Lehrstuhl für Realzeit-Computersysteme

Realzeitfähiges Datenmanagement für eingebettete Systeme mit aktiven Realzeitdatenbanken

Alexander Münnich

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. (komm.) Dr.-Ing. Wolfram Boeck, em.
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr.-Ing. Georg Färber
	2. Univ.-Prof. Dr. rer. nat. Dr. rer. nat. habil. Manfred Broy

Die Dissertation wurde am 20.6.2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 28.9.2001 angenommen.

Inhaltsverzeichnis

Zusammenfassung

Traditionell verwalten Realzeitanwendungen ihre Daten in applikationsspezifischen Strukturen. Dies reicht jedoch bei datendominierten Realzeitsystemen, d. h. Anwendungen, die einen großen, konkurrierend genutzten Datensatz zu verwalten haben, nicht mehr aus. Besser ist es, diese datenbankbasiert zu implementieren, insbesondere bietet sich der Einsatz einer aktiven Realzeitdatenbank (ARTDB) an. Daraus ergeben sich mehrere Vorteile, vor allem steht immer ein konsistenter Datensatz als zentral verwaltete Ressource zur Verfügung. Die meisten heute im Bereich ARTDB existierenden Arbeiten beschäftigen sich mit weichen Realzeitsystemen, d. h. der Erzielung einer hohen durchschnittlichen Systemperformance. Die Anforderungen an eine ARTDB für eingebettete Realzeitsysteme mit harten Zeitanforderungen sind jedoch grundlegend anders. Besonders wichtig sind hier die Vorhersagbarkeit des Datenbankverhaltens, um die Verifikation von Zeitanforderungen zu ermöglichen (Echtzeitnachweis), und eine geringe Beeinflussung des Scheduling des Realzeitbetriebssystems durch die Datenbank zur Laufzeit (Optimalität).

Die Ergebnisse dieser Arbeit ermöglichen es, hauptspeicherbasiert ARTDBs auch in Systemen mit harten Zeitbedingungen einzusetzen. Basierend auf einer detaillierten Anforderungsanalyse wurde hierzu ein neues, optimiertes Datenbankmodell entwickelt. Dessen Laufzeitverhalten ist vorhersagbar und durch die Integration in ein Verfahren zum Realzeitnachweis können Zeitanforderungen für die gesamte Applikation, d. h. für das Anwenderprogramm in Verbindung mit der unterliegenden Datenbank, verifiziert werden. Das Datenbankmodell setzt sich folgendermaßen zusammen:

1. Entwicklung des Concurrency-Control Protokolls PRED-DF (predeclaration and data-flow analysis based semantic concurrency-control protocol). PRED-DF beruht auf der vorherigen Deklaration der Transaktionsressourcen, ist Lock-basiert, Deadlock-frei und generiert konfliktserialisierbare Schedules. Als Basis seiner Scheduling-Entscheidungen nutzt es zusätzlich Informationen, die während der Systementwicklung auf Grund der vordefinierten Transaktionen über den Datenfluß in der Applikation gewonnen werden können (z. B. Zyklen im Datenfluß), um so durch die Verwendung unterschiedlich restriktiver Locking-Strategien die Blockierzeiten der einzelnen Datensätze zu reduzieren. Damit kann die Transaktionsparallelität auf der Datenbank maximiert und die Verweildauer von Transaktionen in der Datenbank minimiert werden.
2. Entwicklung des Wissens- und Ausführungsmodells der ARTDB. Durch die aktive Funktionalität der Datenbank werden Datenabhängigkeiten abgeglichen und die einbettende Applikation über spezifische Datenbankzustände unterrichtet. Die Regeln in der Datenbank werden dabei so ausgeführt (bzgl. Frequenz, Priorität etc.), daß zum einen die temporale Konsistenz der Datenobjekte und die rechtzeitige Abarbeitung von Regeln sichergestellt ist und zum anderen die Datenbank in den Echtzeitnachweis eingebunden werden kann. Zusätzlich werden Optimierungsmöglichkeiten aufgezeigt, die es erlauben, die Effizienz bei der Abarbeitung der Konsistenzinformation zu verbessern.

Das Datenbankmodell wurde prototypisch implementiert und getestet. Sein Verhalten und seine Vorteile beim Einsatz in harten Realzeitsystemen werden analysiert und sein Einsatz anhand von Anwendungsbeispielen dargestellt.

Nomenklatur

$\prec, \succ$	Präzedenzbeziehungen (siehe 5.1.2)
$A(\mathcal{D}, \mathcal{R})$	Analysegraph (siehe 4.3.1)
A_{ik}	Action der Regel ρ_{ik} (siehe 4.3.1)
$\mathcal{A}_i$	Menge der Ausgangsdaten von x_i (siehe 4.3.1)
$a(x_i)$	Alter des kontinuierlichen Umweltdatums x_i (siehe 3.2.3)
$a_y(x_i)$	Alter des Umweltdatums x_i, das zur Berechnung von y verwendet wurde (siehe 3.2.3)
$a^l_{i,j}$, a_i	Elemente der Ereignistupel T^l_i des Ereignisstroms ES^l (siehe 5.1.1)
$\mathbb{B}$	Menge der boolschen Werte, $\mathbb{B} = \{t, f\}$, $t \triangleq$ true $f \triangleq$ false
$C_a(I)$	Rechenzeitanforderungsfunktion (siehe 5.1.1)
$\hat{c}, \check{c}$	maximale, minimale Verarbeitungszeit eines Task bzw. einer Transaktion (siehe 5.1.2)
c_{ik}	Condition der Regel ρ_{ik} (siehe 4.3.1)
$\mathcal{D}$	Menge der Daten in der Datenbank, $\mathcal{D} = \{x_1, x_2, \ldots x_k\}$ (siehe 4.1)
d	Deadline (siehe 5.1.2)
$\delta(x_i, x_j)$	Dispersion, d. h. zeitliche Korrelation zwischen x_i und x_j (siehe 3.2.3)
$E_{\mathrm{ap},i}$	internes Ereignis der Applikation (siehe 5.1.2)
$E^l_a(t), E^l_r(t)$	Ereignisfunktion zum Ereignisstrom ES^l (siehe 5.1.1)
ES^l	Ereignisstrom zum Ereignistyp l (siehe 5.1.1)
$G(s)$	Konflikt-Graph des Schedules s (siehe B.1)
g	Menge der Abb. g_j, die die Zuordnung von R auf W durch τ beschreiben (siehe 4.1)
$j_k(x_i)$	Jitter des Aufnahmezeitpunktes eines Datums x_i (siehe 4.3.1)
$\mathcal{K}$	Menge der Konsistenzinformation einer Applikation (siehe 4.3.1)
$\mathbb{N}$	Menge der natürlichen Zahlen
$p_i(x_i)$	Operation auf der Datenbank, $p_i(x_i) \in \{r(x_i), w(x_i)\}$ (siehe 4.1)
$t_s(x_i)$	Aufnahmezeitpunkt des Datums x_i durch den zugeordneten Sensor (siehe 3.2.3)
R_i	Lese-Datensatz der Transaktion τ_i (siehe 4.1)
$\mathcal{R}$	gemeinsame Regelbasis (siehe 4.3.1)
$\mathcal{R}'$	Regelbasis (siehe 4.3.1)
ρ_{ik}	k-te Regel, die mit dem Datum x_i verknüpft ist (siehe 4.3.1)
$r(x_i)$	Lesen des Wertes eines Datenobjektes x_i aus der Datenbank $\mathcal{D}$ (siehe 4.1)
s	Schedule (siehe 4.1)
T_i	Task i (siehe 5.1.2)
T_i	Schwellwert für das Alter von x_i (siehe 3.2.3)
T'	Schwellwert der Dispersion (siehe 3.2.3)
$T_{x_i,y}$	Schwellwert für $a_y(x_i)$ (siehe 3.2.3)
$\mathcal{T}$	Gesamtheit aller Transaktionen einer Applikation (siehe 4.1)
$\mathcal{T}_f(\tau_i)$	Friend-Set der Transaktion τ_i (siehe 4.2.2)
$\mathcal{T}_k$	Konfliktmenge (siehe 4.2.2)
$\mathcal{T}_{ka}$	azyklische Konfliktmenge (siehe 4.2.2)
$\mathcal{T}^n_{ka}$	azyklische, normalisierte Konfliktmenge (siehe 4.2.2)
τ_i	Transaktion, $\tau_i \in \mathcal{T}$ (siehe 4.1)
W	Wait-Graph (siehe B.2)
W_i	Schreib-Datensatz der Transaktion τ_i (siehe 4.1)
$w(x_i)$	Schreiben eines neuen Wertes des Datenobjektes x_i in $\mathcal{D}$ (siehe 4.1)
x_i	Datenobjekt der Datenbank $\mathcal{D}$, $x_i \in \mathcal{D}$ (siehe 4.1)
z^l_i	Zykluszeit des i-ten Tuples im Ereignisstrom ES^l (siehe 5.1.1)

1. Einleitung

Auf Grund der steigenden Leistungsfähigkeit von Prozessoren bei gleichzeitig sinkenden Hardware-Preisen dringen in den letzten Jahren eingebettete Systeme in immer mehr Anwendungsgebiete vor. Darunter sind zunehmend auch Bereiche, in denen nicht nur richtiges, sondern auch rechtzeitiges Reagieren von besonderer Bedeutung sind, z. B. bei Sicherheitssystemen im Auto und bei der Steuerung bzw. Regelung von Produktionsanlagen. Gleichzeitig steigen die Ansprüche der Anwender, die immer höhere Forderungen an den Funktionsumfang und den Bedienungskomfort solcher Systeme stellen. Beides zusammen führt zu einer drastischen Zunahme der Komplexität und des Umfangs von Software-Entwicklungsprojekten.

Um die steigende *Komplexität* von Software zu beherrschen, werden Techniken wie Objektorientierung, komponentenbasierte Entwicklung und modulare Programmierung eingesetzt. Zunehmend werden die Systeme auch verteilt implementiert und basieren auf Client-Server Architekturen.

Traditionell verwalten Realzeitanwendungen ihre Daten in applikationsangepaßten Strukturen. Dies reicht jedoch heute auf Grund der steigenden Anforderungen nicht mehr aus, da mit dem *Umfang* der Anwendungen vor allem auch die Größe der Datensätze steigt. Eine leistungsfähige Datenhaltung muß in der Lage sein, die benötigten Daten systematisch strukturiert unter Realzeitbedingungen mit hoher Qualität zur Verfügung zu stellen. Das heißt im einzelnen:

- effiziente Verwaltung großer Datenmengen,
- Koordination von konkurrierenden Zugriffen auf dieselben Daten durch unterschiedliche, voneinander unabhängige Systemteile, z. B. verschiedene Komponenten,
- Adaption unterschiedlicher Datensichten,
- Konsistenzsicherung redundanter Informationen in verschiedenen Datensätzen,
- Nebenläufigkeit/Parallelität in der Datenbank,
- Datenverarbeitung unter definierten Zeitanforderungen (Deadlines).

Dies sind zum Teil die klassischen Aufgabenstellungen von Datenbanken, weshalb sie in jüngster Zeit vermehrt auch in der Entwicklung von eingebetteten Realzeitsystemen eingesetzt werden, insbesondere als sogenannte *aktive Realzeitdatenbanken* (Active Real-Time Database, ARTDB) [Eri97][HB99]. Diese relative junge Datenbankentwicklung kombiniert die Eigenschaften von Realzeitdatenbanken und aktiven Datenbanken. Erstere haben zum Ziel, zusätzlich zur normalen Datenbankfunktionalität, die Einhaltung definierter Zeitanforderungen zu garantieren. Aktive Datenbanken erlauben es, innerhalb der Datenbank Ereignisse und damit verknüpfte Aktionen zu definieren. Zusätzlich können Bedingungen festgelegt werden, unter denen diese Aktionen ausgeführt werden sollen (siehe Kapitel 2).

Beim Einsatz einer ARTDB stehen dort gemeinsam genutzte Daten als zentral verwaltete Ressource zur Verfügung. Für die Softwareentwicklung ergeben sich daraus folgende Vorteile:

- Vereinheitlichung der Datensicht,
- wohldefinierte Schnittstellen zu den verschiedenen Teilen einer Applikation,
- Zentralisierung der Datenhaltung und Vermeidung von Redundanz,
- automatische Konsistenzsicherung und Vermeidung von unnötig häufigen Datenbankabfragen (Polling) mit Hilfe der aktiven Datenbankfunktionalität, d. h. über entsprechende Regeln in der Datenbank.

