Ausarbeitung Zeit- und ereignisgesteuerte Echtzeitsysteme

Stephan Braun

Projektgruppe AUTOLAB der Universität Dortmund

Zeit- und ereignisgesteuerte Echtzeitsysteme

Inhaltsverzeichnis

I. ***Einführung:***

Von Echtzeitsystemen spricht man, wenn ein System ein Ergebnis innerhalb eines vorher fest definierten Zeitintervalls garantiert berechnet, also bis zu einer bestimmten Zeitschranke (deadline), die ihm durch seine Umgebung auferlegt wird.

[1]

Ein Echtzeitsystem muss also nicht nur ein Berechnungsergebnis mit dem richtigen Wert, sondern dieses auch noch rechtzeitig, liefern. Andernfalls hat das System versagt – es schlägt fehl.

„A real-time (computer) system is a (computer) system in which the correctness of the system behavior depends not only on the logical result of the computations, but also on the physical instant at which these results are produced.“ [2]

Ein RT-Computersystem ist immer Teil eines größeren Systems; dieses wird als Echtzeitsystem (ES) bezeichnet. An ein Echtzeitsystem stellt man i.A. Anforderungen nach

- Rechtzeitigkeit (fordert, dass das Ergebnis für den zu steuernden Prozess rechtzeitig vorliegen muss, z.B. Einhalten von Zykluszeiten und Abtastzeitpunkten)
- Gleichzeitigkeit (viele Aufgaben müssen parallel, mit ihren jeweiligen Zeitanforderungen, bearbeitet werden)
- Spontane Reaktion auf Ereignisse (das System muss auf zufällig auftretende interne oder externe Ereignisse aus dem Prozess innerhalb einer definierten Zeit reagieren)

„A real-time (computer) system must react to stimuli from the controlled object within time intervals dictated by its environment.“ [3]

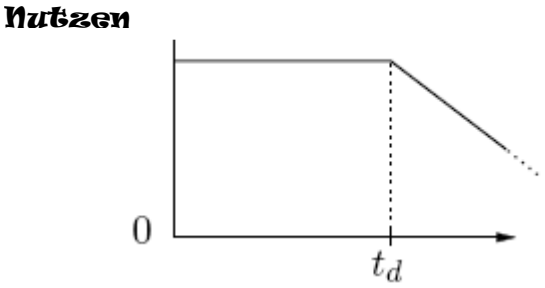
Abhängig von den Folgen des Verpassens einer deadline wird zwischen harter Echtzeit (hard real-time) und weicher Echtzeit (soft real-time) unterschieden.

a) Weiche Echtzeit: solche Systeme arbeiten normalerweise alle ankommenden Eingaben schnell genug ab. Die Antwortzeit erreicht einen akzeptablen Mittelwert; signifikante Abweichungen von diesen sind eher selten. **Das Überschreiten kann in gewissem Maße toleriert werden**.
   Beispiele:
   - ❖ System für Videokonferenzen - bei Verpassen einer deadline ruckelt lediglich das Bild, falls ein Bildsignal zu spät ankommt.
   - ❖ Periodisches Auslesen eines Temperatursensors

[1] Peter Marwedel, Folien zur Vorlesung Eingebettete Systeme, WiSe 06/07, Universität Dortmund

[2] Hermann Kopetz, Real-Time Systems, Kluwer Academic Publishers, 2. Ausgabe 1998, Seite 2

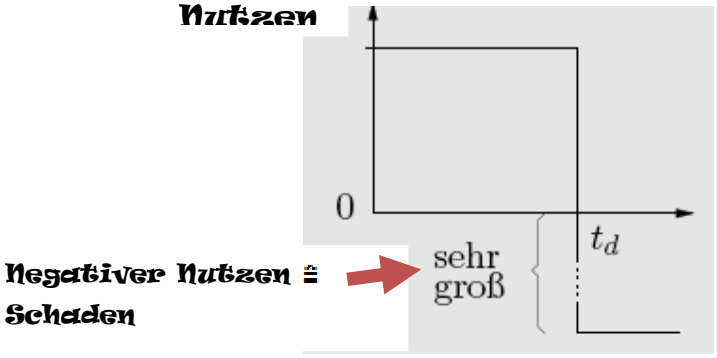
[3] Hermann Kopetz, Real-Time Systems, Kluwer Academic Publishers, 2. Ausgabe 1998, Seite 2



Nach dem Überschreiten der Deadline nimmt der Nutzen des Ergebnisses ab, allerdings ohne negativen Einfluss auf das Gesamtsystem zu nehmen.

b) Harte Echtzeit: eine Überschreitung der Antwortzeit wird als Fehler gewertet; Zeitschranken müssen auf jeden Fall eingehalten werden, andernfalls droht Schaden.

„If a catastrophe could result in case a deadline is missed, the deadline is called hard“[4]



Ein Computersystem wird hartes Echtzeitsystem *(hard real-time system)* genannt, wenn es zumindest eine harte deadline besitzt.

Beispiele:

- Rechtzeitiges Anhalten eines autonomen Fahrzeugs vor einer Ampel
- Elektronische Motorsteuerung – der Motor kann bei nicht korrekt eingehaltener harter deadline stottern/stehen bleiben; dies kann zu einem Ausfall, einem mechanischen Schaden oder Unfall führen.

Charakteristik	*Harte Echtzeit*	*Weiche Echtzeit*
Antwortzeit	hart	weich
Kontrolle des Tempos	Umgebung	Computer
Fehlererkennung	System	Benutzer
Sicherheit	kritisch	unkritisch
Redundanztyp	aktiv	Stand-by
Granularität	Millisekunde	Sekunde

[4] Hermann Kopetz, Real-Time Systems, Kluwer Academic Publishers, 2. Ausgabe 1998, Seite 2

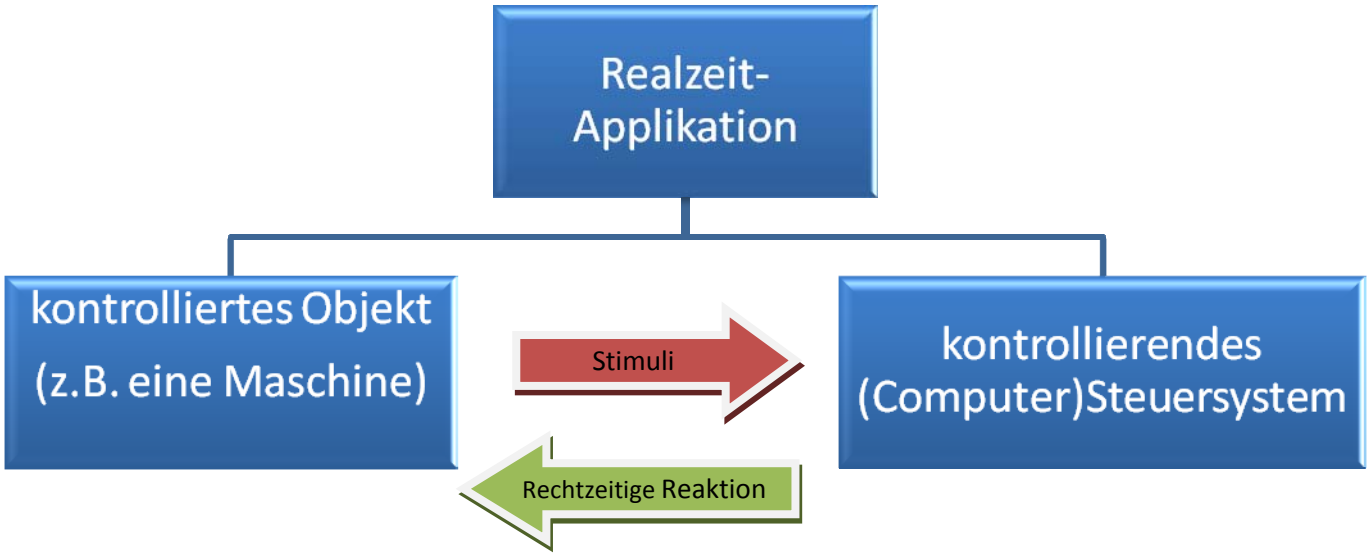
Fazit:

Das Design von Echtzeitsystemen (ES) ist eine Herausforderung: funktionelle Spezifikationen müssen unter Beachtung zeitlicher Beschränkungen erfüllt werden.

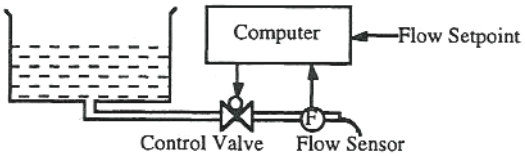
Diese zeitlichen Beschränkungen sind Eigenschaften des jeweiligen Systems; sie hängen von der jeweiligen Systemarchitektur, insbesondere der Anwendungssoftware, den Scheduling-Entscheidungen innerhalb des Betriebssystems, den Verzögerungen der Kommunikationsprotokolle und der zugrundeliegenden Hardware ab.

II. *Echtzeitsystemmodell:*

Echtzeitsysteme werden in Applikationen zur Kontrolle realer technischer Systeme verwendet.



Beispiel: Kontrolle des Flusses in einer Leitung:



[5]

Das Computersystem interagiert mit dem kontrollierten Objekt (hier: die zu kontrollierende Leitung), indem die Pumpenstellung (die Pumpe als Aktuator) verändert wird, falls der durch die Flüssigkeit entstehende Druck zu stark vom vorgegebenen Wert (Flow Setpoint) abweicht. Anschließend beobachtet es die Reaktion des Objekts mittels

[5] Hermann Kopetz, Real-Time Systems, Kluwer Academic Publishers, 2. Ausgabe 1998, Seite 22

Auslesen und Interpretation der vom Sensor ermittelten Informationen. Stellt sich die gewünschte Reaktion ein, so müssen keine weiteren Schritte unternommen werden, falls nicht, kann das Computersystem erneut eine Neueinstellung einleiten.

