. geöffnete Dateien oder belegte Schnittstellen.

2.1.2 Multithreading

Die Fähigkeit mehrere Bearbeitungsstränge in einem Prozess gleichzeitig abarbeiten zu können, wird Multithreading genannt. Es ist von dem jeweilig verwendeten Betriebssystem bzw. von der Hardware abhängig, ob es sich dabei um eine reale oder nur um eine scheinbare Gleichzeitigkeit handelt. Durch Verwendung eines

Mehrprozessorsystems und eines Betriebssystems, welches den Prozessen erlaubt mehrere CPUs für verschiedene Threads gleichzeitig zu verwenden, kann eine reale Gleichzeitigkeit erreichen werden. Ein Thread (deutsch „Faden“ bzw. „Ausführungsstrang“) ist

- eine selbstständige,
- ein sequentielles Programm ausführende,
- zu anderen Threads nebenläufig arbeitende,
- von einem Betriebs- oder Laufzeitsystem zur Verfügung gestellte

Aktivität. Werden Threads vom Betriebssystem selbst unterstützt, so spricht man von „nativen Threads“. Oft möchte man nicht nur auf Betriebssystemebene nebenläufig arbeiten können, sondern auch innerhalb eines einzigen Programms. Die Programmiersprache Java unterstützt dies beispielsweise in Form von Threads, deren nebenläufige Ausführung durch die virtuelle Maschine geschieht. Innerhalb eines Prozesses kann es mehrere Threads geben, die stets alle zusammen in dem selben Adressraum ablaufen. Mehrere Threads können wie Prozesse zeitlich verzahnt ausgeführt werden, so dass sich der Eindruck von Gleichzeitigkeit ergibt. Threads besitzen im Vergleich zu Prozessen den Vorteil, vom Scheduler sehr viel schneller umgeschaltet werden zu können, d. h. der Kontextwechsel vollzieht sich rascher. Der Ablauf wird dann vom Java-Interpreter geregelt, ebenso die Synchronisation und die verzahnte Ausführung. Die Sprachdefinition von Java lässt die Art der Implementierung – also nativ oder nicht – von Threads bewusst frei.

Die Verwendung von Threads führt im technischen Sinne nicht zu einer beschleunigten Ausführung. Im Gegenteil, durch Verwaltungssaufwände bei der Erzeugung und Koordination ergibt sich im Allgemeinen sogar eine insgesamt vergrößerte Ausführungszeit, jedoch bedingt die effizientere Ressourcennutzung einerseits die bessere Auslastung der vorhandenen Hardware und andererseits entsteht für den Anwender der Eindruck einer flüssigen Verarbeitung. Erst beim Einsatz von mehreren Prozessoren in einer Maschine ergibt sich ein echter positiver Laufzeiteffekt durch die Möglichkeit, Threads auf verschiedenen CPUs zur Ausführung zu bringen.

2.1.3 Prozesssynchronisation und -kommunikation

Die nebenläufige Programmierung von Systemen muss eine Synchronisation und Kommunikation ermöglichen. Nebenläufige Prozesse stehen in der Regel in Wechselwirkung, weshalb man sie nicht an beliebige Stellen in unabhängige Ausführungspfade aufteilen kann. Die korrekte Formulierung der Prozesswechselwirkungen nennt man Prozesssynchronisation. Unter Kommunikation versteht man den Nachrichtenaustausch zwischen nebenläufigen Prozessen.

Synchronisation und Kommunikation

Das Warten eines Prozesses auf ein Ereignis, das ein anderer Prozess auslöst, ist die einfachste Form der Prozesssynchronisation oder Prozesswechselwirkung. Eine Verallgemeinerung der Prozesssynchronisation stellt die Prozesskommunikation dar, d. h. die Zusendung von Daten von einem Prozess zu einem anderen. Sie erfordert das Warten auf das Ereignis „Eintreffen der Daten“ und erfordert außerdem die Bereitstellung eines logischen Datenübertragungsweges zwischen den Prozessumgebungen. Im Zusammenhang mit Synchronisation stößt man immer wieder auf das vielzitierte Problem der „dinierenden Philosophen“ (engl. dining philosophers). Es macht deutlich, dass Prozesse geschickt synchronisiert werden müssen. Tut man dies nicht, können Verklemmungen (engl. deadlocks) die Folge sein.