Dies führt zum einen zu einer Erhöhung der Datenqualität, da die Integrität der Daten auch bei konkurrierenden Zugriffen und auch in verteilten Systemen gewährleistet ist. Zum anderen ergibt sich aber auch eine deutliche Vereinfachung der Systemarchitektur [MBFW99][BMW00] und eine merkliche Entlastung des Softwareentwicklers von überflüssigen Aufgaben, da die Daten und deren Konsistenzinformationen einheitlich zentral gespeichert sind und Updates abhängiger Datensätze automatisch durchgeführt werden können. Dies bedeutet bessere und einfachere Softwareentwicklung und damit geringere Kosten, mehr Sicherheit und mehr Verläßlichkeit.

In vielen Anwendungsbereichen werden aktive Realzeitdatenbanken bereits erfolgreich eingesetzt: komplexe Meßgeräte [MBFW99][BMW00], Netzwerk-Servicedienste [SPSR93], Autonome Mobile Systeme [PSS93], Steuerungs-, Regelungs- und Fertigungstechnik (z. B. Produktionssteuerung und -überwachung [TSYT97]), Simulation und Test (HIL) [Blö00].

Insbesondere beim Einsatz von eingebetteten Systemen in zeit- und sicherheitskritischen Anwendungen mit harten Zeitanforderungen (Deadlines) ist es notwendig, im vorhinein zu überprüfen, ob das System in der Lage ist, immer rechtzeitig zu reagieren (*Realzeitnachweis*). Wird eine Datenbank eingesetzt, muß diese in die Analyse mit einbezogen werden. Als problematisch erweist sich dabei die Tatsache, daß das Laufzeitverhalten einer Datenbank in der Regel nicht vorhersagbar ist und zusätzlich die Einflüsse der Datenbank auf das Scheduling des Realzeit-Betriebssystems, sowie die Kopplung von Datenbank, Applikation und aktiver Funktionalität zu berücksichtigen sind.

Zur Analyse von traditionellen Realzeitanwendungen sind eine Vielzahl von Algorithmen bekannt (siehe Abschnitt 2.1.2). Es existiert allerdings gegenwärtig kein Verfahren, das zusätzlich die Verwendung einer Realzeitdatenbank oder den Einsatz einer aktiven Datenbank berücksichtigt. In dieser Arbeit wird daher eine Methode vorgestellt, die bei Einprozessorsystemen den Realzeitnachweis auch für solche eingebettete Systeme führen kann, die eine aktive Realzeitdatenbank (ARTDB) verwenden. Das Verfahren baut auf einem speziell für den Anwendungsbereich optimierten Datenbankmodell auf. Dieses verwendet ein neu entwickeltes, semantisches Concurrency-Control Protokoll (PRED-DF) und unterstützt einen an die Anwendungsdomäne angepaßten Satz an aktiver Funktionalität. Der Softwareentwickler hat so den Vorteil, aktive Realzeitdatenbanken auch in eingebetteten, harten Realzeitsystemen nutzen zu können, da es nun auch hier möglich ist, die Einhaltung von Zeitanforderungen im vorhinein zu überprüfen.

Im folgenden gibt Kapitel 2 einen Überblick über den aktuellen Stand der Forschung. Hierbei werden sowohl Realzeitsysteme und Nachweismethoden zur Einhaltung von Zeitbedingungen, als auch aktive Realzeitdatenbanken behandelt. Vor diesem Hintergrund werden am Ende des Kapitels die genauen Zielsetzungen der Arbeit präzisiert und es wird ein Überblick über die weitere Gliederung der Arbeit gegeben.

2. Stand der Forschung

2.1. Realzeitsysteme

2.1.1. Begriffsdefinitionen

Als *Realzeitsysteme* bezeichnet man Anwendungen, bei denen ...

> „... die Korrektheit des Systems nicht nur vom logischen Ergebnis der Berechnung abhängt, sondern auch vom Zeitpunkt, zu dem dieses Ergebnis bereitgestellt wird [Sta88]."

Das heißt, nicht nur die *richtige*, sondern auch die *rechtzeitige* Bereitstellung eines Resultats ist in solchen Systemen für die korrekte Funktion einer Applikation notwendig.

In der Regel werden die Zeitschranken für die Bearbeitung (*Deadlines*) durch den einbettenden technischen Prozeß vorgegeben. Je nachdem, in welchem Anwendungsbereich die Systeme eingesetzt werden, sind Überschreitungen der Deadlines mit unterschiedlichen Konsequenzen verbunden. Bei harten Deadlines sind die Konsequenzen einer Zeitüberschreitung in der Regel katastrophal, d. h. sehr teuer und unter Umständen sogar mit der Gefährdung von Menschenleben verbunden. Bei weichen Deadlines dagegen ist meist nur eine Verminderung der Systemperformance oder der Servicequalität die Folge. Entsprechend Abbildung 2.1 lassen sich Zeitbedingungen durch die sogenannte Wertefunktion V(t) kategorisieren [Loc86]. V(t) gibt den Verlauf des Wertes über der Zeit t an, den die Erledigung einer bestimmten Aufgabe für das Gesamtsystem hat. Die vorliegende Arbeit beschäftigt sich mit Realzeitsystemen mit harten Deadlines.

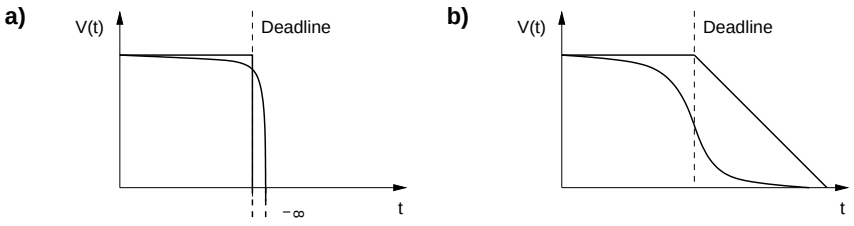


Abbildung 2.1.: Kategorisierung von Deadlines mit Hilfe der Wertefunktion V(t). Teilbild a) zeigt mögliche Verläufe der Wertefunktion für harte Zeitbedingungen, b) zeigt beispielhaft verschiedene Verläufe der Wertefunktion V(t) für weiche Deadlines.

Da in Realzeitsystemen mehrere Aufgaben praktisch gleichzeitig erledigt werden müssen, laufen die unterschiedlichen Bearbeitungseinheiten einer Applikation (*Tasks*) unter der Kontrolle eines Realzeit-Betriebssystems (RTOS) quasiparallel ab. Prinzipiell existieren dabei zwei verschiedene Möglichkeiten des *Scheduling*, d. h. diesen Tasks Rechenzeit zuzuteilen: zeitgesteuerte (statische) und ereignisgesteuerte (dynamische) Rechenzeitzuteilung. Beim *statischen Scheduling* wird bereits vor dem Systemstart ein vollständiger „Fahrplan" zur Behandlung der unterschiedlichen Tasks festgelegt. Dieses Verfahren ist besonders für Systeme mit einem großen Anteil

vorhersagbarer, periodisch wiederkehrender Vorgänge geeignet, z. B. Regelungssysteme. Beim *dynamischen Scheduling* wird der aktive Task zur Laufzeit auf Grund des aktuell vorliegenden Systemzustandes vom Scheduler bestimmt. Dieses Vorgehen ist besonders für Systeme geeignet, deren Systemzustand sich nicht periodisch, sondern unregelmäßig und spontan ändert. In diesem Fall kann mit dynamischem Scheduling schneller auf Ereignisse aus dem einbettenden Prozeß reagiert werden und es muß weniger Rechenleistung vorgehalten werden [Fär92][Loc92].

Welcher Task bei dynamischem Scheduling dem Prozessor zugeteilt wird, richtet sich nach dem verwendeten Scheduling Algorithmus, z. B. FCFS (First Come First Served), Round Robin, prioritätsbasiertes Scheduling. Zusätzlich gibt es Verfahren bei denen die Zuteilung nach einem über die Zeit veränderlichen Kriterium erfolgt, z. B. Earliest-Deadline-First (EDF) oder Least-Laxity-First[1].

Die vorliegende Arbeit bezieht sich auf ereignisgesteuerte Realzeitsysteme, bei denen die Rechenzeit nach preemptivem EDF-Scheduling[2] zugeteilt wird. Zum einen läßt sich zeigen, daß dieses Schedulingverfahren auf Einprozessorsystemen optimal ist [SSRB98][3], zum anderen ist EDF-Scheduling ein weit verbreitetes und häufig untersuchtes Scheduling Verfahren.

2.1.2. Realzeitnachweis

Bei Applikationen mit harten Deadlines ist es unerläßlich, deren Zeitverhalten im vorhinein zu analysieren, um sicherzustellen, daß *alle* Zeitbedingungen in *allen* möglichen Anwendungsfällen eingehalten werden. Dies bezeichnet man als Realzeitnachweis oder Realzeitanalyse. Es ist hierfür nicht ausreichend, die Systeme lediglich im Probebetrieb zu testen oder die Einhaltung von Deadlines durch Simulation zu überprüfen, da so nicht sichergestellt ist, daß alle möglichen Systemzustände erfaßt werden. Auch die Verwendung von Rechnersystemen, die für die Anwendung stark überdimensioniert sind, ist – abgesehen von den dadurch zusätzlich verursachten, überproportional hohen Kosten – keine Garantie für die Einhaltung aller Deadlines.

Grundsätzlich gliedert sich jeder Realzeitnachweis in zwei Teile: eine Beschreibungstechnik für die Umwelt der Applikation (Prozeßmodell), und ein Nachweisverfahren, das, basierend auf einem abstrahierten Taskmodell der Applikation, untersucht, ob das betrachtete System unter dem Einfluß der Umwelt immer rechtzeitig reagiert.

Prozeßmodelle

Das Prozeßmodell gibt Auskunft darüber, welche Ereignisse in welcher zeitlichen Korrelation auf das System wirken. Hierfür existieren zwei grundlegend verschiedene Ansätze:

Statistische Beschreibungstechniken Bei der Prozeßbeschreibung durch statistische Größen werden Wahrscheinlichkeitsverteilungen für die Ankunftszeiten der unterschiedlichen Ereignisse angegeben. Diese werden meist negativ exponentiell oder Gauß-verteilt angenommen. Ziel ist es in der Regel, eine Aussage über die Systemperformance oder über mittlere Antwortzeiten zu machen.

[1] Laxity = Deadline − (aktuelle Zeit) − (vom Task bis zur Fertigstellung benötigte CPU-Zeit).

[2] „Preemptiv“ bedeutet, daß allein das Betriebssystem über die Zuteilung der Rechenzeit entscheidet, d. h. Tasks sind unterbrechbar und es ist immer der Task aktiv, der rechenbereit ist und die höchste Priorität besitzt.

[3] „Optimal“ heißt in diesem Zusammenhang, daß, falls überhaupt ein gültiger Schedule auf einem Einprozessorsystem existiert, auch durch EDF-Scheduling ein solcher erzeugt wird.

Darüber hinaus sind lediglich Aussagen über die Wahrscheinlichkeit möglich, mit der eine bestimmte Deadline eingehalten wird. Ein Nachweis über die stets korrekte Funktion einer Anwendung kann so nicht geführt werden. Diese Beschreibungstechnik ist zum Nachweis der Realzeitfähigkeit für Systeme mit harten Deadlines aus diesem Grund ungeeignet.

Formale Techniken zur Prozeßbeschreibung Mit formalen Beschreibungstechniken versucht man, den ungünstigsten Fall an möglichen Umweltereignissen mathematisch so exakt wie möglich zu beschreiben. Die meisten Autoren gehen dabei von einem periodischen Auftreten der Ereignisse aus. Die Deadline entspricht dem Bearbeitungszyklus [LL73]. Ergänzende Arbeiten behandeln auch aperiodische Ereignisse. Diese sind aber in der Regel Tasks mit weicher Deadline zugeteilt [SSL89][SB94]. Werden mit aperiodischen Ereignissen harte Deadlines verbunden (sporadische Ereignisse [Mok83]), so wird in den Arbeiten ein Mindestabstand zwischen den einzelnen Ereignissen gefordert, d. h. aperiodische Tasks werden im Worst-Case wie periodische Tasks modelliert.

Eine wichtige Erweiterung und Verfeinerung der formalen Beschreibung von Ereignissen und deren zeitlicher Korrelation stellt ihre Beschreibung durch Ereignisströme und Ereignisabhängigkeitsmatrizen dar [Gre93a][Gre93b]. Diese erlauben es, sowohl eine genaue Aussage über die maximale Anzahl von Ereignissen innerhalb eines definierten Zeitintervalls zu treffen (auch die Beschreibung von Bursts ist möglich), als auch die zeitliche Korrelation von Ereignissen unterschiedlichen Typs näher zu spezifizieren. Detailliert wird diese Beschreibungstechnik in Abschnitt 5.1.1 beschrieben.