II. 1 Kontrollierendes Computersteuersystem:

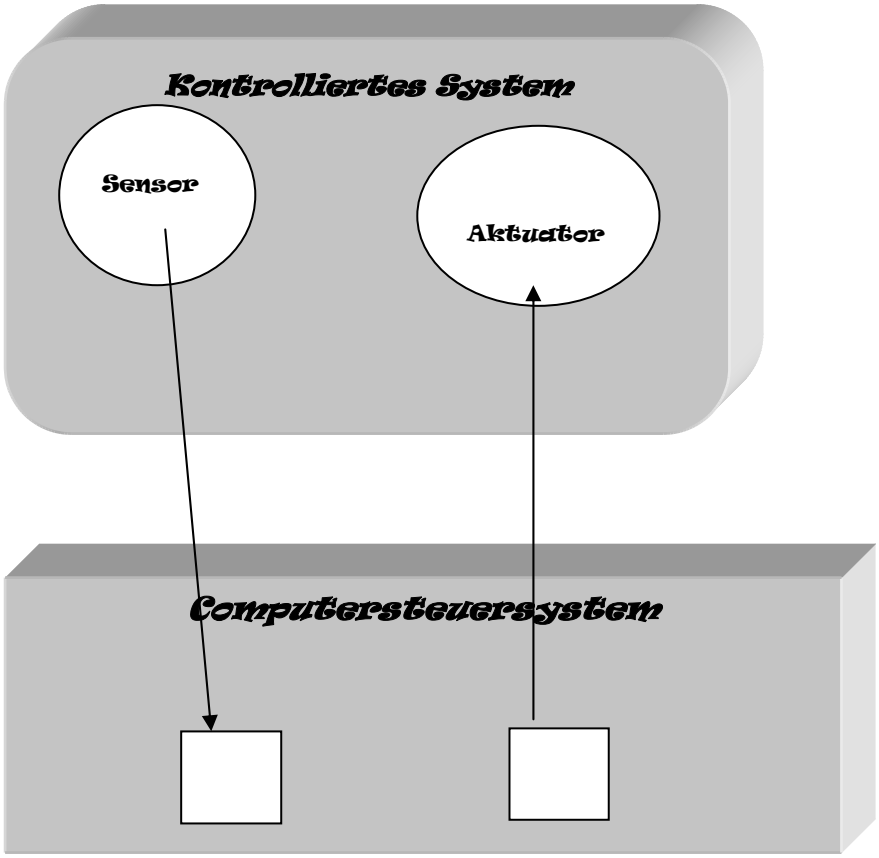


Abb.: Kontrollierendes Computersteuersystem

Annahmen:

- Verteiltes Computersystem:
  - Mehrere eigenständige Rechner (= *Knoten*)
  - Kommunikation via LAN; Kommunikation besteht lediglich aus Nachrichtenaustausch
  - Knoten sind *fail-silent*, d.h. entweder stellt ein Prozess ein korrektes Ergebnis zur Verfügung oder überhaupt keins, er verhält sich also still, falls er seine Dienste nicht korrekt anbieten kann.
  - fehlerhafte Nachrichten können entdeckt und ggf. verworfen werden
  - Spezielle Knoten: *Interface-Knoten*, unterstützen Verbindung zu „intelligenten Instrumenten“, z.B. Sensoren und Aktuatoren, die als Interface zum kontrollierten Objekt angesehen werden.
  - Alle Knoten besitzen eine eigene Uhr bzw. einen Taktgeber, die synchronisiert laufen; dies hat zur Folge, dass eine globale Zeitbasis geschaffen wird, die eine ausreichende Granularität (= Detaillierungsgrad) für alle Klienten in den verschiedenen Knoten des verteilten Systems bietet.

Das verteilte Computersystem besteht aus einer Menge miteinander kommunizierender Teilsysteme. Es lässt sich als Netz einzelner Teilsysteme interpretieren. Die Kanten können wahlweise durch spezielle Hardware oder ein LAN realisiert sein. Wichtig ist, dass alle Uhren synchronisiert sind, um Fehler bei den zeitabhängigen Entscheidungen zu verhindern.

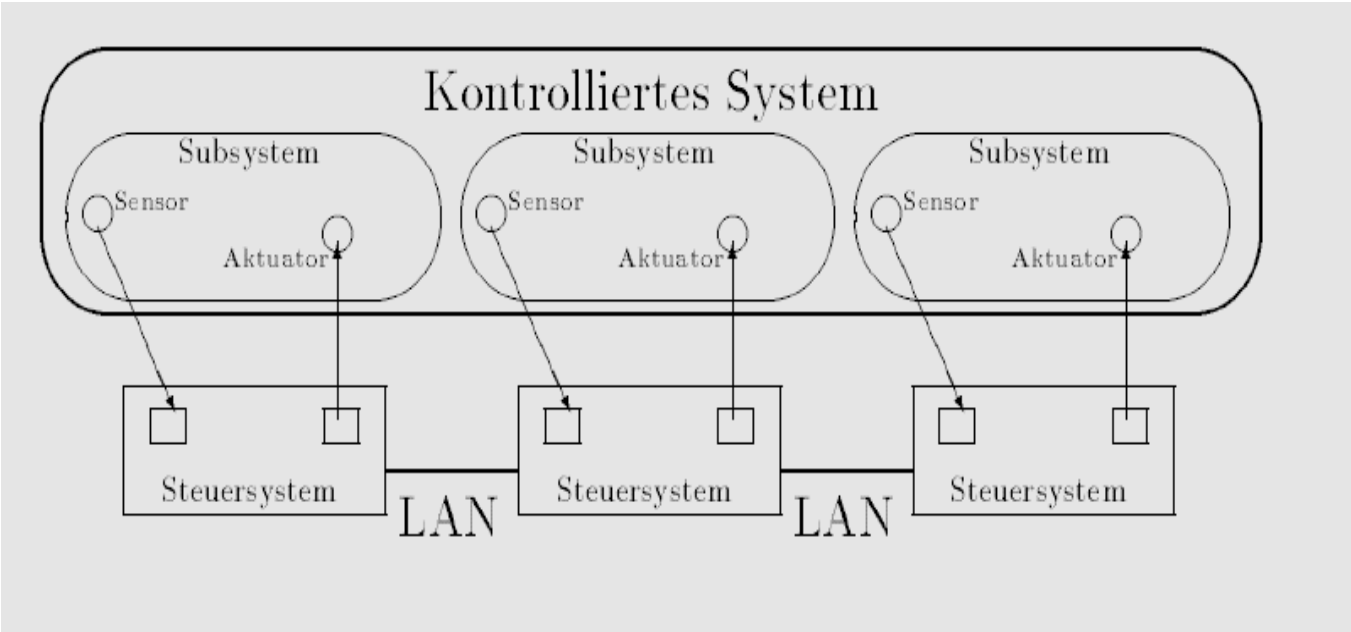


Abb.: Verteiltes Computersteuersystem[6]

II.2 Verhalten der Echtzeitapplikation:

Ein kontrolliertes Objekt, z.B. ein Auto, ändert seinen Zustand im Zeitablauf.

Das Verhalten wird mit Hilfe einer Menge von Prozessen modelliert, die die nötigen Berechnungen anhand von zeitabhängigen Zustandsvariablen durchführen. Halten wir die Zeit nun an – wir frieren sie ein – so können wir den momentanen Zustand eines Objekts beschreiben, indem wir die Werte der zugehörigen Zustandsvariablen notieren. Mögliche Zustandsvariablen des kontrollierten Autos sind z.B. die Position, die Geschwindigkeit, die Schalterstellungen des Scheibenwischers und die Zylinderstellung. Normalerweise sind nicht immer alle Variablen von Interesse, sondern nur die für den jeweiligen Prozess signifikanten. Solch eine ***signifikante, zeitabhängige Zustandsvariable*** wird ***RT-Entity*** genannt und kann entweder extern in der Umgebung (im kontrollierten Objekt) oder intern (im Computersystem) angesiedelt sein. Jede RT-Entity hat *statische* Attribute, die sich während ihrer Lebenszeit nicht ändern, z.B. Name, Typ, Wertebereich, maximale Änderungs-/ Schwankungsrate. Sie hat aber auch *dynamische* Attribute; diese können sich

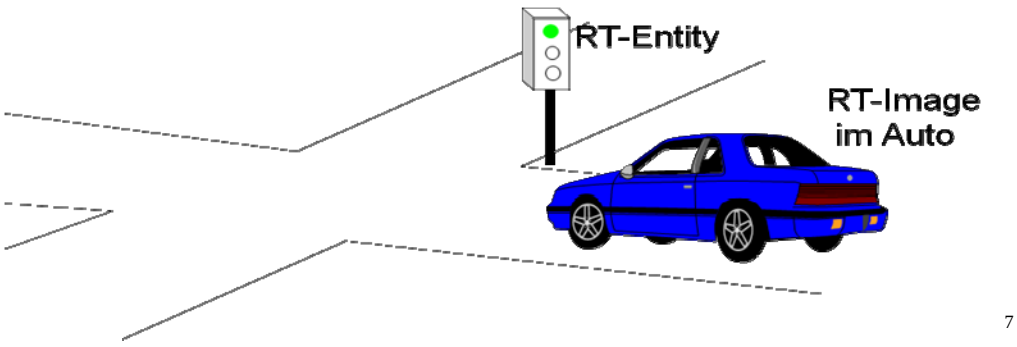
[6] M. Oetken, Seminar Echtzeitbetriebssystemkerne, 1997

im Laufe der Zeit ändern, so z.B. Wertebelegung oder spezifische Änderungsrate der Werte zu einem bestimmten Zeitpunkt.

Jede RT-Entity ist im Einflussbereich (*sphere of control*, ***SOC***) eines Subsystems, welches die jeweiligen Werte zu einem bestimmten Zeitpunkt festsetzt. Dieses Subsystem kann z.B. im kontrollierten Objekt oder in einem bestimmten Knoten des verteilten Systems beheimatet sein. Außerhalb dieses Einflussbereichs können die RT-Entities nur betrachtet/observiert, aber nicht modifiziert werden.

Beispiel:
Die aktuelle Kolbenposition in einem Zylinder (= RT-Entity) ist im Einflussbereich eines Autos. Außerhalb eines Autos kann diese Position nur beobachtet werden.

Eine funktionale Anforderung an Echtzeitsysteme ist also die Möglichkeit des Beobachtens und das Sammeln dieser Beobachtungen der RT-Entities innerhalb eines kontrollierten Objekts. Eine solche Beobachtung (*observation*) einer RT-Entity wird durch ein ***RT-Image*** im Computersystem repräsentiert. Ein solches RT-Image kann entweder in einem Knoten des Computersystems oder in seiner Umgebung (z.B. in einem Aktuator) gespeichert werden.
Da der Zustand eines kontrollierten Objekts in Abhängigkeit der Zeit definiert wird, ist die zeitliche Gültigkeit eines gegebenen RT-Images limitiert. Die Gültigkeitsdauer ist abhängig von der Dynamik des kontrollierten Objekts: ändert es seinen Zustand häufig, so besitzt das zugehörige RT-Image nur eine kurze Gültigkeit. Die Dynamik bestimmt somit das Intervall der zeitlichen Gültigkeit (*temporal accuracy interval*): RT-Images werden im Zeitablauf ungültig und müssen durch Aktuellere ersetzt werden.