Verklemmungen (Deadlocks)

Eine Menge von Threads (Prozessen) heißt verklemmt, wenn jeder Thread (Prozess) dieser Menge auf ein Ereignis im Zustand „blockiert“ wartet, das nur durch einen anderen Thread (Prozess) dieser Menge ausgelöst werden kann. Im Prinzip warten dann diese Threads (Prozesse) ewig, da ja keiner dieser Threads jemals wieder auf den Prozessor zugeordnet wird, denn jeder dieser Threads ist ja blockiert. Eine Verklemmung liegt also nur dann vor, wenn eine Menge von Prozessen bzw. Threads, die entweder um eine Ressource konkurrieren oder miteinander kommunizieren, *dauerhaft* sind.

Verklemmungsursachen

Es gibt prinzipiell zwei verschiedene Ursachen von Verklemmung. Die erste Ursache von Verklemmung liegt innerhalb des nebenläufigen Programms selbst d. h. es liegt eine fehlerhafte (irrtümliche) Programmierung vor, die sich in der Regel nur bedingt auf das Restsystem auswirkt. Die zweite Ursache einer Verklemmung kann zwischen zwei oder mehreren unabhängigen Prozessen auf eine wenig durchdachte Ressourcenverwaltung im System zurückzuführen sein, die vergleichsweise schwerwiegende Auswirkungen auf das Gesamtsystem haben kann.

Im ersten Fall kann man diese Verklemmungssituation aus Anwendersicht relativ einfach mit Hilfe eines Timeouts erkennen. Diese Verklemmungssituation ist unter Umständen sogar reproduzierbar, während im zweiten Fall eine Verklemmung sporadisch auftritt, je nachdem welche sonstigen Anwendungen, die jede für sich Ressourcen benötigt, gleichzeitig im System abgearbeitet werden.

Verklemmungsbedingungen

Folgende Bedingungen müssen erfüllt sein, damit es zu einer Verklemmung kommen kann:

- Exklusivität der Ressourcennutzung
- Halten von Ressourcen beim Warten auf weitere Ressourcen
- Keine Verdrängungsmöglichkeit von Ressourcen
- Zirkuläre Wartebedingung

Maßnahmen

Jede dieser Bedingungen ist notwendig, um eine Verklemmung auszulösen. Es existieren verschiedene Strategien, um Verklemmungen entgegen zu wirken: Die Prävention, die Vermeidung von Deadlocks, das Entdecken und Beseitigen sowie – von eher theoretischer Bedeutung – das Ignorieren von Deadlocks.

2.2 Grundlegende Modelle für die Nebenläufigkeit

Anforderungen

An ein Modell für die Nebenläufigkeit werden nachstehende Anforderungen gestellt (Berry, 1998). Es muss

- einfach und intuitiv,
- kompositional,
- mathematisch fundiert, um Basis für formale Semantik und Verifikation zu bieten, und
- physikalisch sinnvoll

Modelle

sein. Drei grundlegende und vollkommen unterschiedliche Modelle bieten sich zur Realisierung dieser Anforderungen an. Sie werden durch Analogien zur Physik beschrieben (Berry, 1998):

- **Das Chemische Modell:** Die Recheneinheiten werden als eine Art Moleküle betrachtet. Die Kommunikation unter ihnen findet durch Kontakt und anschließende Reaktion statt.
- **Das Newtonsche Modell:** Die Recheneinheiten werden als Planeten angesehen. Die Planeten tauschen untereinander die Information bezüglich ihres Gewichts und Position aus und bewegen sich gemäß dieser Informationen und ihrer momentanen Geschwindigkeit und Beschleunigung. Dieser Informationsaustausch findet zu jeder Zeiteinheit statt und verbraucht keine Zeit.
- **Das Vibrationsmodell:** Die Recheneinheiten werden als Moleküle eines Kristallgitters angesehen. Wenn ein Molekül angestoßen wird, dann stößt es seine Nachbarn an, welche wiederum ihre Nachbarn anstoßen und so weiter und so fort. Dies erzeugt eine Welle, die sich mit definierter Geschwindigkeit im Kristallgitter ausbreitet.