Nachweisverfahren

Historisch gesehen wurde das erste Nachweisverfahren von Liu und Layland veröffentlicht [LL73]. Es befaßt sich mit Anwendungen für Einprozessorsysteme, die auf synchronen, periodischen Tasks mit festen Prioritäten basieren. Dem Taskmodell liegen folgende, restriktive Annahmen zugrunde:

- A1 Alle Tasks mit harter Deadline sind periodisch. Die Zeit zwischen zwei Aufrufen ist konstant.
- A2 Jeder Task ist beendet, bevor die nächste Anforderung erfolgt (d. h. Deadline = Periode).
- A3 Tasks sind unabhängig voneinander. Ein bestimmter Task hängt nicht vom Zustand eines anderen Tasks ab. Das heißt, es gibt keine Taskpräzedenzen und keine gemeinsam genutzten Ressourcen.
- A4 Die Bearbeitungszeit eines Tasks ist konstant und variiert nicht mit der Zeit.
- A5 Nichtperiodische Tasks sind eine Ausnahme, sie haben keine harten Echtzeitbedingungen.

Zum heutigen Zeitpunkt existiert eine Vielzahl von Algorithmen, die diese Einschränkungen in verschiedener Hinsicht aufweichen und somit die Verifikation von Realzeitanforderungen in pseudo-polynomialer Zeit auch für eine große Zahl an komplex strukturierten Tasksets zulassen [LW82][TBW94][Spu95]. Für den Bereich des EDF-Scheduling gibt [SSRB98] eine ausführliche Zusammenfassung.

Für Systeme, die die Rechenzeit nach EDF zuteilen, wird in [Gre93b] ein Analysealgorithmus vorgestellt, der auf der flexiblen und genauen Beschreibung der einbettenden Umwelt durch Er-

eignisströme und Ereignisabhängigkeitsmatrizen beruht. Die Arbeit verwendet ein Taskmodell mit den folgenenden Lockerungen gegenüber A1 - A5:

- Tasks können nichtperiodisch sein (A1, A5).
- Für die einzelnen Tasks existieren Warteschlangen (A2).
- Gegenseitiger Ausschluß von Tasks, Taskpräzedenzen und abhängige Ereignisse können modelliert werden (A3).

Zusätzlich zu den bisher erwähnten Arbeiten zum Realzeitnachweis wurden insbesondere in jüngster Zeit mehrere Verfahren entwickelt, die die Verifikation von Zeitanforderungen auf der Basis einer formalen Anwendungsspezifikation, beispielsweise mit Hilfe von „Timed Automata“ [EWY99] oder „Algebra of Communicating Shared Ressources“ (ACSR) [KLP+98], erlauben. Solche Ansätze erlauben zwar einen streng formalen Nachweis sowohl der funktionalen als auch der zeitlichen Anforderungen an ein System, problematisch ist allerdings stets die Explosion des zu betrachtenden Zustandsraumes. Deshalb ist die Analyse komplexer Systeme mit diesen Methoden in der Praxis nur äußerst schwierig zu realisieren.

Aus diesem Grund und insbesondere wegen der genauen und flexiblen Möglichkeit der Umweltbeschreibung basieren die hier erarbeiteten Methoden zum Realzeitnachweis auf der weiter oben zitierten Arbeit von Gresser.

Zwei wichtige Voraussetzungen aller Nachweisverfahren sind die Vorhersagbarkeit des Applikationsverhaltens zur Laufzeit (Predictability) und die Möglichkeit, die Worst-Case Execution Time (WCET) aller Programmteile bestimmen zu können. Die Forderung nach Vorhersagbarkeit überträgt sich dabei direkt auf die Datenbank und den Determinismus deren Verhaltens. Dieser Punkt bildet eine wichtige Fragestellung der vorliegenden Arbeit. Die Bestimmung der Worst-Case Execution Time ist nicht Gegenstand dieser Arbeit. Die benötigten Zeiten werden als bekannt vorausgesetzt. Zu deren Bestimmung existieren mehrere unterschiedliche Ansätze und Verfahren [CBW94][Pet00][LHT00]. Hier wird auf die entsprechende Literatur verwiesen.

2.2. Zeitverhalten aktiver Realzeitdatenbanken

2.2.1. Realzeitdatenbanken

Der Benutzer greift auf eine Datenbank mittels Transaktionen zu (siehe Abschnitt 3.2.1). Eine Transaktion (TA) ist die Zusammenfassung mehrerer Lese- und Schreiboperationen auf der Datenbank, die der Benutzer als eine logische Einheit begreift und stellt damit die Abbildung eines Vorgangs in der „realen Welt“ auf die „Welt der Datenbank“ dar (z. B. eine Kontoauszahlung).

Realzeitdatenbanken haben zum Ziel, zusätzlich zur normalen Datenbankfunktionalität, die Realzeitfähigkeit in der Transaktionsverarbeitung zu gewährleisten [BLS97][BFW97]. Klassische Datenbanken sind für den Einsatz in Realzeitsystemen nicht geeignet, da sie in der Regel dahingehend optimiert sind, für die Operationen auf der Datenbank den Durchsatz zu maximieren und die mittlere Antwortzeit zu minimieren. Für Transaktionen ist dann eine Vorhersage über die Einhaltung von Zeitbedingungen und eine Aussage über das Verhalten zur Laufzeit nicht möglich. Die wichtigsten Quellen der Nicht-Vorhersagbarkeit des Laufzeitverhaltens von Datenbanken sind [Ram93]:

- *Daten- und Ressourcenkonflikte.* Auf Grund von Konflikten zwischen verschiedenen Transaktionen (Zugriff auf dieselben Daten) können sich diese gegenseitig blockieren.

- *„Abort“ mit „Rollback“ und „Restart“ von Transaktionen.* Transaktionen können bei Konflikten durch die Datenbank abgebrochen und neu gestartet werden. Unter Umständen ist die Zahl der notwendigen Neustarts, d. h. die maximale Ausführungszeit einer Transaktion sehr hoch.
- *Dynamisches Paging und I/O.* In traditionellen Datenbanken werden die Daten auf Sekundärspeichern (z. B. Harddisks) gehalten. Je nachdem, ob sich die Daten bereits im Hauptspeicher oder Cache befinden oder erst aus dem Sekundärspeicher gelesen werden müssen, sind die Zugriffszeiten sehr unterschiedlich. Laufzeitabschätzungen sind so nur sehr schwer möglich bzw. äußerst pessimistisch.
- *Abhängigkeit der Transaktionsausführung von Datenwerten.* Die Ausführungspfade von Transaktionen hängen von den Werten in der Datenbank ab.

Bei Systemen mit harten Zeitanforderungen ist aber Vorhersagbarkeit, wie in Abschnitt 2.1.2 bereits dargestellt, eine wesentliche Voraussetzung für den Nachweis der Realzeitfähigkeit. Bei der Entwicklung von Realzeitdatenbanken versucht man daher, diese Probleme zu beheben. Ansatzpunkte hierzu sind die Verlagerung der Datenbank in den Hauptspeicher und optimierte Concurrency-Control Protokolle, die auch die Priorisierung von Transaktionen zulassen.

Concurrency-Control Protokolle

Das Concurrency-Control Protokoll (CC-Protokoll) hat die Aufgabe, Konflikte zwischen Transaktionen, die gleichzeitig konkurrierend auf dieselben Daten in der Datenbank zugreifen, aufzulösen. Dies muß so geschehen, daß die in der Datenbank festgehaltenen Ergebnisse einem definierten Korrektheitskriterium (i. d. R. Konflikt-Serialisierbarkeit) entsprechen und eventuell zugewiesene Transaktionsprioritäten berücksichtigt werden (siehe Abschnitt 3.2.2). Dabei ist zwischen der Detektion des Konfliktes und der darauf folgenden Auflösung zu unterscheiden.

Konfliktdetektion Prinzipiell existieren drei verschiedene Möglichkeiten, Konflikte zwischen Transaktionen festzustellen:

1. *Locks.* Um sicherzustellen, daß Datenobjekte immer nur unter gegenseitigem Ausschluß genutzt werden, dürfen unterschiedliche Transaktionen ein Datum nur bearbeiten, wenn sie einen entsprechenden Lock auf dieses Datum besitzen (z. B. Two-Phase Locking Protocol, 2PL [EGLT76]). Es wird zwischen unterschiedlichen Lock-Modi unterschieden (exclusive, shared).
2. *Zeitstempel.* Bei jedem Zugriff auf Datenobjekte wird ein Zeitstempel vergeben, der eine Überprüfung der korrekten Reihenfolge bei der Transaktionsbearbeitung erlaubt. Dieser leitet sich z. B. aus der Entstehungszeit einer Transaktion ab, die dann auch die Serialisierungsreihenfolge bestimmt [BEHR82].
3. *Graphbasierte Verfahren.* Anhand eines in der Datenbank verwalteten Graphen wird überprüft, ob Transaktionskonflikte aufgetreten sind (z. B. Serialization Graph Testing, SGT [YWLS94]).

Des weiteren sind diese Verfahren noch dahingehend zu unterscheiden, zu welchem Zeitpunkt auf einen Konflikt hin überprüft wird. *Pessimistische Verfahren* überwachen schon während der Ausführung einer Transaktion, ob ein Konflikt eingetreten ist. *Optimistische Verfahren* prüfen erst am Ende, bevor die Ergebnisse endgültig in der Datenbank festgeschrieben werden.

Konfliktlösung Ist ein Konflikt festgestellt, muß ein passendes Verfahren diesen Konflikt auflösen, d. h. entscheiden, wie mit den einzelnen Transaktionen zu verfahren ist. Hierfür existieren vier verschiedene Möglichkeiten:

1. *Blockierung.* Eine oder mehrere Transaktionen werden blockiert bzw. verzögert.
2. *Abbruch (Abort).* Eine oder mehrere Transaktionen werden abgebrochen und später von vorne neu gestartet.
3. *Multiversioning.* In der Datenbank werden mehrere historische Versionen eines Datums gehalten. Je nach ihrer Lage in der Serialisierungsreihenfolge greift eine Transaktion auf die für sie passende Version zu [BHR80].
4. *Anpassung der Serialisierungsordnung.* Man versucht Konflikte aufzulösen, indem man Transaktionen zur Zertifizierungszeit umsortiert [LR99].

Für die meisten Kombinationen von Konfliktdetektion und Konfliktauflösung existieren entsprechende Anpassungen für Realzeitdatenbanken, zum Beispiel: 2PL-HP (2PL-High Priority, Locking und Abbruch [AGM89]), 2PL und Multiversioning [BHR80], Optimistisches CC-Protokoll [LR99]. Bei der Konfliktlösung durch Blockierung treten zusätzliche Probleme durch Prioritätsinversionen und Deadlocks auf. Diese müssen durch entsprechende Mechanismen (z. B. Prioritätsvererbung, Prioritätsanhebung, Wound-Wait/Wait-Die) behandelt bzw. vermieden werden [SRL90] [RSI78].

Eine weitere Möglichkeit ist, Serialisierbarkeit und Rechtzeitigkeit gegeneinander abzuwägen und neue CC-Protokolle zu entwickeln, die schwächere Korrektheitskriterien zur Grundlage haben, um die Einhaltung von Deadlines zu erleichtern. Diese CC-Protokolle gestatten zugunsten von höherer Parallelität und aktuelleren Datenwerten (Verbesserung der temporalen Konsistenz, siehe Abschnitt 3.2.3) ein gewisses Maß an Inkonsistenz der Datenobjekte innerhalb der Datenbank. Ein Beispiel hierfür ist die Aufteilung der Datenbank in interne und externe Datenobjekte. Über die Festlegung einer Transaktions-Kompatibilitäts-Matrix wird Transaktionen, die neuere Werte für externe Datenobjekte bereitstellen, der Datenbankzugriff auch bei Datenkonflikten gestattet [Lin89]. Ein anderes Beispiel ist ϵ-Serialisierbarkeit. Hier wird eine maximale Abweichung der Datenbank vom konsistenten Zustand zugelassen. Die Einhaltung dieser Abweichung wird durch einen Divergenz-Kontroll-Algorithmus überwacht [PL92][WYP92].

Eine Zusammenfassung der unterschiedlichen Concurrency-Control Protokolle für Realzeitdatenbanken findet man in [YWLS94][UB92]. Untersuchungen bezüglich des Zeitverhaltens solcher Protokolle beschränken sich jedoch in der Regel auf die Untersuchung der Systemperformance [AGM92][LS96]. Rechtzeitige Reaktion ist nicht Gegenstand der Untersuchungen. Lediglich in [BBDW97] wird eine Worst-Case Analyse für diskresidente Datenbanksysteme vorgestellt (2PL-HP). Hier wird das Problem allerdings nur für ein fixes System periodischer Transaktionen fester Priorität behandelt. Eine Anbindung an die einbettende Anwendung wird nicht untersucht.