[7]

Das RT-Image der Observation wird dann zeitlich ungültig, wenn die Information „Die Ampel ist grün“ außerhalb ihres Gültigkeitsintervall verwendet wird, z.B. wenn ein Auto die Kreuzung befährt, nachdem die Stellung auf gelb umgeschaltet wurde. Die Folge kann ein Unfall sein. Folglich ist eine obere Schranke des Intervalls durch die Dauer der Grünphase gegeben.

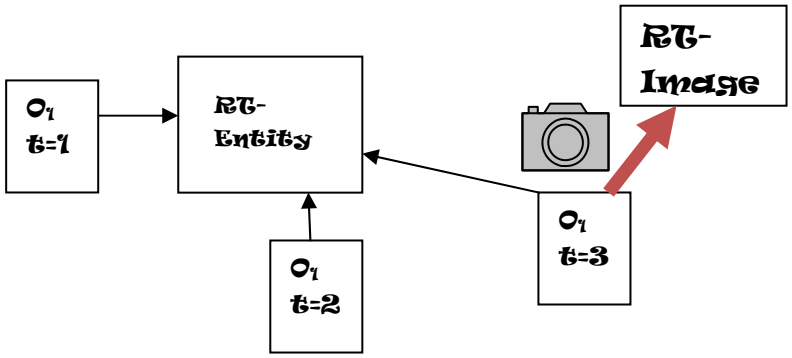
[7] Hermann Kopetz, Real-Time Systems, Kluwer Academic Publishers, 2. Ausgabe 1998, Seite 3

Weiteres Beispiel:

Die grüne Ampel wird beobachtet. Wie lange ist die Aussage: „Die Ampel ist nicht rot“ zeitlich gültig? Antwort: Für die Dauer der Gelb-Phase!

Die Menge der zeitlich gültigen RT-Images wird ***RT-Database*** genannt. Sie muss jedes Mal aktualisiert werden, wenn eine RT-Entity ihren Wert ändert. Diese Updates können periodisch durch das Fortschreiten der Zeit (*time-triggered*) oder auch unmittelbar nach einer Zustandsänderung (*event-triggered*) ausgelöst werden.

Die den RT-Images zugrundeliegenden Beobachtungen (Observationen) – ein RT-Image ist quasi ein Foto einer momentan gültigen RT-Entity-Observation; RT-Images basieren auf Observationen - geben für eine RT-Entity (Entity-Name) und einen Zeitpunkt (point of observation t_{obs}) deren Zustand (Wert) an. Die Observations sind atomare Datenstrukturen und werden in Tupelnotation dargestellt: O = <Name, t_{obs}, Wert>



Beispiel:

Observation O = <Ampelstellung, $t_{aktuell}$, grün>. Die Ampelstellung ist auf grün geschaltet.

Technisch werden die Observationen mittels eines lokalen Mikroprozessors (Feldbusknoten), der mit einem Sensor verknüpft ist, erfasst. Sie werden als Einzelnachricht vom erfassenden Knoten an den Rest des Systems verschickt. Das Nachrichtenübermittlungsprinzip berücksichtigt die Atomarität der Observationsdatenstruktur.

- Observationsarten:

  - Untimed Observation
  - Indirect Observation
  - State Observation
  - Event Observation

Während die RT-Images nur eine begrenzte Zeit gültig sind, stellen die Observationen eine stets gültige Aussage über den Zustand einer RT-Entity zu einem bestimmten Zeitpunkt dar.

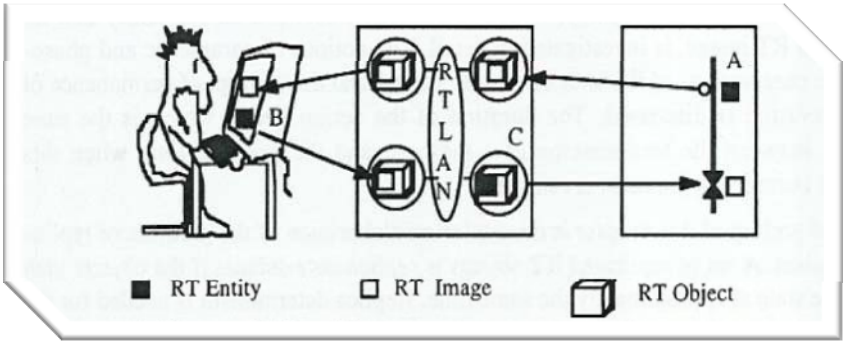
- ***RT-Objekte****:*

RT-Objekte sind als Container für RT-Images oder RT-Entities innerhalb eines Computerknotens in dem kontrollierenden verteilten Computersystem zu verstehen.

Ein RT-Objekt k

- hat eine zugehörige Echtzeituhr, die eine Granularität (bezeichnet den Detaillierungsgrad) t_k besitzt und in Übereinstimmung mit den dynamischen Gegebenheiten der zugehörigen RT-Entity steht.
- aktiviert eine Objektprozedur, sobald die Zeit einen vorgegebenen Wert erreicht

Observationen bzw. die RT-Images der kontrollierten Objekte werden an die RT-Objekte geschickt, dort bearbeitet/manipuliert und gespeichert.



[8]

Aus Sicht eines Systemdesigners kann somit das Verhalten eines Echtzeitsystems/einer Echtzeitanwendung als Menge von RT-Entities, Observationen dieser Entities bzw den daraus resultierenden RT-Images und Operationen auf diesen modelliert werden. Das Echtzeitsystem (das kontrollierende Computersteuersystem) wertet die gewonnenen Informationen aus und überträgt seine Reaktionen mittels der Aktuatoren auf das zu kontrollierende Objekt. Die Zeit, die für die Übertragung der Werte von der RT-Entity zum RT-Objekt benötigt wird, wird *temporal accuracy* genannt, also die Zeitspanne zwischen Erstellung einer Observation/eines RT-Images und dessen Verarbeitung im RT-Objekt. Analog ist *value accuracy* als die Differenz des Wertes des RT-Images zum Zeitpunkt t_{Image} zum tatsächlichen, aktuellen Bearbeitungszeitpunkt $t_{aktuell}$ zu definieren. Ändern sich die beobachteten Werte stetig, dann lässt sich die Genauigkeit in Abhängigkeit des Gradienten der Wertänderung in Relation zur Zeit bestimmen.

Die Dynamik der Wertänderungen einer RT-Entity bestimmt die Anforderungen an die Antwortzeiten eines Computersystems. Ist die Antwortzeit um ein Vielfaches länger als die Zeit zwischen signifikanten Zustandsänderungen, so ist es äußerst schwierig, ein Objekt in Echtzeit zu kontrollieren. Nur wenn die Geschwindigkeit des Computersteuersystems mit der Dynamik des zu kontrollierenden Objekts übereinstimmt, ist eine Echtzeitkontrolle möglich.

III. Einführung zeit- und ereignisgesteuerte Echtzeitsysteme

Abhängig von den Auslösemechanismen für den Start der Kommunikations- und der Verarbeitungsaktivitäten in jedem Knoten des Computersystems existieren zwei unterschiedliche Ansätze zum Design von Echtzeitsystemen /-anwendungen:

[8] Hermann Kopetz, Real-Time Systems, Kluwer Academic Publishers, 2. Ausgabe 1998, Seite 98

1.) Ereignisgesteuerter (event-triggered) Ansatz:

Alle Kommunikations- und Verarbeitungsaktivitäten werden gestartet, sobald eine *signifikante Zustandsänderung* (z.B. ein Ereignis in einer RT-Entity oder dem zugehörigen RT-Objekt) bemerkt wird. Ein Ereignis *(event)* ist das Geschehen zu einem bestimmten Punkt in der Echtzeit.

2.) Zeitgesteuerter (time-triggered) Ansatz:

Alle Kommunikations- und Verarbeitungsaktivitäten werden periodisch zu bestimmten Zeitpunkten gestartet.

Beispiel:
Der Entwurf eines computergesteuerten Aufzugsystems in einem Hochhaus kann sowohl zeit- als auch ereignisgesteuert sein:
In einer ereignisgesteuerten Implementierung verursacht jeder Knopfdruck einen Interrupt im Computersystem und aktiviert einen Task, um die Aufzugssteuerung neu zu planen und den Fahrgastwusch zu erfüllen.
In einer zeitgesteuerten Implementierung wird jeder Knopfdruck in einem Zwischenspeicher für jedes Knopfelement gesammelt. Die Speicherelemente werden periodisch abgetastet. Ein solcher Abtastzyklus kann z.B. 500 msek betragen. Anschließend wird der Zwischenspeicher zurückgesetzt. Nach einem solchen Zyklus wird die Aufzugsplanung gestartet und ein neuer Fahrplan berechnet, um die Fahrgastwünsche zu erfüllen.
Sollte ein Fahrgast ungeduldig mehrmals den Knopf drücken, so wird dies in den einzelnen Implementierungen unterschiedlich gehandhabt:
Im ereignisgesteuerten Ansatz wird der erneute Knopfdruck an das Computersystem weitergeleitet, während im zeitgesteuerten Ansatz dieser ignoriert wird, solange der Zwischenspeicher noch nicht zurückgesetzt wurde.

Hauptvorteil des zeitgesteuerten Ansatzes ist die Vorhersagbarkeit, der des ereignisgesteuerten die Flexibilität.

IV. Ereignisgesteuerte Echtzeitsysteme (event-triggered systems):

Ein Echtzeitsystem ist ereignisgesteuert *(event-triggered)*, wenn die Steuerungssignale ausschließlich vom Auftreten von Ereignissen abhängen. Z.B.

- Termination eines Tasks
- Empfang einer Nachricht
- ein externer Interrupt (.d.h. eine asynchron auftretende Anforderung einer spezifischen Task-Aktivierung, ausgelöst durch ein Ereignis außerhalb der aktuellen Berechnung)

In einem rein ereignisgesteuerten System werden alle Aktivitäten durch das Vorkommen signifikanter Ereignisse in RT-Entities oder RT-Objekten initiiert. Solch ein signifikantes Ereignis ist eine Zustandsänderung (in einer RT-Entity oder in einem RT-Objekt), die eine Bearbeitung durch das Computersteuerungssystem erfordert. Somit werden alle Aktivitäten durch ein Ereignis angestoßen. Andere Zustandsänderungen in RT-Entities werden als unsignifikant angesehen und können vernachlässigt werden. Diese Änderungen

des Zustands können mittels Interruptmechanismen implementiert werden, die der CPU das Vorliegen einer Änderung anzeigen.