Unterschiede

Alle drei Modelle erfüllen die oben genannten Anforderungen auf unterschiedliche Weise. Der Hauptunterschied ist dabei die erforderliche Zeit für das Aufbauen einer Kommunikation:

- *Chemisches Modell:* Immer unterschiedliche Zeit
- *Newtonsches Modell:* Keine Zeit, d. h. Null-Zeit
- *Vibrationsmodell:* Konstante Zeit

Das Chemische Modell ist nichtdeterministisch und asynchron. Seine mathematische Version bietet die Basis für die Semantik der Sprachen und des Kalküls der interaktiven Prozesse. Für reaktive Prozesse eignet es sich nicht, da die Rechtzeitigkeit nicht dargestellt werden kann.

Das Newtonsche Modell ist deterministisch und, da der Austausch der Information und ihre Verarbeitung in null Zeit stattfindet, „perfekt synchron“. Dieses Modell dient als Basis für die Definition der synchronen Sprachen und deren Semantik. Für deren Implementierung wird aber das komplexere Vibrationsmodell, wo die Information sich verzögert ausbreitet, verwendet, weil es besser der Arbeitsweise der Rechner entspricht.

Ein Beispiel, wo das Newtonsche und das Vibrationsmodell auch gemeinsam verwendet werden, ist der synchrone Schaltungsentwurf, vgl. Abbildung 2.1:

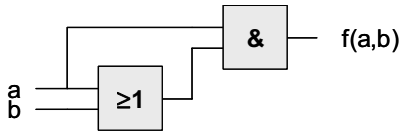

Abb. 2.1: Logische Schaltung

Hier wird angenommen, dass die Eingänge alle synchron in „0 Zeit“ ausgewertet werden (Newtonsche Modell). Aber in Wirklichkeit besteht diese Schaltung aus Gattern (logisches Und „&“ bzw. Oder „≥1“ im Beispiel in Abbildung 2.1), die eine Laufzeit besitzen (Vibrationsmodell). So liegt das Ergebnis von $(a \vee b)$ im Vergleich zu a mit der Verspätung Δ (= Gatterschaltzeit von „≥1“) an. Bei Einsatz von Schaltwerken in Schaltungen wird durch eine gemeinsame Taktung sichergestellt, dass die richtigen Signale ausgewertet werden. Dabei wird vorausgesetzt, dass der Abstand zweier konsekutiver Takt-Signale nicht kürzer ist als die Summe aller Gatterschaltzeiten Δ des längsten Pfades. Eine analoge Bedingung existiert für die Anwendung des Modells der perfekten Synchronität (siehe weiter unten).

2.3 Verteilte Systeme

Insbesondere Automobilhersteller und deren Zulieferer stehen im Bereich der Softwareentwicklung der Herausforderung einer stark steigenden Anzahl verteilter eingebetteter Spezialsysteme im Fahrzeug gegenüber. Gesucht wird daher eine Architektur-unabhängige Vorgehensweise zum Entwurf solcher Systeme, die noch keine Annahmen bezüglich der verteilten Implementierung trifft und das Ziel verfolgt, eine höhere Systemauslastung sowie eine bessere Wiederverwendung für weitere Baureihen zu gewährleisten. Ein Lösungsansatz hierfür wird in diesem Abschnitt skizziert.

Der aktuelle Trend der Elektronik-Entwicklung im Fahrzeug geht zu immer mehr Funktionalität, die auf einer ebenfalls stark wachsenden Anzahl von Steuergeräten bzw. Prozessoren implementiert wird. Um die 70 davon werden in Abhängigkeit der vom Kunden gewählten Ausstattungsvariante in aktuellen Oberklassemodellen eingesetzt. In den allermeisten Fällen führt jeder von ihnen eine ganz dedizierte Aufgabe durch. Ist die Arbeitslast des Prozessors hoch oder die Funktion stark sicherheitskritisch, macht diese Strategie durchaus Sinn.