Hauptspeicherbasierte Datenbanken

Durch die Verlagerung der Datenbank in den Hauptspeicher ist es zum einen möglich, eine wesentlich bessere Performance zu erzielen, zum anderen kann der Beitrag der Disk-I/O zur Unvorhersagbarkeit des Datenbankverhaltens vermieden werden [GMS92][WKO+84][Eic87]. Viele Arbeiten zu hauptspeicherbasierten Datenbanken beziehen sich jedoch auf Recovery- und Loggingaspekte [LC87][KB91] und auch hier liegt der Schwerpunkt bis jetzt vorwiegend auf der

Untersuchung bzw. der Steigerung der Systemperformance und nicht auf der Entwicklung von vorhersagbaren CC-Protokollen [LL93].

In [UB98] wird ein Locking-basiertes CC-Protokoll für hauptspeicherbasierte Datenbanken vorgestellt (PRED), das eine Vorhersagbarkeit des Datenbankverhaltens erlaubt und auf der Prädeklaration der verwendeten Lese- und Schreibsets einer Transaktion beruht. Auch hier steht die Untersuchung der Datenbankperformance im Vordergrund, eine Analyse des Realzeitverhaltens wird nicht durchgeführt.

In [Kim99] wird ein Verfahren zur vorhersagbaren Transaktionsbearbeitung vorgestellt, das auf einer Klassifizierung von Transaktionen (bezüglich timing constraints, arrival pattern, data requirements etc.) beruht. Für periodische Transaktionen ist hier eine Aussage über die Einhaltung von Zeitbedingungen möglich. Die erzeugten Schedules sind jedoch nicht serialisierbar. Eine Analyse von spontan auftretenden Ereignissen und eine Einbindung der Datenbank in einen Realzeitnachweis für die gesamte Applikation, d. h. die gemeinsame Betrachtung von Anwendungsprogramm und Datenbank, findet nicht statt.

2.2.2. Aktive Datenbanken

> „Ein Datenbanksystem heißt aktiv, wenn es zusätzlich zu den üblichen Fähigkeiten in der Lage ist, definierbare Situationen in der Datenbank zu erkennen und als Folge davon bestimmte, ebenfalls definierbare Reaktionen auszulösen [DG96].“

In diesem Zusammenhang wird unter einer *definierbaren Situation* das Eintreten eines Ereignisses in der Datenbank oder deren Umfeld (siehe Abbildung 2.2) bei gleichzeitiger Erfüllung einer Bedingung an den Zustand der Datenbank und/oder ihrer Umgebung verstanden. Unter einer *definierbaren Reaktion* ist jede mögliche Antwort der Applikation auf das Eintreten einer Situation zu verstehen, z. B. das Ausführen eines bestimmten Programmstücks oder Änderungen in der Datenbank.

Ursprünglich wurde der Begriff der „aktiven Datenbank“ erstmals im Zusammenhang mit dem Update von Views und der automatischen Konsistenzsicherung eingeführt [Mor83]. Im weiteren Verlauf wurden aktive Datenbanksysteme zusätzlich eingesetzt, um Überwachungs- und Alarmfunktionalitäten ohne unnötigen Overhead, z. B. für häufiges Polling des Datenbankzustandes, zu realisieren. Seitdem ist aktive Funktionalität in zahlreiche, sowohl relationale, als auch objektorientierte Datenbanken integriert worden, beispielsweise POSTGRES [SRH90], Starburst [WF92], SAMOS [GD92] und Ode [GJ91].

Die wichtigsten Vorteile einer aktiven Datenbank sind:

- Konsistenzinformation wird zentral in der Datenbank gespeichert und nicht redundant verteilt in unterschiedlichen Applikationsteilen,
- durch die zentrale Speicherung der Konsistenzinformation ist diese sicher immer für alle Clients identisch,
- häufiges Polling der Datenbank durch Clients zum Abprüfen auf bestimmte Situationen ist überflüssig.

Regeln

Grundsätzlich existieren zwei verschiedene Möglichkeiten, aktive Funktionalität in einer Datenbank zu definieren [HW91]:

- *musterbasierter Ansatz*, d. h. Definition von Regeln, die falls sie erfüllt sind, eine Aktion auslösen und
- *ereignisbasierter Ansatz*, d. h. ein definiertes Ereignis löst, falls eine Bedingung erfüllt ist, eine Aktion aus.

Der musterbasierte Ansatz weist zwei gravierende Probleme auf: Er ist schwierig effizient zu implementieren (Mustererkennung) und es ist nicht eindeutig definiert, im Kontext welcher Datenbankoperationen die einzelnen Regeln jeweils ausgewertet werden sollen. Zusätzlich ist die Definition von Zeitanforderungen schwierig, da die explizite Zuordnung zu einem Ereignis fehlt. Im ereignisbasierten Ansatz gibt es diese Probleme nicht. Hier wird die aktive Funktionalität in Form von sogenannten Event-Condition-Action-Regeln (ECA-Regeln) mit der folgenden allgemeinen Semantik festgelegt:

DEFINE RULE *ruleName*
ON *Event*
IF *Condition*
DO *Action.*

Zum heutigen Zeitpunkt sind ECA-Regeln das am weitesten verbreitete Paradigma zur Definition aktiver Datenbankfunktionalität.

Ereignisse (Event) in aktiven Datenbanken werden in primitive und zusammengesetzte Ereignisse unterteilt. Primitive Elemente sind das Grundelement, aus denen zusammengesetzte Ereignisse durch logische Verknüpfung zusammengesetzt werden können. Primitive Ereignisse können in folgende Klassen eingeteilt werden: Datenbankereignisse (z. B. insert, update, delete), temporale Ereignisse, DBMS-Ereignisse (Datenbank Management System Ereginisse, z. B. Transaktionsstart, -abort, -commit), abstrakte Ereignisse, d. h. Ereignisse, die außerhalb der Datenbank stattfinden und dieser mitgeteilt werden müssen. Abbildung 2.2 faßt die möglichen Ereignistypen noch einmal zusammen.

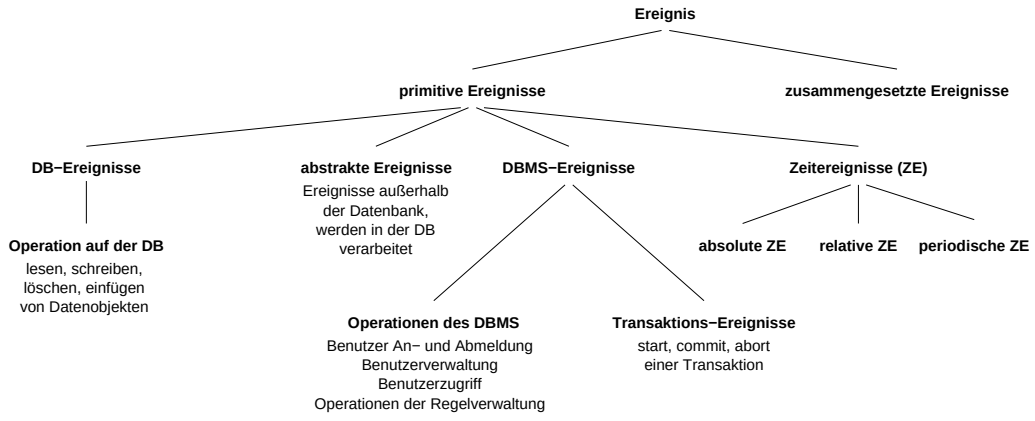


Abbildung 2.2.: Mögliche Ereignistypen in einem traditionellen aktiven Datenbank Management System (ADBMS).

Eine *Bedingung (Condition)* in einer aktiven Datenbank ist ein Prädikat über den Datenbankzustand. Eine *Aktion (Action)* ist ein beliebiges ausführbares Programmstück, das wiederum

Operationen auf der Datenbank enthalten kann.

Ausführungssemantik

Kopplungsmodi Für die Verknüpfung zwischen dem Auftreten eines Ereignisses, der Evaluierung der Bedingung und dem Ausführen der Aktion existieren unterschiedliche Kopplungsmodi [DBM88]:

- *Immediate.* Beim Auftreten eines Ereignisses wird die aktuelle Transaktion suspendiert, die Bedingung sofort ausgewertet und gegebenenfalls die Aktion ausgeführt. Danach wird die auslösende Transaktion wieder aufgenommen.
- *Deferred.* Die Auswertung der Bedingung und das Auslösen der Aktion geschieht am Ende der auslösenden Transaktion, aber vor deren „Commit".
- *Detached (decoupled).* Die Auswertung der Bedingung und das Ausführen der Aktion erfolgt in einer getrennten Transaktion. Abhängig davon, ob die ausgelöste Transaktion völlig unabhängig von der auslösenden Transaktion durchgeführt wird, oder nur bei deren „commit", unterscheidet man zusätzlich: *causally independant* und *causally dependant.*

Um die Problematik der „real actions", also von nach außen sichtbaren, irreversiblen Aktionen der Datenbank, besser handhaben zu können (siehe Abschnitt 3.2.1), wurden außerdem zwei zusätzliche Ausführungsmodi definiert [BBKZ93]: *sequential causally dependant* (die ausgelöste Transaktion startet nicht vor dem „commit" der auslösenden Transaktion), *exclusive causally dependant* (beide Transaktionen laufen parallel, die ausgelöste Transaktion kann aber nur gemeinsam mit der auslösenden Transaktion erfolgreich beendet werden).

Dynamische Regeln Auf Grund des dynamischen Umfeldes einer aktiven Datenbank ist es üblicherweise notwendig, daß die Regelbasis dynamisch zur Laufzeit geändert werden kann, d. h. das Hinzufügen, Entfernen und Ändern von Regeln muß möglich sein. Die Regeln können somit nicht statisch in das DBMS compiliert werden.

Cascade Triggering Falls infolge von Ereignissen Aktionen ausgelöst werden, so können diese erneut Ereignisse für andere Regeln erzeugen und so fort. Im schlechtesten Fall entstehen so unendliche Zyklen. Bis heute existieren nur sehr wenige Arbeiten zu dieser Problematik [AWH92]. Eine Möglichkeit, solche Situationen zu vermeiden, ist zum Beispiel, ein Limit für die Tiefe solcher „Trigger-Ketten" zu setzen. Eine weitere Möglichkeit ist, kaskadierendes Triggern völlig zu verbieten. Dies hat allerdings unter Umständen einen inkonsistenten Datenbankzustand zur Folge.

Konflikt-Lösung Tritt ein Ereignis in der Datenbank auf, so muß entschieden werden, welche der eventuell mehreren zutreffenden Regeln auszuwerten sind und in welcher Reihenfolge. Bis jetzt gibt es dazu keine übereinstimmende Meinung in der Literatur [BFKM85][HW91]. Eine Möglichkeit ist die Vergabe von Prioritäten für die einzelnen Regeln [ACL91].

2.2.3. Aktive Realzeitdatenbanken

Aktive Realzeitdatenbanken versuchen, aktive Mechanismen in Realzeitdatenbanken zu integrieren und trotzdem weiterhin Rechtzeitigkeit zu garantieren. Das größte Problem besteht dabei

darin, auch für die Kopplung aus Regelwerk und Datenbank Vorhersagbarkeit zu realisieren.

Eine wichtige Quelle der Nicht-Vorhersagbarkeit sind die unterschiedlichen Kopplungsmodi. So wird z. B. bei „Immediate"-Kopplung die gesamte Zeit zur Ausführung der Regeln der auslösenden Transaktion zugeschlagen. Damit ändert sich die Worst-Case Execution Time (WCET) einer Transaktion dynamisch, was vom Scheduler zu berücksichtigen ist. Ebenso problematisch ist das Auftreten kaskadierend getriggerter Regeln, sowie die vorhersagbare Erkennung von komplexen, unter Umständen zusammengesetzten Ereignissen [GGD94][Mel98]. Kritisch sind auch dynamisch wechselnde Regelsätze. Ein Ansatz hierzu ist, in der Datenbank zwischen unterschiedlichen Modi mit konstantem Regelsatz umzuschalten [SRLR89].

In der Regel werden in aktiven Datenbanken weder Zeitanforderungen berücksichtigt, noch wird eine Garantie für die Einhaltung von Zeitbeschränkungen gegeben. Es existieren allerdings Erweiterungen zur ECA-Definition, die die Angabe von Zeitanforderungen erlauben, wie z. B. in HiPAC [CBB+89]:

DEFINE RULE *ruleName*
ON *Event*
IF *Condition*
DO COMPLETE *Action* **WITHIN** *t seconds.*

Zeitbedingungen können dabei allgemein entweder in bezug auf den Zeitpunkt des Auftretens eines Ereignisses oder in bezug auf den Zeitpunkt der Ereignisdetektion festgelegt werden. Diese Zeiten können sich, je nach Komplexität des Ereignisses und der Mächtigkeit des Rule-Managers, ganz erheblich unterscheiden.