Unterscheidung signifikanter Ereignisse:

1.) *Vorhersagbare Ereignisse (predictable events; P-events):*
Grundlage der P-Events sind z.B. physikalische Gesetze in Verbindung mit dem aktuellen Systemzustand. Aus einem Ereignis lässt sich schließen, dass bestimmte Ereignisse als Konsequenz folgen werden. Diese Kenntnis ermöglicht es, schon vorher Ressourcen bereitzustellen oder andere Aktivitäten vorzubereiten, um die notwendige Prozessorleistung zeitlich vorzuziehen und somit die Antwortzeiten der Tasks zu den Folgeereignissen, die schon bekannt sind, zu verbessern.

Beispiel: Das Ansteigen des elektrischen Stroms durch einen Draht hat einen Anstieg der Temperatur des Drahtes zur Folge.

Somit können die P-Events antizipiert werden; das Auftreten ist deterministisch, falls bestimmte Voraussetzungen erfüllt sind; infolgedessen können die benötigten Ressourcen für die zukünftige Übermittlung und Berechnung der Reaktionen rechtzeitig reserviert werden.

2.) *Zufällige Ereignisse (chance events; C-events):*
Das Auftreten von C-Events hängt vom Auftreten eines zufälligen Prozesses ab; dieser ist also nicht deterministisch vorhersagbar. Um das Verhalten einer großen Anzahl solcher C-Prozesse zu beschreiben, kann man sich statistischer Hilfsmittel bedienen. Ein Beispiel für einen solchen C-Event ist das Auftreten eines Fehlers, entweder im kontrollierten Objekt oder im Computersteuersystem, der eine entscheidende Zustandsänderung in mehreren der RT-Entities des zu kontrollierenden Objekts zur Folge hat. Da ein Computersteuersystem nur eine endliche Berechnungskapazität besitzt, ist es nicht möglich, diese Ereignisse unmittelbar, ohne Verzögerung zu berücksichtigen. Also müssen Mechanismen für die Zwischenspeicherung solcher „Überflutungen“ und die Einschränkungen des Informationsflusses vom kontrollierten Objekt zum Computersteuersystem bereitgestellt werden.

IV.1 Flusskontrolle:

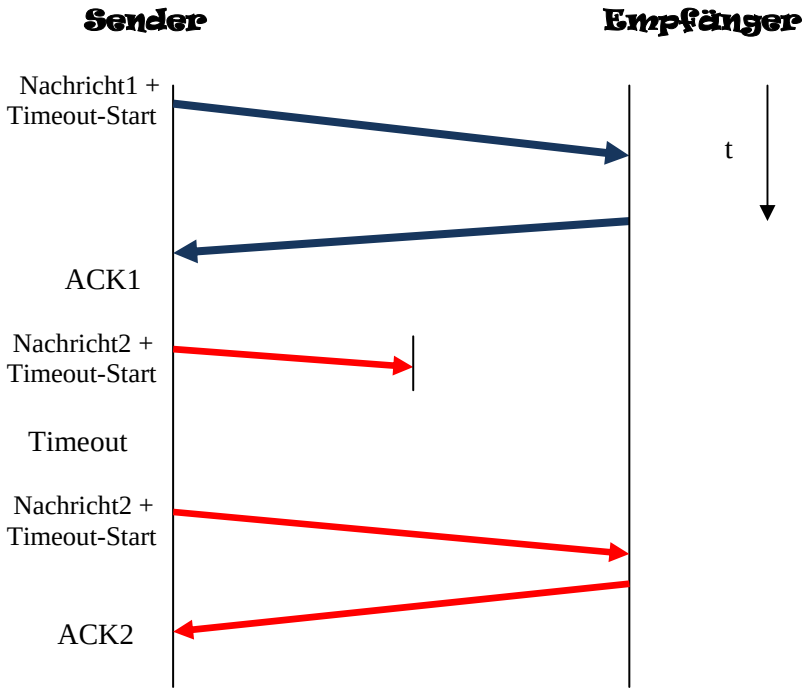
Die Flusskontrolle beschäftigt sich mit der Synchronisation der Sende- und Empfangsgeschwindigkeit beim Versand und Empfang von Informationen. Ziel ist es, dass der Empfänger dem Sender folgen kann.

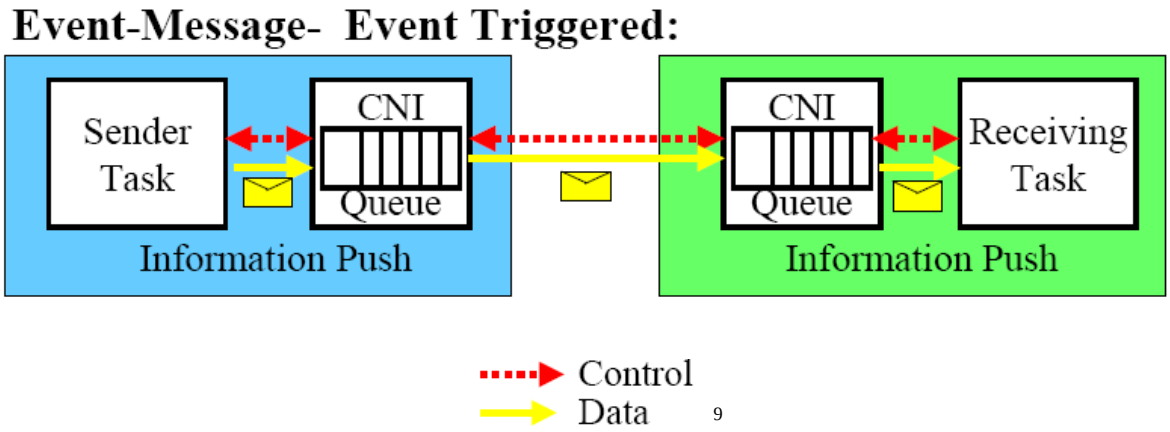
Da das zu kontrollierende Objekt nicht im direkten Einflussbereich (SOC) des Computersteuersystems liegt, gibt es keine Möglichkeit, das Auftreten von C-Events zu limitieren, falls das Computersteuersystem hinterherhinkt. Folglich müssen Vorkehrungen zur Pufferung der Überflutungen im Interface zwischen Objekt und Computersystem getroffen werden. Es existieren mehrere Möglichkeiten, den Informationsfluss im Interface zu beschränken, so z.B. Zwischenpufferung der Ereignisse in Hardware/Software.

In einem ereignisgesteuerten Echtzeitsystem muss ein ***expliziter*** Flusskontrollmechanismus mit Puffer zwischen einem sendenden und einem empfangenden Knoten implementiert werden. Die Zeitspanne, die eine Ereignisnachricht in einem Puffer auf ihre Bearbeitung warten muss, reduziert die *temporal accuracy* der Observationen/der RT-Images und muss streng limitiert werden. Ein großes Problem besteht darin, die richtige Puffergröße zu wählen, da auch hier auf Wahrscheinlichkeitsberechnungen zurückgegriffen werden muss.
Um Flusskontrolle zu erreichen, werden *Kommunikationsprotokolle* eingesetzt.

IV.2 Kommunikationsprotokolle in ereignisgesteuerten Echtzeitsystemen als Mittel zur Flusskontrolle:

Nur der Sender kennt den Zeitpunkt zu dem eine Nachricht gesendet werden muss, somit wird die Fehlererkennung in der Kommunikation mittels eines *senderseitigen Timeouts* erreicht: Der Sender wartet auf eine Bestätigungsnachricht des Empfängers, ob er die Nachricht erhalten hat. Erhält er diese Bestätigung nicht, so wird nach dem Timeout die Nachricht erneut versendet (*Positive Acknowledge or Retransmission*). In diesen Protokollen wird k-Fehlertoleranz durch k-maliges Wiederversenden der verlorengegangenen Nachricht erreicht. Das *PAR-Protokoll* hat allerdings den Nachteil, dass eine Überlastung (*thrashing*) möglich ist: Durch das erneute und gegebenenfalls mehrmalige Senden einer verlorengegangenen Nachricht, kann es vorkommen, dass das Kommunikationssystem die zu bewältigenden Aufgaben nicht vollständig abarbeiten kann und langsamer wird. Die einzige Möglichkeit, thrashing in einem ereignisgesteuerten System zu vermeiden, ist, die Ressourcenanforderungen des Systems kontinuierlich zu überwachen und es dem Empfänger zu ermöglichen, den Datenfluss vom Sender aus zu kontrollieren, sobald eine Abnahme des Durchsatzes festgestellt wird (*back-pressure flow control*).





[9]

Um einen Stau im Falle eines gleichzeitigen Ereignisauftretens zu vermeiden, müssen dafür ebenfalls explizite Maßnahmen in der Netzwerkschicht des Kommunikationssystems implementiert werden.

Um eine beständige Ordnung der Ereignisse auf Empfängerseite beizubehalten, sind atomare Broadcastprotokolle als grundlegende Kommunikationssysteme vorgeschlagen worden. Diese Kommunikationsart zeichnet sich dadurch aus, dass alle Nachrichten ausgeliefert werden müssen, es erfolgt eine Auslieferung in gleicher Reihenfolge (totale Ordnung). Diese führen zu einer *asynchronen Kommunikation*, da sie nicht-blockierend ist - es wird kein Acknowledge gesendet - eine Nachricht wird an alle oder an keinen gesendet. Atomic Broadcast stellt sicher, dass Prozesse Nachrichten zuverlässig verschicken, so dass sie über die Nachrichten, die sie verschicken sowie über deren Reihenfolge, einig sind. Das *Non-Blocking Atomic Commitment-Protokoll* verlangt von den Teilnehmern, dass sie sich auf den Ausgang einer Transaktion einigen: entweder Commit oder Abort, auch wenn Teilnehmer ausfallen können. Die zeitliche Ungewissheit dieser asynchronen Protokolle hat allerdings einen nachteiligen Einfluss auf die zeitliche Genauigkeit der Information (*temporal accuracy*), da die Differenz zwischen maximaler und minimaler Latenz beträchtlich sein kann.