Andererseits kommt die ausschließliche Verwendung eines Steuergeräts für eine einzige Aufgabe gerade im Bereich der

Komfortelektronik einer Verschwendung von Ressourcen gleich. Darüber hinaus wünschen sich viele Entwicklungsingenieure, dass eine einmal von ihnen erfolgreich entworfene Funktion für ein Fahrzeug auch in weiteren Baureihen mit einer ggf. ganz anderen Elektrik-/Elektronik-Architektur ebenfalls wiederverwendet werden kann – dies beschleunigt die Time-to-Market und spart Entwicklungskosten.

Partitionierung, Komponierbarkeit

Gesucht wird daher eine Architektur-unabhängige Vorgehensweise zum Entwurf solcher Systeme, die noch keine Annahmen bezüglich der verteilten Implementierung trifft und das Ziel verfolgt, eine höhere Systemauslastung sowie eine bessere Wiederverwendung für weitere Baureihen zu gewährleisten. Die hier verwendeten Schlüsselbegriffe sind die *„Partitionierung“* (Verteilung von Funktionen nach ihrer Entwicklung) und die *„Komponierbarkeit“*.

TTA, TTP

Komponierbarkeit bedeutet hier, dass Eigenschaften, die auf der Ebene der Komponenten gegeben sind, beispielsweise ein garantiertes zeitliches Verhalten, ebenfalls auf der Systemebene Gültigkeit besitzen. Eines von mehreren Beispielen hierfür ist die *Time Triggered Architecture (TTA),* eine *zeitgesteuerte* Architektur, die von der Technischen Universität Wien entwickelt wurde (Kopetz et al., 2002). Sie erlaubt mithilfe einer exakten Vorausplanung *aller* zeitlichen Aspekte des Systems dessen zeitliche Reaktionsdauer zu minimieren und Systemkomponenten so zusammenzusetzen, dass die Interaktionsmuster optimal zueinander passen. Alle Schnittstellen sind bezüglich des Zeitbereichs exakt definiert und verändern sich damit auch bei der Systemintegration nicht: Das Gesamtsystem ist komponierbar in Bezug auf das zeitliche Verhalten. Die exakte Kenntnis des zeitlichen Ablaufs im Vorhinein ist allerdings auch ein Nachteil der TTA, der zu einer geringen Flexibilität führt. Wann immer aber das Echtzeitverhalten verteilter eingebetteter Systeme optimiert werden muss, z. B. beim Drive-by-Wire, ist die TTA eine gute Wahl.

Einige Automobilhersteller haben diese Herausforderung bereits erkannt und mit entsprechenden Gegenmaßnahmen begonnen, konnten sie allerdings bisher mangels industriell verfügbarer, technologischer und methodischer Unterstützung noch nicht in die Serie umsetzen.

Auch bei den Softwarefirmen gibt es bereits vereinzelt erste Ansätze die Unterstützung bieten. So hat beispielsweise erst kürzlich ein Softwarehersteller ein CASE-Tool entwickelt, das es dem Entwickler erlaubt, Funktionen, die bereits mit Hilfe anderer Spezifikationswerkzeuge fertig entwickelt wurden, auf ein Prozessor-Netzwerk per Drag-and-Drop zu verteilen. Als Ergebnis wird ein Protokoll für die nach der Verteilung erforderliche

asynchrone Kommunikation zwischen den Prozessoren automatisch erstellt.

Dieser Ansatz ist sehr zu begrüßen, weil er den Entwickler bei der Umsetzung von Protokollinformation zur Kommunikation zwischen einem nun verteilten System entlastet. Genau dieser teilweise sehr komplexe Kommunikationsfluss sollte aber genau analysiert werden: Deadlocks müssen genauso vermieden werden wie endlose Signal-Pingpongs. Insgesamt muss Sorge getragen werden, dass das implementierte Verhalten des verteilten eingebetteten Systems hinsichtlich Funktionalität und zeitlichem Verhalten identisch mit dem ursprünglich geplanten System ist.