Unter den ersten Projekten, die die Integration von aktiver Funktionalität und Realzeitanforderungen zum Ziel hatte, war HiPAC [CBB+89]; hier ist „Realzeit" jedoch nur als „future work" angedacht. Drei Forschungsprojekte der neueren Zeit sind REACH [BBKZ93], STRIP [AKGM96] und DeeDS [AHE+96]. REACH ist der Prototyp einer diskbasierten aktiven Realzeitdatenbank. STRIP ist eine hauptspeicherbasierte Datenbank für weiche Zeitanforderungen. Der Datenbankprototyp DeeDS unterstützt reaktive Mechanismen, Verteilung und dynamisches Scheduling bei harten und weichen Zeitbedingungen. Der Ansatz ist hier, existierende Komponenten zu verwenden und nur die für die neue Funktionalität notwendigen Teile neu zu implementieren. Weitgehende Vorhersagbarkeit wird durch die Verteilung auf zwei Prozessoren erreicht.

Inzwischen existiert auch eine Anzahl an kommerziellen Implementierungen von aktiven Realzeitdatenbanken. Augenblicklich ist jedoch auf Grund fehlender Vorhersagbarkeit keine davon für den Einsatz in harten Echtzeitsystemen geeignet [Blö00].

2.3. Zielsetzungen der Arbeit

Der Großteil bisher existierender Arbeiten beschäftigen sich entweder mit der Entwicklung von Realzeitdatenbanken oder mit dem Realzeitnachweis von eingebetteten Systemen. Bei der Entwicklung von Datenbanken liegt der Schwerpunkt insbesondere auf Protokollen, die die Priorisierung von Transaktionen erlauben und die auf die Erzielung einer möglichst hohen Systemperformance hin optimiert sind: Realzeit im häufig mißverstandenen Sinne von „schnell". Geeignet sind diese Verfahren vor allem für weiche Realzeitsysteme. Nur sehr wenige Arbeiten existieren zu Systemen mit harten Zeitanforderungen und zum expliziten Nachweis über die Einhaltung von Zeitbedingungen: Realzeit im korrekten Sinne von „rechtzeitig". Ein solcher Nachweis ist

beim Einsatz zeitkritischer Systeme in sicherheitsrelevanten, harten Realzeitsystemen aber unbedingt notwendig. Nicht behandelt wird bis heute auch die Integration der Datenbank in die Applikation. Durch die Kopplung der beiden Systemteile entstehen zusätzliche Probleme für den Realzeitnachweis solcher Applikationen.

In dieser Arbeit wird daher eine Methode vorgestellt, die auch für eingebettete Realzeitsysteme mit einem Prozessor, die eine aktive Realzeitdatenbank (ARTDB) verwenden, den Realzeitnachweis führen kann. Das vorgestellte Verfahren baut auf einem für den Anwendungsbereich optimierten Modell einer hauptspeicherbasierten Datenbank auf. Es basiert auf einem in dieser Arbeit neu entwickelten semantischen Concurrency-Control Protokoll (PRED-DF) und unterstützt einen reduzierten, an die Anwendungsdomäne angepaßten Satz aktiver Funktionalität.

Dabei werden im einzelnen die folgenden Arbeiten geleistet:

Anforderungsanalyse Analyse der Anforderungen an eine aktive Realzeitdatenbank in der Anwendungsdomäne *datendominierte Realzeitsysteme*. Die Anforderungsanalyse basiert auf einem allgemeinen, abstrakten Systemmodell. Kritische und verzichtbare Funktionalität werden analysiert (Kapitel 3). Fragen der temporalen Konsistenz werden betrachtet und eine Konsistenzbedingung für abgeleitete Datenobjekte definiert.

Datenbankmodell Aufbauend auf der Anforderungsanalyse wird ein neues, für die Anwendungsdomäne optimiertes Modell einer aktiven Realzeitdatenbank entwickelt (Kapitel 4). Dabei werden insbesondere die folgenden Kernpunkte behandelt:

- *Definition des Concurrency-Control Protokolls PRED-DF.* PRED-DF nutzt Wissen über die Transaktionen und den Datenfluß in der Applikation, um Vorhersagbarkeit des Datenbankverhaltens und eine Minimierung der Blockierzeiten in der Datenbank zu erreichen. Dadurch ist ein genauer Realzeitnachweis möglich, der die benötigte Prozessorzeit sehr knapp, d. h. wenig pessimistisch kalkulieren kann.

- *Definition des Wissens- und Ausführungsmodells für den aktiven Datenbankanteil.* Die aktive Funktionalität der Datenbank dient zur Sicherung der logischen und temporalen Konsistenz des Datensatzes und zur Benachrichtigung der Applikation über besondere Datenbankzustände. Kritisch ist dabei besonders die Abarbeitung der aus der Datenbank heraus ausgelösten Aktivitäten im Gesamtkontext der Applikation. Optimierungsmöglichkeiten im Ausführungsmodell werden untersucht.

- *Prototypische Implementierung des Datenbankmodells.* Das in dieser Arbeit entwickelte Datenbankmodell wurde prototypisch implementiert, evaluiert und die Ausführungszeiten gemessen. Durch die Implementierung der Datenbank als Library wurde eine enge und effiziente Kopplung zwischen Datenbank und Applikation erreicht.

Realzeitnachweis Integration des Datenbankmodells in ein Verfahren zum Realzeitnachweis der gesamten Applikation. Analyse und Bewertung des Datenbankmodells aus Sicht des Realzeitnachweises für harte Realzeitsysteme (Kapitel 5).

Anwendungsbeispiele Exemplarisch wird der Einsatz der in dieser Arbeit entwickelten Techniken zum Datenmanagement für eingebettete, harte Realzeitsysteme in einer Fallstudie gezeigt. Zusätzlich wird das Potential des Datenbankmodells an Hand einer realen Anwendung in der Industrie (Software für Meßgeräte) aufgezeigt (Anhang A).

Abbildung 2.3 gibt abschließend einen graphischen Gesamtüberblick über die Strukturierung der Arbeit und die Zuordnung der Themenbereiche zu den einzelnen Kapiteln.

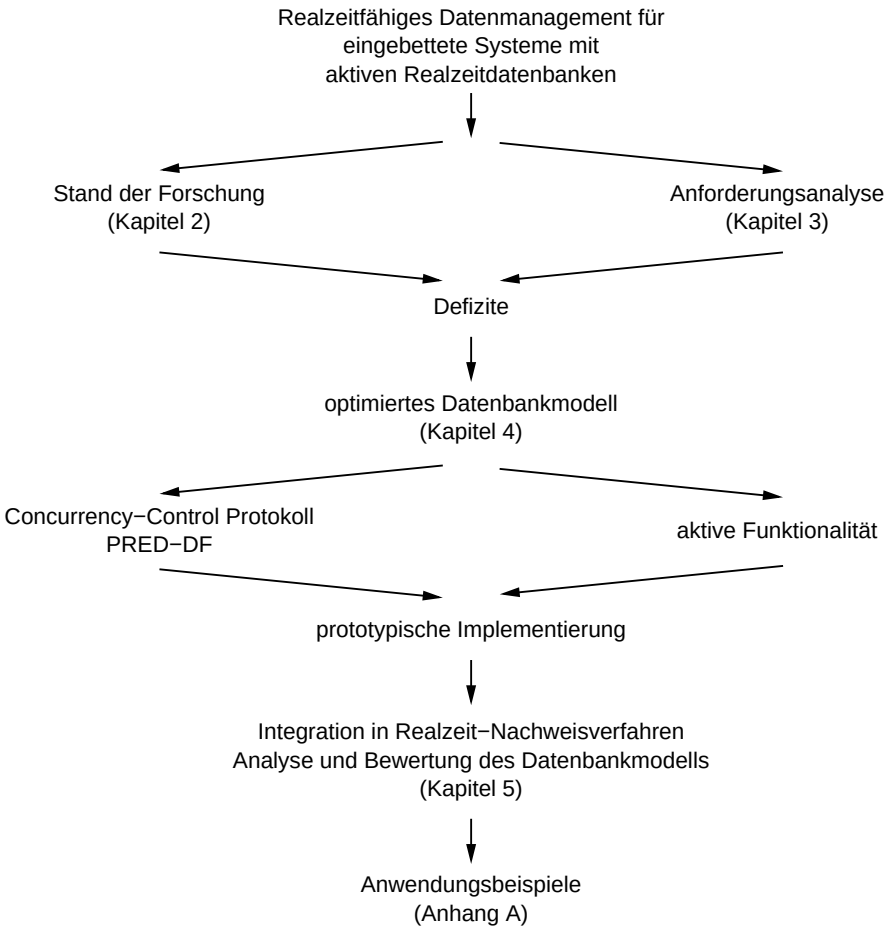


Abbildung 2.3.: Überblick über Inhalt und Strukturierung der vorliegenden Arbeit.

Nicht Gegenstand dieser Arbeit sind Aspekte des Logging, Undo, Recovery, DB-Abfragesprachen und die graphische Spezifikation von ECA-Regeln, da diese Probleme für die in dieser Arbeit behandelten Kern-Fragestellungen von untergeordneter Bedeutung sind. Hier sei auf die entsprechende Literatur verwiesen [Pat98][LC87][Hag87][LS92][EL95].

3. Anforderungsanalyse

3.1. Systemcharakteristika

In dieser Arbeit werden eingebettete Realzeitsysteme betrachtet, die auf Grund ihrer Komplexität und des Umfangs ihrer Datensätze (z. B. zahlreiche Umwelt- und Sensordaten) datenbankbasiert implementiert werden. Beispiele hierfür sind Anwendungen in der Produktionssteuerung und -regelung, Verkehrs- und Luftfahrtkontrollsysteme, Navigationssysteme und Autonome Mobile Systeme. Die Vorteile, die sich daraus für Systemarchitektur und -entwicklung ergeben, wurden bereits in Kapitel 1 kurz diskutiert. Diese Vorteile rechtfertigen auch den durch die Datenbank verursachten, erhöhten Ressourcenverbrauch. Allgemein werden solche Anwendungen als *datendomierte Realzeitsysteme (data-dominated real-time systems)* bezeichnet. Applikationen dieser Systemklasse sind durch folgende, charakteristische Eigenschaften gekennzeichnet:

1. *Umfangreicher Datensatz.* Die Anwendung verfügt über einen komplexen und umfangreichen Datensatz. Die Zahl der Datensätze in der Datenbank ist bekannt oder es existiert eine obere Grenze dafür.

2. *Konkurrierende Datenzugriffe.* Die Daten des Systems werden von unterschiedlichen, unabhängigen Systemteilen parallel genutzt.

3. *Redundante Datenobjekte.* Der Datenbestand enthält redundante Informationen in der Form, daß zwischen den einzelnen Datensätzen logische Abhängigkeiten bestehen. Dies hat im wesentlichen zwei Ursachen: Erstens benötigen unterschiedliche Systemteile (z. B. Komponenten) in der Regel ihre Eingangsgrößen in verschiedenen Darstellungen und Formaten, die alle in der Datenbank vorgehalten werden müssen. Zweitens werden Rechenergebnisse, die auf Datenbankeinträgen beruhen und die auch von anderen Systemkomponenten benötigt werden, ebenfalls in der Datenbank abgelegt [MBFW99] [BMW00]. Eine Transformation in eine redundanzfreie Darstellung ist in der Regel nicht sinnvoll, da in diesem Fall jeder einzelne Applikationsteil über das Wissen verfügen müßte, wie aus dieser redundanzfreien Darstellung die von ihm benötigten Daten gewonnen werden können. Das Wissen über die Konsistenzbeziehungen zwischen den Datenobjekten ist aber oft nicht jedem einzelnen Benutzer der Datenbank bekannt bzw. zugänglich. Zusätzlich sind viele Verfahren zur Berechnung von abgeleiteten Datensätzen nicht umkehrbar (z. B. keine eindeutige Umkehrfunktion einer mathematischen Formel). Diese Umkehrungen werden aber benötigt, falls von zwei abhängigen Datensätzen nur einer in der Datenbank gespeichert wird. Eine redundanzfrei Darstellung des Datenbestandes ist daher nicht möglich.

4. *Geringer Replikationsgrad.* Die Daten haben geringen Replikationsgrad, d. h. es existieren in der Regel viele unterschiedliche Datensätze aber keine großen Tabellen mit gleich strukturierten Datensätzen unterschiedlichen Inhalts.

5. *Überwachung von Daten.* Daten müssen, z. B. auf die Einhaltung von Grenzwerten hin, überwacht werden.

6. *Definierte Realzeitanforderungen.* Die Applikation und damit die Datenhaltung unterliegt definierten Zeitanforderungen, d. h. es existieren vorgegebene Deadlines für die Bearbeitung von Transaktionen auf der Datenbank.