IV.3 Scheduling Strategie:

Betriebssystemdienste für ereignisgesteuerte Systeme sind nachfrageorientiert/bedarfsgesteuert – sie reagieren auf (plötzlich) auftretende Ereignisse. Folglich benötigen sie eine dynamische (*online*) Scheduling Strategie. Hierbei wird die Organisation der Tasks während der Laufzeit von spezieller Hardware erledigt. Das

[9] Hermann Kopetz, Smart Transducers

dynamische Scheduling ist flexibel, da es sich an das entstehende Szenario der auftretenden Ereignisse anpasst.
Es existieren mehrere Algorithmen für das dynamische Scheduling-Problem, sowohl für periodische (in gewissen Abständen wiederkehrend) als auch für sporadische (unregelmäßig oder einmalig auftretende) Tasks:

1.) ***Earliest Deadline First*** (EDF) – Bei diesem Verfahren werden Prioritäten für auftretende Tasks dynamisch zur Laufzeit vergeben und können sich häufig ändern.
Funktionsweise:
   a) Alle zu einem betrachteten Zeitpunkt bereitstehende Tasks werden nach ihrer aufsteigenden Deadline geordnet.
   b) Es wird immer genau der Task ausgeführt, dessen Deadline als nächstes kommt, und damit die höchste Priorität hat.

   Es werden stets die Zeitpunkte für das Scheduling betrachtet, zu denen ein Ereignis stattgefunden hat (neu auftretender Task) oder ein gerade noch aktiver Task beendet wird. EDF ist dabei sehr flexibel: Es kann sowohl für präemptives (d.h. unterbrechbares) Scheduling als auch für nicht-präemptives Scheduling verwendet werden. Außerdem kann es in sporadischen und periodischen Szenarien eingesetzt werden

2.) ***Least Laxity First*** (LLF) – Dieses Verfahren ist dem EDF sehr ähnlich, allerdings wird zur Auswahl des nächsten auszuführenden Tasks nicht nur die Deadline betrachtet, sondern zusätzlich die Zeit, die der Task bis zur Erfüllung seiner Aufgabe noch rechnen muss. Die „*Laxity*“ ist die Zeit, die einem Prozess bis zu seiner Deadline noch übrigbleiben würde, wenn er ab sofort bis zu seiner Beendigung den gesamten Prozessor zur Verfügung hätte. Der Prozess-Scheduler wählt diejenigen Prozesse zuerst, die die geringste Laxity haben.

3.) In der Praxis der ereignisgesteuerten Systeme greift man vor allem auf das statische Prioritätsscheduling zurück. Statisch bezieht sich dabei auf die Prioritäten: es ist für den Scheduling-Algorithmus wichtig festzulegen, wann ein Prozess unterbrochen werden darf. Wenn jeder Prozess beliebig unterbrochen werden dürfte, könnte dies dazu führen, dass immer wieder ein Prozess einen anderen blockiert. Zur Lösung dieses Problems werden Prioritäten eingeführt: wichtige zeitkritische Prozesse bekommen eine *hohe*, unkritische eine *niedrige* Priorität. Nun darf ein Prozess nur von einem anderen mit höherer Priorität unterbrochen werden. Werden die Prioritäten für alle Prozesse einmalig und für die gesamte Lebensdauer festgelegt, dann heißen sie statisch.

Ein Beispiel für das statische Prioritätsscheduling ist das ***Deadline Monotonic*** Verfahren (DM).

Annahmen:
1.) Tasks können zu jedem Zeitpunkt unterbrochen werden
2.) Der Overhead für Taskwechsel ist vernachlässigbar gering
3.) Notwendige Ressourcen sind unbegrenzt vorhanden
4.) Alle Tasks sind unabhängig
5.) Die Last aller Prozesse zusammen muss <= 1 sein

Für periodische Tasks:

6.1) Alle Tasks haben Deadlines kleiner oder gleich ihrer Perioden

Für sporadische Taskse:

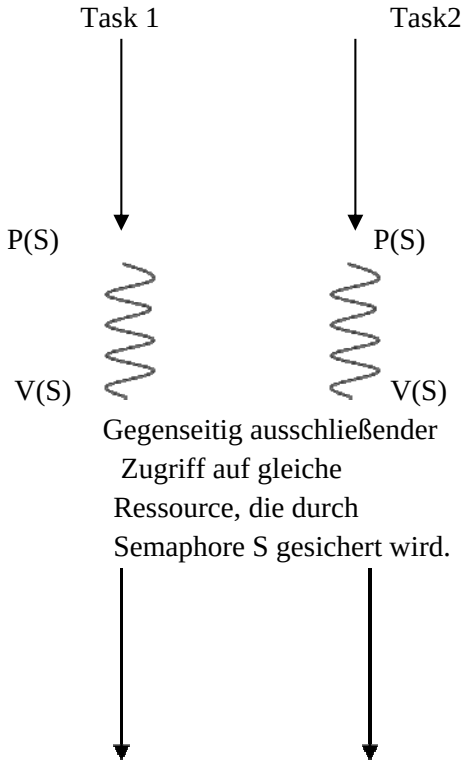
6.2) Alle Tasks haben Deadlines kleiner oder gleich ihrer Minimal-Inter-Arrival-Zeiten (Minimum zwischen zwei Ankunftszeiten desselben Tasks)

Es wird stets der Task mit der höchsten Priorität ausgeführt. Die Zuweisung der Prioritäten orientiert sich hierbei an den Deadlines: Je kürzer die Deadline umso höher wird seine Priorität eingestuft.

Das Deadline Monotonic Verfahren ist eine Verallgemeinerung des ***Rate Monotonic*** Verfahrens (RM). Im Spezialfall, dass für *alle* Tasks relative Deadline und Periodenlänge gleich sind, gilt DMS = RMS.

Allerdings kann das Problem, eine ausführbare Reihenfolge/einen Schedule in der Menge der Echtzeittasks zu finden, NP-hart (NP-schwer) sein, da diese Tasks i.d.R. durch Abhängigkeitsrelationen (gegenseitiger Ausschluss, Ressourcenbeschränkungen) untereinander eingeschränkt sind.

Beispiel Ressourcenabhängigkeit:
Zwei Tasks versuchen auf die gleiche Ressource zuzugreifen und in ihren kritischen Abschnitt einzutreten:
Der kritische Abschnitt ist ein Abschnitt, für den exklusiver Zugriff auf eine Ressource garantiert werden muss. Dies kann mittels einer Semaphore S erreicht werden.



P(S) überprüft, ob die Ressource verfügbar ist; ist sie frei, so wird S auf „used“ gesetzt; und dem anfragenden Task zugewiesen; dies ist eine atomare Aktion, die nicht unterbrochen werden darf. Die Ressource ist ausschließlich für den einen Task reserviert. Die Ressource wird mittels V(S) wieder freigegeben.
Task 1 sichert sich nun die Ressource zuerst. Möchte nun Task 2 ebenfalls auf die Ressource zugreifen, so muss er warten, bis sie von Task 1 freigegeben wurde. Hat nun Task 2 eine kürzere Deadline als Task 1, so ist es sehr schwierig, einen ausführbaren Schedule in Echtzeit zu finden (besonders, wenn dies nicht die einzige Abhängigkeitsrelation ist).

IV.4 Replica Determinism:

In der Regel müssen Echtzeitsysteme *fehlertolerant* sein, sie müssen z.B. rechtzeitigen Service trotz Auftretens von Fehlern gewährleisten. Fehlertoleranz ist eine Systemeigenschaft, die bedeutet, dass Fehlfunktionen so abgefangen werden, dass das gewünschte Systemverhalten trotzdem gesichert bleibt – Fehlfunktionen führen nicht zu Fehlverhalten. In verteilten Echtzeitsystemen wird die Fehlertoleranz durch aktive Redundanz (*active redundancy*) erreicht: liegt in einem Knoten ein Fehler vor, so übernimmt sein Replikat, der ständig aktiviert ist, die Aufgabe. Diese Redundanz basiert auf dem Determinismus von Replikaten (*replica determinism*) und bedeutet, dass korrekt replizierte Systeme sich komplett identisch verhalten. Sie geben die gleiche Ausgabe in derselben Reihenfolge und dies innerhalb des gleichen Zeitintervalls. Dieses identische Verhalten ist dann notwendig, wenn zur *Fehlererkennung* und *Fehlertoleranz* ein einzelnes System repliziert wird.

Das Problem bei ereignisgesteuerter Kommunikation ist, dass es Quittierungen, Timeouts und wiederholte Übermittlungen von Nachrichten gibt. Hierbei kann sich das Verhalten der Replikate schnell unterscheiden, wenn z.B. ein Replikat die Quittung noch innerhalb des Timeouts bekommt und ein anderes Replikat nicht. Soll ein verteiltes System mit Replikaten eine ereignisgesteuerte Kommunikation unterstützen, müssen sich die Replikate untereinander abstimmen, um sich identisch zu verhalten (*Zustandssynchronität*). Verhalten sich die Replikaten deterministisch, ist dies z.B. durch *Triple Modular Redundancy with voting*, falls die Annahme der stillen Fehler (fail-silent) nicht getroffen wurde, möglich; falls dies doch unterstützt wird, besteht die Möglichkeit, Fehlertoleranz mittels *duplex fail-silent selfchecking nodes* zu erreichen.

Zustandssynchronität ist in asychronen, verteilten Echtzeitsystemen, die auf dem dynamischen unterbrechenden Schedulingverfahren basieren, allerdings schwer zu erlangen. Deshalb wurden andere Modelle wie etwa das *leader-follower Modell* von DELTA4 entwickelt. In diesem Modell trifft ein ausgezeichneter Führer (*leader*) alle (nichtdeterministischen) Entscheidungen und zwingt seinen Nachfolger (*follower*), die gleiche Entscheidung zu treffen. Diese Methode erfordert allerdings zusätzliche Kommunikationsaktivitäten zwischen dem leader und seinem follower.