Grundvoraussetzung für die spätere Verteilung einer Funktion auf ein Prozessor-Netzwerk ist ein modularer Ansatz mit einer exakten Beschreibung aller Schnittstellen. Nur so können überhaupt Module, also Teilfunktionalitäten als potentielle Kandidaten zur Verteilung identifiziert werden.

Vor allem aber muss das richtige Kommunikationsprinzip für den zunächst noch rein virtuellen Austausch von Nachrichten zwischen diesen Modulen gewählt werden. Dabei sollte bedacht werden, dass die virtuelle Kommunikation ggf. zu keiner physikalischen führen muss. Dies ist immer dann der Fall, wenn beide Module auf ein und demselben Steuergerät implementiert werden.

Die asynchrone Kommunikation hat hier gleich mehrere Nachteile: Entsprechende Pufferbausteine zum Sammeln von Nachrichten an den Modulschnittstellen müssen vorgehalten werden und auch die Kommunikationsdauer bleibt gänzlich unspezifiziert, was für das Echtzeitverhalten des Systems negative Effekte haben könnte.

Perfekte Synchronie

Abhilfe schafft das sogenannte „perfekt synchrone“ Kommunikationsprinzip (siehe Kapitel 4). Es unterstellt auf der Ebene der Spezifikation zunächst, dass eine Reaktion, also eine Transition von einem Systemzustand zu einem anderen, ebenso wie eine virtuelle Kommunikation gar keine Zeit verbraucht. Zeit vergeht lediglich in stabilen Systemzuständen. „Keine Zeit zu verbrauchen“ darf hier keinesfalls missinterpretiert werden; es handelt sich dabei nur um eine Abstraktion der Realität, die annimmt, dass die tatsächliche Zeitspanne, die das später realisierte System verbrauchen wird, um seinen Zustand zu verändern kürzer sein wird als die Zeit, die zwischen dem Eintreffen aufeinanderfolgender Signale der Sensorik vergeht (vgl. logischer Schaltungsentwurf, Abbildung 2.1). Diese Annahme hat mehrere Vorteile: Die Reaktionszeiten sind in der Phase des Entwurfs noch unabhängig von der konkreten, ggf. verteilten Implementierung. Künstliche, vielleicht sogar falsche, zusätzliche Verzögerungszeiten werden nicht eingeplant; und schließlich kann auf diese Weise jede Reaktion in beliebig viele

Sub-Reaktionen zerlegt werden, ohne das zeitliche Verhalten der Spezifikation zu beeinflussen.

Bei den sogenannten „synchronen Sprachen“ wie etwa Esterel, Lustre und Signal wurde dieses Konzept bereits auf der Ebene der Programmierung eingebetteter Systeme erfolgreich umgesetzt. Allerdings wurden die Vorteile dieser Sprachen außerhalb der Hochschulen oder der französischen Avionik-Industrie bis dato noch nicht in ausreichendem Maße erkannt.

Darüber hinaus müssen diese Konzepte von der Ebene der Programmiersprachen auf die Ebene der Modellierungstechniken erweitert werden. Hierzu gibt es ebenfalls schon Lösungen für die Modellierung zustandsbasierter Systeme, z. B. für die an Statecharts angelehnte Techniken Argos oder μ-Charts (Scholz, 1998). Gerade letztere gibt dem Entwickler eine methodische Hilfestellung bei der Verwendung von Statecharts, um ein eingebettetes System von der ersten, gänzlich architekturabhängigen Idee bis hin zur ggf. verteilten Implementierung zu konstruieren und zu partitionieren. Es handelt sich dabei um ein rein konstruktives Verfahren: Hält der Benutzer einige vorgegebene Entwicklungsregeln streng ein, so verfeinert er das Systemverhalten nicht nur schrittweise, sondern es kann die Partitionierung der Spezifikation auf ein Prozessornetzwerk automatisch garantiert werden. Andernfalls ist eine nachgeschaltete Analyse erforderlich, die mögliche Kommunikationsprobleme ausschließt.