7. *Temporale Konsistenz.* Das Bild des Systems von seiner physikalischen Umwelt muß so exakt sein, daß auf dessen Grundlage stets korrekt reagiert werden kann (Aktualität von Sensordaten, temporale Konsistenz).

Allgemein strukturieren sich datendominierte Realzeitsysteme entsprechend dem in Abbildung 3.1 dargestellten, abstrakten Systemmodell.

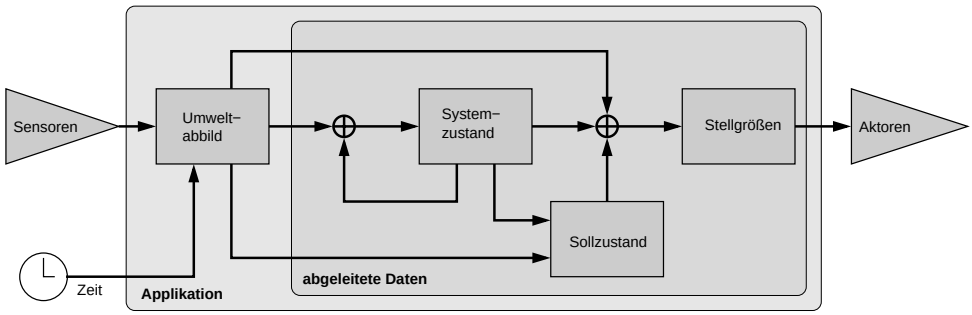


Abbildung 3.1.: Abstraktes Systemmodell datendominierter Realzeitsysteme.

Auf der Eingangsseite der Anwendung befinden sich *Sensoren*, die Informationen über die Umwelt des eingebetteten Systems aufnehmen und die Systemumgebung in Form von Datenobjekten in der Applikation abbilden (*Umweltabbild, Umweltdaten*). Aus diesem Umweltabbild und dem augenblicklichen Systemzustand wird ein neuer, nun gültiger *Systemzustand* abgeleitet. Dieser wird mit dem aktuellen *Sollzustand*, der sich aus dem momentanen Systemzustand und dem Umweltabbild errechnet, verglichen. Ergibt sich eine Abweichung, so werden entsprechende *Stellgrößen* ermittelt, um diese Abweichung zu korrigieren. Die Stellgrößen wirken über die *Aktoren* des Systems korrigierend auf die Umwelt der Applikation ein.

Das Umweltabbild setzt sich aus *kontinuierlichen* und *diskreten Datenobjekten* und der Systemzeit zusammen. Kontinuierliche Datenobjekte (z. B. die Position eines Flugzeuges über Grund) repräsentieren Eigenschaften der physikalischen Systemumgebung, die sich kontinuierlich über die Zeit ändern. Sie verlieren im Laufe der Zeit ihre Gültigkeit, d. h. sie „altern“. Diskrete Datenobjekte dagegen verlieren ihre Aktualität nicht automatisch mit dem Fortschreiten der Zeit, sondern sie bilden Größen der Umgebung ab, deren Wert sich spontan jeweils zu einem bestimmten Zeitpunkt ändert. Alle Daten einer Applikation, die aus Objekten des Umweltabbildes berechnet werden (z. B. der Systemzustand), werden als *abgeleitete Datenobjekte* bezeichnet.

Entsprechend den Anforderungen der Applikation ist ein Teil der Daten aus dem Umweltabbild, Systemzustand, Sollzustand, Stellgößen in der Datenbank gespeichert. Dadurch ergibt sich ein Teil der bereits erwähnten Abhängigkeiten innerhalb der Datenbank.

Zeitanforderungen werden durch das Festlegen einer maximalen Zeitspanne (Deadline), die zwischen dem Eintreten eines externen Ereignisses und der Reaktion durch die Applikation verstreichen darf, definiert. Entsprechende externe Ereignisse sind entweder das Eintreten eines bestimmten Zeitpunktes oder die Werteänderung eines Datums im Umweltabbild.

Aus den hier dargestellten Systemcharakteristika ergeben sich die folgenden Schlüsselanforderungen an eine Datenbank für eingebettete, datendominierte Realzeitsysteme:

a) *Richtigkeit.* Die Datenbank muß die parallele Bearbeitung von Operationen unterschiedlich hoher Priorität in der Datenbank unterstützen und so kontrollieren, daß das resultierende Ergebnis logisch korrekt ist (Abschnitt 3.2). Die Parallelität wird dabei aus der Umwelt über die nebenläufige Struktur der Applikation in die Datenbank übertragen, da unterschiedliche, parallel ablaufende Teile einer Applikation gleichzeitig auf die Datenbank zugreifen.

b) *Rechtzeitigkeit.* Die rechtzeitige Bearbeitung von Operationen auf der Datenbank innerhalb einer definierten Deadline muß unterstützt werden. Werden alle Deadlines eingehalten, so kann auch temporale Konsistenz sichergestellt werden (Abschnitt 3.2).

c) *Konsistenzsicherung.* Die Konsistenz abhängiger Datensätze muß durch die Datenbank soweit wie möglich selbständig sichergestellt werden können. Die Konsistenzinformation hierfür muß zentral in der Datenbank gespeichert sein (Abschnitt 3.3).

d) *Ereignisdetektion.* Das Auftreten bestimmter Zustände im Systemwissen muß von der Datenbank selbst erkannt werden (Abschnitt 3.3). Damit kann z. B. das Abprüfen auf Abweichungen vom Sollzustand und das automatisierte Berechnen von Stellgrößen in der Datenbank durchgeführt werden (vermeiden von Polling).

Zusätzlich sind die folgenden beiden Randbedingungen bei der Realisierung der oben aufgezählten Anforderungen zu berücksichtigen; zum einen, um eine möglichst störungsfreie Integration der Datenbank in die Applikation zu ermöglichen, und zum anderen, um den Ressourcenverbrauch und damit die Kosten möglichst gering zu halten:

- *Optimalität.* Das Scheduling von Tasks durch das Realzeit-Betriebssystem darf von der Sicherung der Datenkonsistenz durch die Datenbank so wenig wie möglich beeinflußt werden (z. B. Blockierung von Transaktionen bzw. Tasks).
- *Effizienz.* Obwohl Performance nicht mit Realzeitfähigkeit verwechselt werden darf, ist ein sparsamer Umgang mit der Ressource CPU notwendig. Das heißt, der Ressourcenverbrauch der Datenbank, sowie die Kosten und die Häufigkeit von „abort“ und „rollback“ durch das System müssen minimiert werden („predictable performance“). Auch bei der Realisierung der aktiven Datenbankfunktionalität sollte die Ressourcennutzung optimiert werden.

3.2. Richtigkeit und Rechtzeitigkeit

Die Anforderungen „Richtigkeit“ und „Rechtzeitigkeit“ werden durch das Transaktionskonzept und das Concurrency-Control Protokoll (CC-Protokoll) realisiert. Temporale Konsistenz wird in Abschnitt 3.2.3 genauer behandelt.

3.2.1. Transaktionen

Benutzeroperationen auf der Datenbank werden in Transaktionen (TAs) zusammengefaßt. Der Nutzer ist dabei für die in seinem Sinne logisch korrekte Bearbeitung der Daten verantwortlich; die Datenbank für die korrekte Abwicklung einer Transaktion in der Datenbank auch bei Systemfehlern. Um dies sicherzustellen, sind an die Transaktionsverarbeitung folgende, grundlegende Anforderungen zu stellen (ACID-Kriterien):

- Atomarität (atomicity). Entweder alle oder keine Operationen einer TA werden in der Datenbank festgehalten („Alles oder Nichts"-Prinzip)[1].
- Konsistenz (consistency). TAs sind die „Einheit" der Datenbankkonsistenz. Für die logische Konsistenz einer einzelnen TA ist der Benutzer verantwortlich.
- Isolation (isolation). Sicherstellung der Konsistenz einer TA trotz gleichzeitiger, paralleler Abarbeitung von mehreren TAs in der Datenbank.
- Dauerhaftigkeit (durability). Ergebnisse einer abgeschlossenen TA sind dauerhaft in der Datenbank gespeichert.

In Realzeitsystemen müssen einzelne Transaktionen entsprechend ihrer Wichtigkeit mit einer Priorität versehen werden können, um Rechtzeitigkeit bei der Transaktionsbearbeitung sicherzustellen.

Allgemein werden Transaktionen vom Benutzer (hier der Applikation) gestartet und beendet. Der Abbruch einer Transaktion (abort und rollback) kann normalerweise vom Benutzer oder von der Datenbank, z. B. zur Auflösung eines TA-Konfliktes, veranlaßt werden. In eingebetteten Realzeitsystemen sind Abbrüche von Transaktionen durch die Datenbank allerdings ungünstig und unter bestimmten Umständen ausgeschlossen: i) Zum einen müssen solche Abbrüche durch die betroffenen Transaktionen selbst aufwendig detektiert werden. Die bisher aufgewendete CPU-Zeit geht verloren. Sie sind daher aus Effizienzgründen ungünstig. ii) Zum anderen können Aktionen innerhalb einer Transaktion, z. B. Einstellungen an der Systemhardware, unter Umständen nicht mehr rückgängig gemacht werden. Ein „rollback" der Transaktion nach dem „abort" ist in diesen Fällen unmöglich (real actions [GR93]). iii) Zusätzlich problematisch ist der Abbruch von Transaktionen für die Garantie von temporaler Konsistenz innerhalb der Datenbank [SL95]. Kontrollierte Abbrüche einer Transaktion durch den Benutzer müssen jedoch immer möglich sein.

Transaktionen in traditionellen Datenbanken haben stark unterschiedliche Ausführungspfade und stark unterschiedliche Ressourcenanforderungen. Dies hat seine Ursache in der direkten Interaktion zwischen Anwender und Datenbank, die es erlaubt, dynamisch immer neue Transaktionen mit unterschiedlichen Inhalten zu generieren. In eingebetteten Realzeitsystemen dagegen werden die Operationen einer Transaktion bereits zur Zeit der Systementwicklung festgelegt. Interaktionen zwischen Anwender und Datenbank sind – wenn überhaupt – nur über genau definierte, vorher festgelegte Dialoge möglich, die der Anwender nur noch zu parametrieren hat (canned transactions [DMK+96]). Transaktionen besitzen daher einen genau definierten Inhalt und die Lese- und Schreibdatensätze sind bereits vor der Laufzeit der Anwendung bekannt.

3.2.2. Concurrency-Control Protokoll

Es ist nicht sinnvoll, die eintreffenden Transaktionen am Eingang der Datenbank hart nach dem First-Come-First-Served Prinzip zu serialisieren [Day99], da so unnötig lange Blockierzeiten auch für solche Transaktionen auftreten, die sich nicht im Konflikt befinden, und Einschränkungen bei der Parallelität im System in Kauf genommen werden müssen.

Allerdings können bei nicht synchronisierter, paralleler Bearbeitung von Transaktionen folgende Anomalien auftreten, d. h. falsche Ergebnisse werden am Ende einer Transaktion in die Datenbank eingetragen (siehe Abbildung 3.2): *Lost Update* (zwei unterschiedliche Transaktionen

[1] Damit erfolgt der Abgleich bezüglich der unterschiedlichen Grade der Atomarität aus Benutzersicht (atomare Einheit: Transaktion) und aus der Sicht der Datenbank (atomare Einheit: Lese/Schreib-Operation).

ändern dasselbe Datum, eine Änderung geht verloren), *Dirty Read* (eine Transaktion liest eine temporäre Version eines Datums, das von einer parallel ablaufenden, noch nicht abgeschlossenen Transaktion gerade bearbeitet wird), *Unrepeatable Read* (eine Transaktion liest, auf Grund einer gleichzeitig parallel arbeitenden Transaktion, unterschiedliche Werte für dasselbe Datum aus der Datenbank).

Lost Update	**Dirty Read**	**Unrepeatable Read**
TA2 read(A)	TA2 write(A)	TA1 read(A)
TA1 write(A)	TA1 read(A)	TA2 write(A)
TA2 write(A)	TA2 write(A)	TA1 read(A)

Abbildung 3.2.: Mögliche Anomalien bei gleichzeitiger, unsynchronisierter Ausführung zweier Transaktionen auf der Datenbank.

Wie bereits in Abschnitt 2.2.1 erwähnt, besteht die Aufgabe eines Concurrency-Control Protokolls darin, bei der gleichzeitigen Bearbeitung mehrerer Transaktionen in einem Schedule, die Operationen der einzelnen Transaktionen auf der Datenbank so zu synchronisieren, daß trotz Parallelität die logische Konsistenz der Datenbank sichergestellt ist, d. h. die obigen Anomalien ausgeschlossen sind (Korrektheit des Schedules).