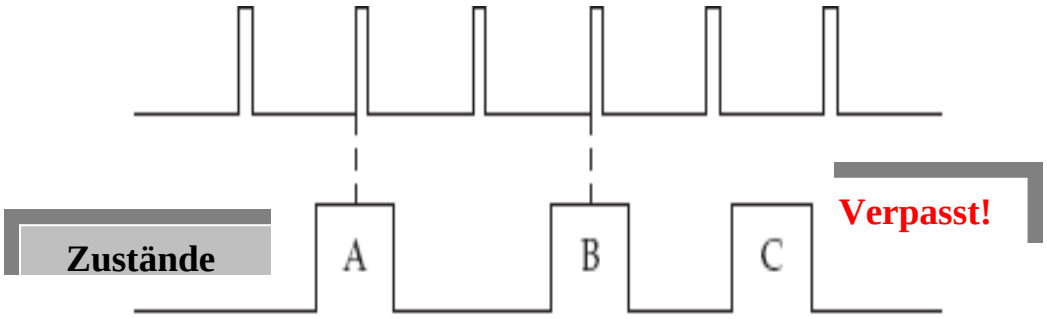
V. ***Zeitgesteuerte Systeme (time-triggered systems):***

Ein Echtzeitsystem ist zeitgesteuert *(time-triggered)*, wenn die Steuerungssignale wie

- ❖ Senden und Empfangen von Nachrichten
- ❖ Erkennung einer externen Zustandsänderung

ausschließlich vom Fortschritt einer (globalen) Zeitnotation ausgehen.

Während in einem ereignisgesteuerten System die Information über auftretende Ereignisse (= Geschehen zu einem bestimmten Zeitpunkt) schnellstmöglich verbreitet werden muss, ist die schnelle Übermittlung der Zustandswerte der RT-Entities das Hauptaugenmerk in zeitgesteuerten Systemen. Ein Zustand *(state)* ist eine Tatsache, die für einen bestimmten Zeitraum besteht. Diese Zustandsübermittlung wird periodisch vorgenommen, wobei diese Perioden/die Zeitpunkte auf das dynamische Verhalten der RT-Entities abgestimmt werden. Die Zeitpunkte der Übermittlung sind Gitterpunkte im sogenannten *observation grid/observation lattice* (Beobachtungsgitter), das eine festgelegte Granularität g_0 besitzt. Die RT-Entities werden praktisch nur zu diesen Zeiten beobachtet. Deshalb müssen die Abstände dieser Punkte vorher eng genug gewählt werden, um zu verhindern, dass wichtige Änderungen zu lange unentdeckt bleiben. Allerdings müssen auch wichtige, kurzlebige Ereignisse beachtet werden (z.B. ein Druckknopf, der nur sehr kurz gedrückt wird, aber nur einmal pro Minute abgefragt wird).



[10]

Solch ein Szenario ist folglich nicht wünschenswert.

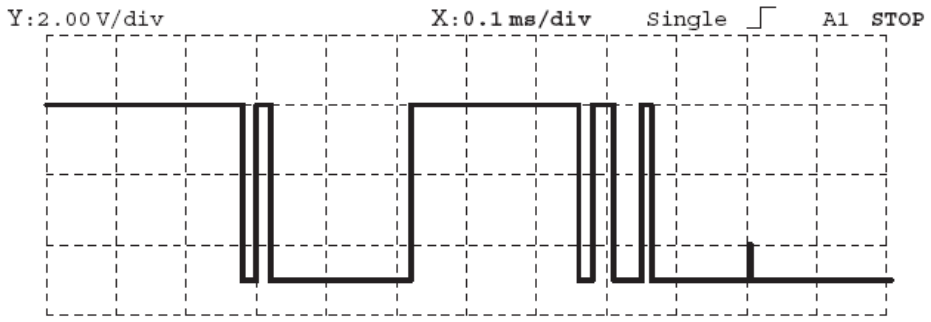
Man kann jedoch die Logik der RT-Entity (z.B. ein Sensor) so konstruieren, dass diese kurzlebigen Signale bis zur nächsten Abfrage zwischengespeichert werden (wichtig für die Flusskontrolle, s.u.).

Beispiel: Ein Druckknopf ist ein Ereignissensor; die interne Logik dieses Sensors muss sicherstellen, dass der Zustand „Knopf gedrückt“ für ein Intervall, das größer als das für die RT-Entity „Druckknopf“ vorgesehene Beobachtungsgitter (observation grid) ist, als wahr angesehen wird, so dass der aktuelle Zustand korrekt wiedergegeben wird.

Allerdings ist es auch nicht sinnvoll, die Abstände zu eng zu wählen. Es kann vorkommen, dass sich ein Signal/Zustand erst stabilisieren muss, bevor es eindeutig interpretiert werden kann.

[10] R. Williams, Real-time Systems Development, Elsevier 2006, Seite 26

Beispiel: Verhalten eines zu schließenden Mikroschalters, am Oszilloskop beobachtet



[11]

Wird ein Mikroschalter, wie im obigen Beispiel angedeutet, geschlossen, so schwingt die Ausgangsspannung für eine Zeitdauer von bis zu 1ms, bevor der Wert endgültig als 0 angesehen werden kann.

Unabhängig von der aktuellen Aktivität in der beobachteten RT-Entity, wird deren Zustand in konstanten Abständen überprüft.

Obwohl der Zeitpunkt einer Zustandsänderung nicht beeinflussbar ist, sind die zeitgesteuerten Systeme von Natur aus eher vorhersehbar als die ereignisgesteuerten Systeme, da sie eine Regelmäßigkeit durch das Setzen des Beobachtungsgitters (observation grid) erzwingen – eine Zustandsänderung kann, wenn überhaupt, nur innerhalb dieser Überwachungssequenz erfolgen.

V.1 Flusskontrolle:

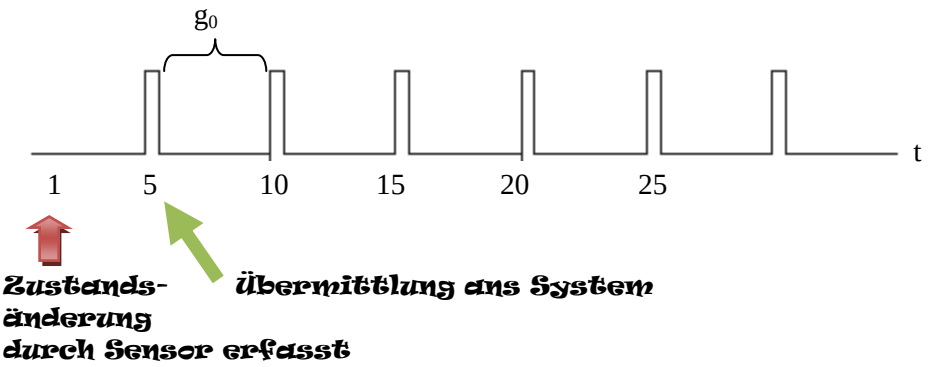
Die Flusskontrolle erfolgt in zeitgesteuerten Systemen im Gegensatz zu ereignisgesteuerten Systemen ***implizit***:
Schon während der Systementwicklung werden angemessene Beobachtungs-, Nachrichten- und Prozessaktivierungsanteile für die verschiedenen RT-Entities festgelegt, welche mit der jeweiligen Verhaltensdynamik in Einklang stehen. Es muss bereits zum Entwicklungszeitpunkt sichergestellt werden, dass alle Empfängerprozesse ihre Aufgaben bewältigen können. Sollten die RT-Entities schneller als ursprünglich vorgesehen ihre Zustände ändern, so ist es möglich, dass einige nur sehr kurzlebige, eventuell aber auch nicht ganz so wichtige Zwischenzustände nicht erfasst werden können und die zugehörigen Observationen/RT-Images verloren gehen können – der Sensor muss dann entscheiden, welcher Zustand als relevant ausgewählt wird.
Vorteilhaft ist, dass selbst zu Spitzenzeiten die Anzahl der Nachrichten pro Zeiteinheit konstant ist.
Die implizite Flusskontrolle funktioniert nur, wenn das Instrumentarium (Sensoren, Aktuatoren) die Zustandssicht (*state view*) unterstützen, z.B. muss die von einem Sensor erfasste Information genau den augenblicklichen Zustand und nicht die Zustandsänderung der RT-Entity wiedergeben, also nicht die absolute Änderung der Werte zweier Zustände. Nötigenfalls muss ein lokaler Mikrocontroller die initiale Ereignisinformation, die durch einen Sensor erfasst wurde, in ihr Zustandsäquivalent umformen.

Die Mitteilung eines Sensors an das kontrollierende Computersystem muss gegebenenfalls solange im Sensor gespeichert werden, bis sie abgefragt werden kann. Diese

[11] R. Williams, Real-time Systems Development, Elsevier 2006, Seite 25

Zwischenspeicherung in den Sensoren verhindert das Überfluten mit neuen Informationen (vgl. ereignisgesteuerte Systeme).

Durch die vorgegebene Granularität g_0 kommt es vor, dass die Benachrichtigung über Zustandsänderungen stark verzögert werden kann, z.B. wenn eine Änderung „weit“ vor einem „Abtastzeitpunkt“ liegt. Somit limitiert die festgelegte Granularität die zeitliche Präzision der abgefragten Änderungen. Ist diese Präzision für den Verwendungszweck nicht ausreichend, z.B. wenn ein Prozess eine sehr feinkörnige Granularität benötigt, das Beobachtungsgitter (*observation grid*) aber eher grobkörnig ist, so ist es zusätzlich möglich, dass der Sensor Informationen über den Zeitpunkt der Änderung relativ zum letzten Abfragezeitpunkt - z.B. 1ms nach dem letzten Abfragezeitpunkt- anfügt (*timed observation*).



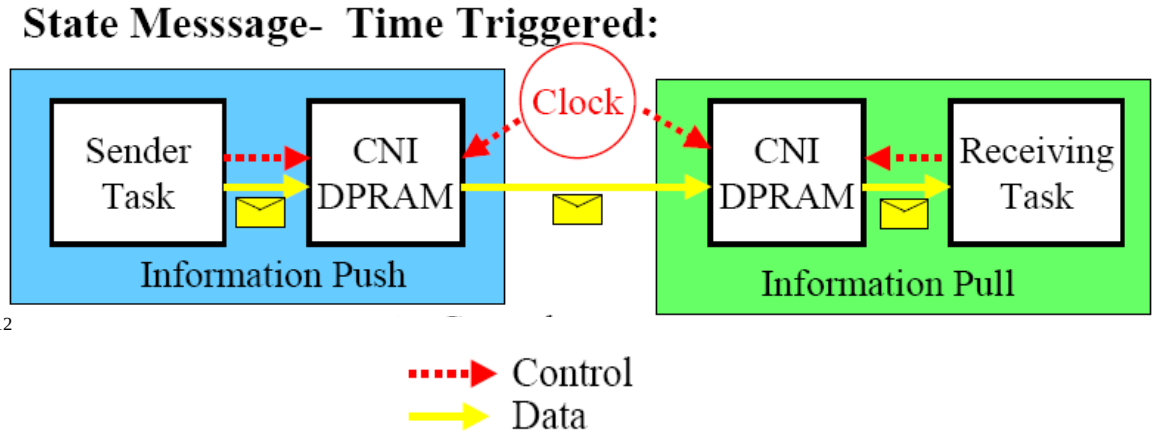
Hierbei erfolgt die Zustandsänderung bereits 4 „Takte“ vor der Abtastung, die Beobachtung ist also schon länger gültig, wird aber erst später an das kontrollierende Computersteuersystem übermittelt.

Timed Observations lösen das Problem der Aufnahmepräzision, aber nicht das Problem der zusätzlichen Verzögerung, die durch die am Anfang festgelegte Granularität der Observationen hervorgerufen wird; sie liefern also „nur“ zusätzliche Zeitinformationen im Datenbereich, dienen aber nicht als Kontrollelement.

V.2 Kommunikationsprotokolle in zeitgesteuerten Echtzeitsystemen als Mittel zur Flusskontrolle:

In zeitgesteuerten Echtzeitsystemen ist es erforderlich, ein Kommunikationsmodell anzuwenden, das die Zustandsinformationen berücksichtigt. Solch ein Modell ist das Zustandsnachrichten-Modell (*state message model*), die zugehörige Nachricht ist eine Zustandsnachricht. Die Semantik einer solchen Nachricht ist mit der Semantik der Programmvariablen in Programmiersprachen verwandt. Eine neue Version einer Zustandsnachricht überschreibt die vorherige. Solche Nachrichten werden zu festgelegten, periodischen Zeitpunkten (den *observation grid points*) erstellt. Diese Zeitpunkte sind von Beginn an allen Kommunikationspartnern bekannt.
Das für die Verbreitung der Zustandsnachrichten verwendete Protokoll ist ein verlässliches broadcast Protokoll (*reliable broadcast protocol*). Dies ist ein Protokoll, dass es dem Empfänger einer Nachricht ermöglicht, verlorengegangene Nachrichten zu entdecken. Da der Sender nur zu bestimmten Zeiten senden darf/kann, besitzt der Empfänger die Kenntnis, wann eine Nachricht an ihn gesendet werden kann. In diesem

Zeitraum erwartet er also eine Nachricht; kommt sie nicht innerhalb dieser Zeitspanne an, so kann er davon ausgehen, dass sie verloren gegangen ist. Da keine positiven Bestätigungen versendet werden, ist der Informationsfluss unidirektional. Auch hier kann Fehlertoleranz durch aktive Redundanz erreicht werden, z.B. durch Verschicken von k Kopien einer Nachricht, ggf. über verschiedene Wege.
Die Erweiterung um zusätzliche Empfänger ist in zeitgesteuerten Systemen möglich, ohne dass das Protokoll oder die Nachrichtenrate im System geändert werden müssen.



[12]

CNI = Communication Network Interface
DPRAM = Dual-ported RAM (=> Zwischenspeicherung)

V.3 Scheduling Strategie:

Betriebssystemdienste für zeitgesteuerte Echtzeitsysteme basieren auf einer Menge von *statischen, vorgegebenen Schedules*. Diese Schedules müssen alle notwendigen Taskabhängigkeiten berücksichtigen, ferner müssen sie eine implizite Synchronisation aller Tasks zur Laufzeit bieten. Die Schedules werden bereits zur Übersetzungszeit/in der Entwicklungsphase erstellt; folglich reicht es aus, zur Laufzeit in einer Tabelle nachzuschauen, welcher Task zum aktuellen Zeitpunkt, dem Aktionsgitter (*action grid*), ausgeführt werden muss. Die Differenz zwischen zwei benachbarten action grids bestimmt die fundamentale Zykluszeit (*basic cycle time*), welche eine untere Schranke für die Reaktion eines zeitgesteuerten Systems darstellt.

In zeitgesteuerten Systemen sind alle Input-/Outputaktivitäten von vorneherein vorgeplant und werden nur durch Abtasten des jeweiligen Sensors/Aktuators zu bestimmten Zeiten angestoßen. Auch der LAN-Zugriff/der Zugriff auf Kommunikationskanäle ist z.B. durch ein synchrones TDMA-Protokoll vorbestimmt.

[12] Hermann Kopetz, Smart Transducers

V.4 Replica Determinism:

Alle Taskwechsel, Modiwechsel und Kommunikationsaktivitäten sind global durch das *action grid* (Zeitpunkt zu dem Tasks ausgeführt werden müssen) synchronisiert. Nichtdeterministische Entscheidungen können somit vermieden werden, wodurch zusätzliche Kommunikation unter den Replikaten nicht benötigt wird.
Die fundamentale Zykluszeit eines zeitgesteuerten Systems liefert eine diskrete Zeitbasis mit einer genau spezifizierten Granularität. Da ein solches System quasi-synchron läuft, kann sowohl *Triple Modular Redundancy* als auch *duplex fail-silent selfchecking nodes* ohne Probleme eingesetzt werden. Ferner ist die Wiedereingliederung reparierter Komponenten problemlos möglich.

VI. Zeit- und ereignisgesteuerte Echtzeitsysteme im Vergleich:

Der Vergleich der unterschiedlichen Ansätze fokussiert die folgenden fünf Aspekte:

1.) Vorhersagbarkeit (*predictability*)
2.) Testbarkeit (*testability*)
3.) Ressourcennutzung (*resource utilization*)
4.) Erweiterbarkeit (*extensibility*)
5.) Annahmen (*assumption coverage*)

VI.1 Vorhersagbarkeit (predictability):

Zeitgesteuerte Systeme verlangen eine gründliche Prüfung der Anforderungen sowie eine genaue Planung der möglichen Abläufe bereits in der Entwicklungsphase des zu implementierenden Systems. Zuallererst ist es notwendig, ein passendes Zeitfenster für die Beobachtungen (*observation grid*), bzw. die nötigen Abtast-/Abfragezeitpunkte festzulegen.

Anschließend muss die maximale Ausführungszeit für alle zeitkritischen Tasks berechnet bzw. geschätzt werden. Nachdem die Tasks den Knoten des Computersteuersystems zugeordnet worden sind, ist es notwendig, die „Kommunikationsslots“ auf dem LAN zu reservieren, sowie passende Ausführungspläne (*execution schedules*) der Tasks *offline* zu erstellen. Aufgrund dieses exakten Planungsaufwands sind detaillierte Entwürfe des zeitlichen Verhaltens eines jeden Tasks verfügbar, was dazu führt, dass das komplette zeitliche Systemverhalten sehr präzise vorhersagbar ist.

In einem ereignisgesteuerten System ist es nicht notwendig, solch detaillierte Entwürfe in der Entwicklungsphase zu erstellen. Ein Grund dafür ist, dass die Ausführungspläne dynamisch als Folge einer expliziten, aktuellen Nachfrage erzeugt werden. Abhängig vom Ablauf und Auftretens der Ereignisse, die dem System in einem speziellen Anwendungsszenario zugeführt werden, entstehen dynamisch verschieden Ablaufpläne.
Das Verhalten ist nicht deterministisch vorhersagbar.

VI.2 Testbarkeit (testability):

Hauptaspekt ist hierbei die *zeitliche Performanz* während verschiedener Systemauslastungen.

Das Vertrauen in das Einhalten der Zeitbedingungen bzw. die Rechtzeitigkeit eines ereignisgesteuerten Echtzeitsystems kann nur dann erreicht werden, wenn das System ausführlich unter simulierter Spitzenauslastung getestet wird. Das Testen unter normaler Auslastung ist dabei nicht ausreichend, da seltene C-Events, die das System ebenfalls verarbeiten können muss, u.U. nicht ausreichend häufig auftreten. Hilfreiche Schlüsse über das Systemverhalten während der Spitzenauslastung lassen sich hieraus nicht ziehen. Die Vorhersagbarkeit des Verhaltens im Falle seltener Ereignissituationen ist ein vorrangiges Ziel; diese Erkenntnisse werden als besonders wertvoll angesehen.

Da es in einem ereignisgesteuerten System keine detaillierten Pläne bezüglich des Zeitverhaltens gibt, ist es nicht sinnvoll, aufschlussreiche Performanztests auf Taskebene durchzuführen. In einem System, in dem alle Schedulingentscheidungen und Kommunikationssystemzugriffe dynamisch vorgenommen werden, existiert keine zeitliche Firewall, d.h. keine zeitlichen Abkapselungen/Schutzmechanismen der Tasks untereinander.

> *„A **temporal firewall** is an unidirectional **data-sharing interface** with **state-data semantics** where **at least one** of the interfacing subsystems accesses the temporal firewall according to an **a priori known schedule** and where at all points in time the information contained in the temporal firewall is **temporally accurate** for at least d_{acc} time units into the future.“*

Somit kann z.B. eine Änderung der Zeitwahl irgendeines Tasks weitreichende Konsequenzen auf das zeitliche Verhalten vieler anderer Tasks zur Folge haben. Die Tasks können nicht strikt voneinander separiert werden. Der kritische Punkt während der Auswertung ereignisgesteuerter Systeme ist somit, herauszufinden, ob die im Systemtest simulierten Auslastungsmuster die Realität hinreichend gut abdecken.

Die Ergebnisse des Performanztests in einem zeitgesteuerten System können mit den beim Entwurf erstellten detaillierten Plänen verglichen werden. Die diskreten Abfrage- und Aktionsraster stellen eine endliche Menge dar (nur zu bestimmten Zeitpunkten werden Zustandsänderungen „wirksam“), somit kann jede Reaktion auf einen Input beobachtet, sowie zeitlich und wertmäßig erfasst werden. Folglich ist das Vorgehen beim Test zeitgesteuerter Systeme systematischer als beim Testen ereignisgesteuerter Systeme, bei denen der Aufwand, einen gleichen Testumfang zu erlangen, unweit aufwändiger ist. Ein weiterer Grund ist, dass es in zeitgesteuerten Systemen weniger mögliche Ausführungsszenarien gibt, da dort die Reihenfolge von Zustandsänderungen zwischen zwei Beobachtungsfenstern unerheblich ist – es zählt nur die Zustandsänderung die als relevant zum Zeitpunkt des Abtastens angesehen wird (vgl. mehrmaliges Drücken des Aufzugsanforderungsknopfes: bei ereignisgesteuerten Systemen wird jedes Drücken des Knopfes an das Computersystem weitergeleitet – in zeitgesteuerten Systemen hingegen nicht).