Allgemein anerkannt und am weitesten verbreitet als Kriterium für die Korrektheit eines Schedules ist die Serialisierbarkeit [EGLT76]. Serialisierbarkeit verlangt, daß die parallele Ausführung von Transaktionen zu einer beliebigen seriellen Ausführung dieser Transaktionen äquivalent ist. Die Korrektheit eines serialisierbaren Schedules läßt sich dann unter der Annahme, daß jede einzelne Transaktion für sich die Datenkonsistenz sicherstellt, durch vollständige Induktion leicht beweisen. Auf Grund der Einfachheit und der offensichtlichen Richtigkeit wird Serialisierbarkeit[2] auch in dieser Arbeit als Korrektheitskriterium gefordert.

Problematisch in Realzeitsystemen ist, daß durch die hohen Ansprüche, die die Serialisierbarkeit an die Synchronisation von Transaktionen stellt, die parallele Bearbeitung von Transaktionen auf der Datenbank stark eingeschränkt ist (z. B. bei der Aktualisierung von Datenwerten durch Update-Transaktionen). Dies kann durch Protokolle, die das Korrektheitskriterium Serialisierbarkeit aufweichen (z. B. ϵ-Serialisierbarkeit, siehe Abschnitt 2.2.1) und damit einen höheren Grad an Parallelität auf der Datenbank zulassen, umgangen werden. Allerdings sind diese für den hier betrachteten Anwendungsbereich aus folgenden Gründen nicht geeignet:

- Die Vorhersage des Datenbankverhaltens, z. B. der Worst-Case Execution Times von Transaktionen, ist nur sehr eingeschränkt bzw. unter sehr pessimistischen Annahmen möglich. In der Regel ist nicht festgelegt, wann Transaktionen ausgeführt, blockiert oder abgebrochen werden, da dies nun vom Systemzustand zum Zeitpunkt des Transaktionsstarts und somit von der Vorgeschichte des Systems und der Veränderung der Umwelt in der Vergangenheit abhängt. Das Verhalten des Concurrency-Control Protokolls zur Laufzeit muß aber so genau wie möglich vorhersagbar sein, um in Verbindung mit der Priorisierung der Transaktionen und einem geeigneten Verifikationsalgorithmus Rechtzeitigkeit garantieren zu können. Des weiteren erhöhen diese Protokolle zwar die Parallelität auf der Datenbank, vermindern damit Blockierzeiten und verbessern die Performance, sie haben aber auch einen erhöhten Verwaltungsaufwand, der diese Vorteile weiter schmälert (Effizienz).
- Die Schedules dieser Protokolle sind nicht serialisierbar. Daher fällt eine wesentlich höhere Verantwortung für korrekte Systemfunktionalität, d. h. die Richtigkeit aller potentiell

[2] genauer: Konflikt-Serialisierbarkeit

möglichen Schedules, dem Entwickler zu. Die Konsequenzen seiner Designentscheidungen (z. B. beim Aufstellen von Kompatibilitätslisten) sind für den Entwickler nur schwer zu übersehen. Zudem müssen sie nach jeder Änderung am System nochmals überprüft werden. Dies ist insbesondere in sicherheitskritischen Systemen äußerst problematisch. Gleichzeitig sind diese Aufgaben nur eingeschränkt automatisierbar.

Zusammenfassend werden daher für ein geeignetes Concurrency-Control Protokoll in dieser Arbeit folgende Eigenschaften gefordert:

- Konflikt-Serialisierbarkeit der Schedules,
- kein Abbruch von Transaktionen und
- Vorhersagbarkeit
- bei gleichzeitig möglichst großer Optimalität und Effizienz.

Ein Concurrency-Control Protokoll, das diese Eigenschaften optimal für eingebettete Realzeitsysteme kombiniert, existiert im Augenblick nicht.

3.2.3. Temporale Konsistenz

Kontinuierliche Daten

Realzeitapplikationen arbeiten auf Grund des jedem Prozessor zugrundeliegenden diskreten Zeitmodells auf einem zeitlich versetzten Abbild der kontinuierlichen Umwelt („Umweltabbild“ in Abbildung 3.1). Zu jedem Zeitpunkt muß dieses Bild den Zustand der Umwelt so exakt widerspiegeln, daß eine korrekte Funktionalität der Anwendung gewährleistet ist. Dies bezieht sich insbesondere auf die Aktualität der kontinuierlichen Datenobjekte in der Datenbank und die Frage, wie die Aufnahmezeitpunkte der einzelnen Daten zeitlich relativ zueinander liegen. Das heißt, in einer Realzeitdatenbank muß nicht nur logische Konsistenz und Vorhersagbarkeit der Transaktionsbearbeitung garantiert werden. Zusätzlich müssen der Applikation durch rechtzeitiges „Erneuern“ von Umwelt- und abgeleiteten Daten auch zeitlich aktuelle Eingangsgrößen zur Verfügung gestellt werden. Ist die Übereinstimmung zwischen dem Umweltabbild und der physikalischen Realität der Systemumgebung entsprechend den Anforderungen der Applikation gegeben, so spricht man von *temporaler Konsistenz*.

Umweltdaten In der Regel wird die temporale Konsistenz von kontinuierlichen Umweltdaten über die zwei Kenngrößen *Alter* und *Dispersion* definiert [Ram93][SL95]:

Definition 1 (Alter kontinuierlicher Umweltdaten) *Das Alter $a(x_i)$ eines Umweltdatums x_i gibt die Zeitdifferenz zwischen dem Aufnahmezeitpunkt $t_s(x_i)$ durch den zugeordneten Sensor und dem Zeitpunkt t an, zu dem der Wert von x_i aus der Datenbank gelesen wird:*

$$a(x_i) := t - t_s(x_i). \tag{3.1}$$

Falls das Alter eines Datenobjektes x_i stets unter einer applikationsspezifischen Schwelle T_i liegt, d. h. $a(x_i) \leq T_i$, wird das Datum x_i als *absolut temporal konsistent* bezeichnet. Es gilt $t_s(x_i) \in [t - T_i; t]$.

Definition 2 (Dispersion kontinuierlicher Umweltdaten) *Die Dispersion $\delta(x_1, x_2)$ charakterisiert die zeitliche Korrelation zweier Datenobjekte des Umweltabbilds x_i und x_j:*

$$\delta(x_i, x_j) := |a(x_i) - a(x_j)| = |t_s(x_i) - t_s(x_j)|. \tag{3.2}$$

Eine Menge von Daten $\{x_1, x_2, \ldots, x_m\}$ wird als *relativ temporal konsistent* bezeichnet, falls für alle t gilt $\delta(x_i, x_j) \leq T'(x_i, x_j)$, $\forall x_i, x_j \in \{x_1, x_2, \ldots, x_m\}$. $T'(x_i, x_j)$ bezeichnet dabei eine vom Systemdesigner festzulegende Schwelle für den maximal zulässigen Wert der Dispersion dieser Datenobjekte.

Abgeleitete Datenobjekte Bei abgeleiteten kontinuierlichen Datenobjekten ist die Definition der temporalen Konsistenz schwieriger. In der Literatur wird das Alter eines abgeleiteten Datenobjektes y meist über das Alter der für die Berechnung verwendeten Ausgangsdaten definiert [Ram93][SL95][TVC00]:

$$a(y) := \max[\, a(x_1), a(x_2), \ldots, a(x_n)\,], \text{ falls } y = f(x_1, x_2, \ldots, x_n). \tag{3.3}$$

Diese Definition führt allerdings zu Problemen, falls ein Datenobjekt y von mehreren Umweltdaten abhängig ist, und man durch die Definition einer Schwelle T_y die zeitliche Konsistenz sicherstellen möchte bzw. den Grad der zeitlichen Konsistenz charakterisieren möchte:

1. Angenommen ein Datum y hängt von x_1 und x_2 ab. x_1 und x_2 ändern sich unterschiedlich rasch über die Zeit und werden daher regelmäßig alle 50 bzw. 150 Zeiteinheiten aktualisiert. Deadline ist die Periodendauer. Je nach dem Zeitpunkt der Betrachtung, ergibt sich so im schlechtesten Fall ein Alter von bis zu 300 Zeiteinheiten für y (siehe Abbildung 3.3). Egal welchen Wert man für $T_y < 300$ fordert, um die Aktualität von y zu garantieren (in der Abbildung z. B. $T_y = 100$), kann temporale Konsistenz nach der obigen Definition selbst dann nicht sichergestellt werden, falls y immer gemeinsam mit x_1 und x_2 aktualisiert wird. Es ist nach dieser Definition also möglich, daß ein Datum auch dann nicht absolut zeitlich konsistent ist, falls es auf der Grundlage temporal konsistenter Ausgangsdaten berechnet wurde.
2. Die alternativ mögliche Festlegung $T_y \geq 300$ verursacht zwar keine Verletzung der temporalen Konsistenzbedingung, die Aktualität von y kann damit allerdings nicht sichergestellt werden.

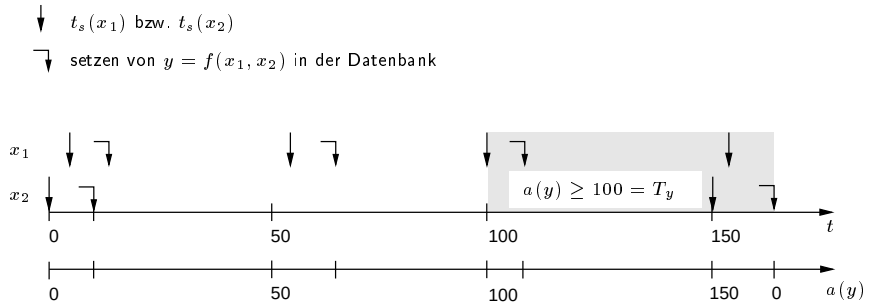


Abbildung 3.3.: Beispiel für die problematische Definition des Alters von abgeleiteten kontinuierlichen Datenobjekten in einer Realzeitdatenbank (siehe Text).

In dieser Arbeit wird daher das folgende Maß für die absolute temporale Konsistenz von abgeleiteten Datenobjekten definiert:

Definition 3 (absolute temporale Konsistenz abgeleiteter Datenobjekte) *Ist x_j ein zum Zeitpunkt t aus der Datenbank gelesenes, abgeleitetes Datenobjekt mit $x_j = f(x_1, x_2, \ldots, x_n)$, so ist x_j absolut temporal konsistent, falls für alle t gilt:*

$$a_j(x_i) := t - t_s(x_i) \leq T_{ij}, \quad \forall x_i \in \{x_1, x_2, \ldots x_n\}. \tag{3.4}$$

Hierbei bezeichnet $a_j(x_i)$ das Alter des Wertes von x_i, der für die Berechnung von x_j herangezogen wurde. Falls x_i ein Umweltdatum ist, gilt $a_j(x_j) = a(x_j)$.

Neben der absoluten temporalen Konsistenz eines abgeleiteten Datums können außerdem Forderungen an die zeitliche Dispersion der Datenobjekte $\{x_1, x_2, \ldots, x_n\}, y = f(x_1, x_2, \ldots, x_n)$, gestellt werden:

$$\begin{aligned} |a_y(x_i) - a_y(x_j)| &\leq T'_y(x_i, x_j),\ \forall x_i, x_j \in \{x_1, x_2, \ldots, x_n\}, \\ T'_y(x_i, x_j) &= T'_y(x_j, x_i). \end{aligned} \tag{3.5}$$

So können Systeme abhängiger Umweltdaten entstehen, wenn mehrere unterschiedliche Daten jeweils paarweise über Dispersionsrelationen miteinander verknüpft sind.

Die individuellen Schwellen T_{ij} und $T'_y(x_i, x_j)$ ergeben sich jeweils aus den Anforderungen der Applikation und sind vom Systementwickler zu bestimmen. Die genaue Festlegung ist in der Regel schwierig und nur mit profunder Kenntnis des betrachteten Systems möglich. Werte für $T_{x_i,y}$ können zum Beispiel mit Hilfe des Nyquist-Theorems und der linearen Systemanalyse abgeleitet oder mit Hilfe der folgenden Ungleichung näherungsweise grob abgeschätzt werden. Hierbei bezeichnet Δy_{x_i} die Abweichung von y, die auf Grund einer Änderung in x_i toleriert wird:

$$\max_t \left| \frac{\partial y}{\partial x_i} \frac{\partial x_i}{\partial t} \right| T_{x_i,y} \leq \Delta y_{x_i}. \tag{3.6}$$

Diskrete Daten

Für diskrete Datenobjekte macht die bisher vorgestellte Definition temporaler Konsistenz keinen Sinn, da sich diese Daten spontan zu einem unbestimmten Zeitpunkt ändern. Hier sind alternative Zeitbedingungen notwendig, die es z. B. erlauben, Bedingungen an die „Ausbreitungsgeschwindigkeit“ der neuen Information im System zu stellen (siehe hierzu Abschnitt 4.3.1).