VI.3 Ressourcennutzung (resource utilization):

In einem zeitgesteuerten System sind alle Schedules fest für die Höchstbelastung des jeweiligen Betriebsmodus vorgegeben. Das Zeitfenster für den jeweiligen Task muss mindestens so groß sein wie die maximale Ausführungszeit des Tasks. Ist die Differenz zwischen der durchschnittlichen und der maximalen Ausführungszeit eines Tasks groß, so wird im Allgemeinen nur ein kleiner Anteil der dem Task zugewiesenen Berechnungszeit benötigt. Sind viele verschiedene Operationsmodi des Systems zu betrachten, so ist es möglich, dass die Anzahl der statischen Schedules kombinatorisch explodiert.

In einem ereignisgesteuerten System müssen hingegen nur für diejenigen Tasks, die durch die jeweiligen Umstände angestoßen wurden, Schedules erstellt werden. Diese werden allerdings dynamisch erstellt, was zur Folge hat, dass die CPU bereits nach der tatsächlichen und nicht der maximalen Ausführungszeit des betreffenden Tasks wieder zur Verfügung steht. Allerdings sind zusätzliche Laufzeitressourcen (*run-time resources*) für die Ausführung der dynamischen Scheduling-Algorithmen, für die Synchronisation und das Speichermanagement zur Verfügung zu stellen.

Sind die Systemauslastungen niedrig oder durchschnittlich, so ist die Nutzung der Ressourcen in ereignisgesteuerten Systemen besser als die des vergleichbaren zeitgesteuerten Systems (Grundlage ist hier die durchschnittliche Ausführungszeit; in ereignisgesteuerten die tatsächliche). Erreicht ein System seine Spitzenauslastung, kann sich die Situation umkehren, da die für die Ausführung der Tasks verfügbare Zeit empfindlich durch die ansteigende Verarbeitungszeit für das Abwickeln der auftretenden Interrupts (es sind mehr auftretende Ereignisse zu bearbeiten), das Speichermanagement, die Synchronisation und die Scheduling-Algorithmen verbraucht wird.

VI.4 Erweiterbarkeit (extensibility):

Jedes System, das sich bewährt hat, muss im Laufe seiner Lebenszeit modifiziert und erweitert werden. Der Aufwand bzw. die Kosten, bestehende Funktionalität zu ändern und neue Funktionalitäten hinzuzufügen, wird durch den Begriff der ***Erweiterbarkeit*** beschrieben.

Es sind zwei Schritte nötig, um solche Änderungen durchzuführen:

a) Implementierung eines neuen oder modifizierten Tasks
b) Verifikation der zeitlichen Eigenschaften des geänderten Systems

In ereignisgesteuerten Systemen ist es relativ einfach, bereits bestehende Tasks zu modifizieren oder neue Tasks einem bestehenden Knoten zuzuordnen, da alle Scheduling und Synchronisationsentscheidungen bis zur Aktivierung der betreffenden Tasks verschoben werden; sie werden erst zur Laufzeit getroffen. Das Hinzufügen eines komplett neuen Knotens hat die Modifikation einiger Protokollparameter zur Folge. Allerdings kann eine lokale Änderung der Ausführungszeit eines Tasks Auswirkungen auf die zeitlichen Eigenschaften eines anderen Tasks in einem anderen Knoten haben, so z.B. kann sein LAN-Zugriff verzögert werden. Somit kann selbst eine lokale Änderung Auswirkungen auf das ganze System haben. Da keine bindenden zeitlichen Parameter für die individuellen Tasks gegeben sind, muss das komplette System hinsichtlich der Antwortzeiten erneut getestet werden, falls auch nur ein einzelner Task geändert wurde.

Sind sowohl die Wahrscheinlichkeit einer Änderung als auch die Zeit für den Systemtest proportional zur Anzahl der Tasks, so wachsen die Kosten, die für die Folgen der Änderungen aufzubringen sind, mehr als linear mit zunehmender Systemgröße. Bei großen Anwendungen ist der ereignisgesteuerte Ansatz unter Kostengesichtspunkten also nicht besonders zu empfehlen.

Aus Sicht der Antwortzeiten, ist jeder Task in einem zeitgesteuerten System eigenständig. Solange die Ausführungszeit eines modifizierten Tasks nicht größer als die maximale Ausführungszeit des ursprünglichen Tasks ist, hat die Änderung keine Auswirkungen auf die Antwortzeit des restlichen Systems. Sollte aber diese maximale Ausführungszeit überschritten werden, oder ein neuer Task hinzugefügt werden, müssen die statischen Schedules überarbeitet werden.
Der Aufwand, einen kompletten Knoten zum System hinzuzufügen, hängt vom Informationsfluss vom/zum neuen Knoten ab: Ist der Knoten *passiv*, sendet er z.B. keine Informationen zum System (z.B. ein Display), sind keine Modifikationen des existierenden Systems notwendig (die Kommunikationsprotokolle sind unidirektional und verwenden das „globale broadcasting“); ist der neue Knoten hingegen aktiv (er sendet Informationen zum System), so muss ihm ein Kommunikationsfenster zugewiesen werden und daraus resultierend müssen die Komunikationsabläufe (*Schedules*) neu gestaltet/berechnet werden.

Sollte es erforderlich sein, die Anzahl der Tasks dynamisch ändern zu können, ist es nicht möglich, statische Schedules zu erstellen. Die einzige Alternative ist dann die Implementierung der Funktionalität als ereignisgesteuertes System.

Zusammenfassend kann man feststellen, dass die Erweiterung eines ereignisgesteuerten Systems den erneuten Test des gesamten Systems verlangt; die Erweiterung zeitgesteuerter Systeme hat die Neuberechnung der statischen Schedules zur Folge.

VI.5 Annahmen/Voraussetzungen (assumption coverage)

Jedes Kontrollsystem basiert auf einer Menge von Annahmen über das Verhalten der Komponenten in der Umgebung. Die *Assumption Coverage* ist die Wahrscheinlichkeit, dass die Annahmen dem Szenario der realen Welt entsprechen.

Jedem Kontrollsystem liegt eine Menge von Annahmen/Voraussetzungen bezüglich des Verhaltens des zu kontrollierenden Objekts und des Computerkontrollsystems zu Grunde. Die Wahrscheinlichkeit, dass diese Annahmen/Voraussetzungen verletzt werden, limitiert die Zuverlässigkeit/Systemstabilität der gesamten Anwendung. Die Annahmen/Voraussetzungen, die sich auf das Verhalten des zu kontrollierenden Objekts beziehen werden als externe Annahmen/Voraussetzungen (*external assumptions*) bezeichnet. Diejenigen, die sich auf das Verhalten des Computersteuersystems beziehen, werden interne Annahmen/Voraussetzungen (*internal assumptions*) genannt.

In einem zeitgesteuerten System beziehen sich die kritischen externen Annahmen/Voraussetzungen auf die Häufigkeit der Wertänderungen in den RT-Entities. Die Dynamik einer RT-Entity bestimmt die Granularität des Beobachtungsfensters und somit die minimale Dauer eines Zustandes, so dass dieser auf jeden Fall vom System

bemerkt wird. Werden diese Annahmen/Voraussetzungen verletzt, können kurzlebige Zustände u.U. nicht bemerkt werden. Die dominierende interne Annahme/Voraussetzung in einem zeitgesteuerten System bezieht sich auf die maximale Ausführungszeit eines Tasks, die bereits beim Systementwurf festgesetzt werden muss.

In einem ereignisgesteuerten System ist die kritische externe Annahme/Voraussetzung die „Zufälligkeit“ des Auftretens der Ereignisse, sowohl unter normaler Systemauslastung als auch unter Spitzenauslastung. Diese Voraussetzung ist die Basis für die notwendigen Systemtests und die Vorhersage, ob das System die gegebenen deadlines einhalten wird.
Die kritische interne Annahme/Voraussetzung bezieht sich auf die Dienste des dynamischen Schedulers: Findet dieser eine ausführbare Lösung für das jeweilige Taskszenario unter Berücksichtigung der gegebenen Einschränkungen, oder nicht?

Zusammenfassung:

Eigenschaft	*zeitgesteuertes System*	*ereignisgesteuertes System*
zeitliche Vorhersagbarkeit	+	-
dynamische Ressourcenzuweisung	-	+
Determinismus	+	-
Flexibilität	-	+

Fazit:

Die Implementierung einer gegebenen externen Spezifikation durch ein zeitgesteuertes System verlangt eine sehr detaillierte Entwicklungsphase, in welcher die maximale Ausführungszeit aller zeitkritischen Programme festgelegt und die Ausführungspläne (*execution schedules*) für alle Betriebsmodi erstellt werden müssen. Als Resultat dieses detaillierten Planungsaufwands, ist das Zeitverhalten zeitgesteuerter Systeme vorhersagbar.

Bei Implementierung eines ereignisgesteuerten Systems ist dieser Aufwand nicht nötig. Auf der anderen Seite verlangt die Verifikation ereignisgesteuerter Systeme sehr umfangreiche Systemtests. Da es bei diesem Ansatz keine zeitlichen Abkapselungen der Tasks gibt (vgl. Erweiterbarkeit), ist dieser Ansatz für große Systeme nicht so gut geeignet wie der zeitgesteuerte Ansatz. Aus Sicht der Ressourcennutzung hingegen ist der ereignisgesteuerte Ansatz in vielen Fällen dem zeigesteuerten Ansatz überlegen.

Versuche, die Vorteile beider Ansätze zu kombinieren sind von großem Forschungsinteresse. Eine mögliche Lösung ist es, den ereignisgesteuerten Entwurf für die Knoten des Computersystems und den zeitgesteuerten Entwurf für die Kommunikation der Knoten untereinander anzuwenden, dies bietet den Vorteil, dass das System zeitliche Abkapselung zur Verfügung stellen würde und sehr gut auf der Architekturebene testbar wäre (*zeitgesteuerter Ansatz*) sowie eine relativ große Flexibilität des Schedulings und der Ressourcenausnutzung innerhalb der Knoten gegeben wäre (*ereignisgesteuerter Ansatz*).

*VII. **Literatur:***

- H. Kopetz, „*Real-Time Systems: Design Principles for Distributed Embedded Applications*“
- H. Kopetz, „*Event-Triggered versus Time-Triggered Real-Time Systems*“
- H. Kopetz, „*Should Responsive Systems be Event-Triggered or Time-Triggered?*“
- R. Williams, „*Real-time Systems Development*“, Elsevier 2006
- H. Wörn, U. Brinkschulte, „*Echtzeitsysteme*“, Springer 2005
- M. Oetken, Seminar „*Echtzeitbetriebssystemkerne*“, 1997