3.3. Aktive Funktionalität

Wie in Abschnitt 3.1 dargestellt, dient die aktive Funktionalität einer Datenbank dazu, auf bestimmte, spezifizierbare Ereignisse in der Datenbank entsprechend reagieren zu können. Hierzu hat der Satz an Ereignistypen, die in einem klassischen, aktiven Datenbanksystem definiert werden können, normalerweise den in Abbildung 2.2, Abschnitt 2.2.2, dargestellten Umfang.

Die große Zahl unterschiedlicher Ereignistypen und insbesondere zusammengesetzte Ereignisse machen die zeitlich vorhersagbare Ereignisdetektion innerhalb der Datenbank sehr aufwendig und schwierig [Mel98][Eri97]. In der Regel ist die Implementierung eines eigenen Regelmanagers (Rule-Manager) und Ereignismonitors sowie eines eigenen Transaktions-Schedulers innerhalb der Datenbank selbst notwendig. Mit welcher Priorität detektierte Ereignisse bearbeitet werden und in welchem Verhältnis diese zu den Abläufen im Rest der Applikation stehen, ist nicht

klar zu definieren. Zudem ist ein eigener Datenbankscheduler aus Implementierungssicht und aus Performancegründen in der Anwendungsdomäne der eingebetteten Realzeitsysteme nicht wünschenswert (siehe Abschnitt 4.4). In dieser Arbeit dient die aktive Funktionalität der Datenbank zum einen dazu, die Konsistenz abhängiger Datensätze sicherzustellen und die dazugehörige Konsistenzinformation zentral zu speichern. Zum anderen ist die effiziente Überwachung einzelner Datensätze, z. B. auf das Überschreiten von Grenzwerten hin, zu ermöglichen. Direkte Interaktionsmöglichkeiten des menschlichen Benutzers mit der Datenbank existieren nicht.

Für die Anwendungsdomäne ist es daher ausreichend, die folgenden reduzierten Möglichkeiten zur Definition von ECA-Regeln aktiver Funktionalität zur Verfügung zu stellen:

Ereignis (Event). Ein Ereignis in der Datenbank ist das Schreiben eines Datenobjektes in der Datenbank, denn nur zu diesen Gelegenheiten kann sich der Inhalt der Datenbank substantiell ändern („An event is a happening of interest“ [GJS92]).

Zeitereignisse und abstrakte Ereignisse werden von der Applikation selbst mittels des unterliegenden Realzeit-Betriebssystems gehandhabt und müssen nicht von der Datenbank bearbeitet werden. Die Behandlung von Transaktionsereignissen würde, über die Speicherung von allgemeingültiger Konsistenzinformation bzw. von mit Datenobjekten verknüpften Operationen hinaus, die Definition von transaktionsspezifischen Regeln in der Datenbank erfordern. Solche Ereignisse werden direkt in der Applikation selbst behandelt. Auf die Behandlung zusammengesetzter Ereignisse durch die Datenbank wird aus Gründen der Komplexität und der Vorhersagbarkeit verzichtet. Sie können durch die Verwendung komplexerer Bedingungen im „Condition“-Teil einer ECA-Regel modelliert werden [BBKZ93].

Bedingung (Condition). Eine Bedingung ist entweder ein beliebiges Prädikat über den gegenwärtigen Datenbankzustand oder der Aufruf einer externen Funktion. Beidesmal muß der Rückgabewert eindeutig „wahr“ oder „falsch“ sein. Das Setzen einer Bedingung ist nicht zwingend notwendig.

Aktion (Action). Eine Aktion ist entweder eine Modifikation des Datenbankzustandes, d. h. das Schreiben und Lesen von einem oder mehreren Datenobjekten und/oder das Auslösen eines Ereignisses, das von der einbettenden Applikation behandelt wird. Eine Aktion wird immer dann ausgeführt, wenn ein Ereignis eingetreten ist und die Bedingung wahr ist. Eine ausgeführte Aktion kann selbst wieder der Auslöser einer neuen Regel sein.

Der vorgestellte Satz an aktiver Funktionalität hat sich als tragfähig für die hier betrachtete Anwendungsdomäne erwiesen. Eine exemplarische Anwendung zeigt die in Anhang A.1 durchgeführte Fallstudie. Er wird in ähnlicher Form auch erfolgreich in Projekten der Industrie (Firma Rohde & Schwarz) eingesetzt und dort in einer aktiven Datenbank verwendet, die das zentrale Element einer konfigurierbaren Basisarchitektur für komplexe Meßgeräte bildet [MBFW99]. (siehe Anhang A.2). In diesem Umfeld entstand auch der Prototyp eines Werkzeugs zur graphischen Regelspezifikation [Sch99].

Abschließend vergleicht Tabelle 3.1 das in dieser Arbeit unterstützte Modell einer aktiven Datenbank mit den im „Active Database Management System Manifesto“ [DGG96] dargestellten, zentralen Kernanforderungen. Trotz der Einschränkungen bei den Definitionsmöglichkeiten für unterschiedliche Ereignisse und Ereignistypen, die sich aus den besonderen Randbedingungen des Anwendungsgebietes ergeben, werden die dort aufgestellten Forderungen im Kern erfüllt.

ADBMS Manifesto	diese Arbeit
Ein aktives Datenbanksystem (ADBMS) hat alle Eigenschaften einer normalen Datenbank (insbesondere existieren ein Transaktionskonzept und ein Concurrency-Control Verfahren).	Für die betrachtete Anwendungsdomäne können hinsichtlich des Funktionsumfanges die in Abschnitt 3.4 dargestellten Einschränkungen gemacht werden. Ein Transaktionsmodell ist implementiert, Konflikt-Serialisierbarkeit der Schedules wird durch das Concurrency-Control Protokoll PRED-DF garantiert. PRED-DF ist ein Schwerpunkt dieser Arbeit.
Die aktive Datenbank unterstützt die Definition und das Management von Regeln, das heißt: • Es existieren Möglichkeiten zur Definition von Events, Conditions und Actions, • Unterstützung einer Regel-Basis und von dynamischen Regeln, das heißt, Regeln können zur Laufzeit hinzugefügt, geändert und entfernt werden.	Definition und Management von Regeln wird unterstützt: • Es besteht die Möglichkeit, ECA-Regeln zu definieren. Die Regelbasis kann zur Laufzeit dynamisch geändert werden. Es gibt keine zusammengesetzten Ereignisse. • Als Ereignistyp ist nur das Schreiben eines Datums zugelassen, da sich in der betrachteten Anwendungsdomäne nur dann der Informationsgehalt der Datenbank substantiell ändern kann. Alle anderen Ereignistypen werden von der Applikation behandelt (z. B. Zeitereignisse) oder sind nicht von Bedeutung (z. B. Operationen des DBMS). • siehe hierzu auch Abschnitt 3.3 und 4.3.1.
Ein ADBMS hat ein Ausführungsmodell, das heißt, es muß in der Lage sein: • das Auftreten von Ereignissen zu erkennen, • Bedingungen zu evaluieren, • Aktionen auszuführen. Dem ADBMS muß eine wohldefinierte Ausführungssemantik für zusammengesetzte Ereignisse und ein definierter Mechnismus zur Konfliktlösung zugrunde liegen.	• Es ist ein Abarbeitungsmodell zur Erkennung von Ereignissen, Evaluierung von Bedingungen und Ausführung von Aktionen definiert (siehe Abschnitt 4.3.2). • Zusammengesetzte Ereignisse werden nicht betrachtet. • Die Bearbeitung von Regeln ist indirekt über die Priorität der Transaktionen priorisiert.

Tabelle 3.1.: Vergleich zwischen dem im „Active Database Management System Manifesto“ [DGG96] geforderten und dem in dieser Arbeit unterstützen Funktionsumfang einer aktiven Datenbank.

3.4. Weitere Anforderungen

Datenbank-Abfragesprachen Im Kontext der in dieser Arbeit behandelten datendominierten Realzeitsysteme mit harten Deadlines sind komplexe Abfragesprachen, wie zum Beispiel RTSQL [PFW97], von untergeordneter Bedeutung. In der betrachteten Anwendungsdomäne werden keine komplexen Anfragen an die Datenbank formuliert. Der Replikationsgrad einzelner Datensätze in der Datenbank ist gering. Die Datenbank dient zur realzeitfähigen, konsistenten Verwaltung einzelner Datenobjekte als Ressourcen der Applikation. Funktionen zum Suchen, Filtern und Zusammenstellen von Informationen aus umfangreichen, gleich strukturierten Datensätzen werden in den Systemteilen mit harten Zeitanforderungen nicht benötigt.

Logging, Recovery, Durabilty In hauptspeicherbasierten Datenbanken ist es problematisch, die Dauerhaftigkeit von Transaktionsergebnissen sicherzustellen, da die Daten ausschließlich im Speicher abgelegt werden und ein Systemausfall damit automatisch den Verlust des gesamten Datensatzes bedeuten kann. Dennoch ist es häufig erwünscht, Informationen auch über das Ausschalten eines Gerätes hinaus bis zur nächsten Inbetriebnahme zu speichern. Gleichzeitig ist es nicht nötig, z. B. die Werte kontinuierlicher Umweltdaten über einen längeren Zeitraum zu sichern, da sie innnerhalb kurzer Zeit ohnehin ihre Gültigkeit verlieren. In der Regel ist es also ausreichend, Dauerhaftigkeit für einen Teil des gesamten Datensatzes zu sichern.

Zur Lösung dieses Problems sind entsprechende realzeitfähige Mechanismen für Logging und Recovery notwendig. Eine Möglichkeit besteht beispielsweise darin, wichtige Datensätze in batteriegepuffertem Speicher abzulegen. Darüber hinaus existieren unterschiedliche andere Ansätze. Allerdings wurden diese Fragestellungen bisher auch in der Literatur nur sehr wenig systematisch behandelt und sind gegenwärtig noch Gegenstand aktueller Forschung [SRS00].

4. Datenbankmodell

Das folgende Kapitel definiert das in dieser Arbeit verwendete Datenbankmodell. Dieses wird dann in Kapitel 5 in den Realzeitnachweis integriert. Der erste Abschnitt gibt einen Überblick über die grundlegenden Modellannahmen (z. B. Datenbank, Transaktion), die in den folgenden Abschnitten benötigt werden. Die beiden zentralen Punkte bilden dann das in dieser Arbeit neu entwickelte Concurrency-Control Protokoll PRED-DF (Abschnitt 4.2) und die Behandlung der aktiven Datenbankfunktionalität (Abschnitt 4.3). Abschnitt 4.4 erläutert eine mögliche Implementierung des Datenbankmodells.

4.1. Allgemeines

Unter einer Datenbank verstehen wir die zentralisierte, organisierte Zusammenfassung unterschiedlicher Datenobjekte. Ein Datenobjekt kann dabei selbst wieder eine Kombination mehrerer Objekte sein, z. B. eine Struktur, ein Array etc. Eine weitere Aussage über die Organisation der Daten wird nicht getroffen.

Definition 4 (Datenbank) *Eine Datenbank $\mathcal{D}$ ist eine Menge von k unteilbaren, disjunkten Datenobjekten x_i, $\mathcal{D} = \{x_1, x_2, \ldots, x_k\}$, $k < \infty$.*

Auf $\mathcal{D}$ werden Lese- und Schreiboperationen ausgeführt, die zu Transaktionen (TAs) zusammengefaßt sind (Read-Write-Modell für Transaktionen):

Definition 5 (Transaktion) *Eine Transaktion τ ist eine endliche Sequenz von n Aktionen des Typs $r(x)$ und $w(x)$, $\tau = p_1\, p_2 \ldots p_n$, $p_i \in \{r(x_i), w(x_i)\}$, $i = 1 \ldots n$, $n < \infty$, $x_i \in \mathcal{D}$. $r(x_i)$ bedeutet, den Wert des Datums x_i aus der Datenbank zu lesen, $w(x_i)$ steht für einen schreibenden Zugriff auf $x_i \in \mathcal{D}$.*

Das Transaktionsmodell ist flach. Das heißt, es gibt keine eingebetteten oder geschachtelten Transaktionen (nested transactions [Mos81]). Diese hätten zwar den Vorteil der erhöhten Parallelisierbarkeit in der Transaktionsbearbeitung (Inter- und Intra-Transaktionsparallelität), sie verursachen aber einen wesentlich höheren Verwaltungsaufwand und sind problematisch bezüglich der Isolation von Transaktionen. Zudem müssen dann die einzelnen Subtransaktionen unabhängig von den Applikationstasks gescheduled werden, um die erhöhte Parallelisierbarkeit voll nutzen zu können. Dies ist für das betrachtete Anwendungsgebiet nicht sinnvoll, da hierfür ein Datenbankmanager als eigenständiger, unabhängiger Systembaustein notwendig wäre, der aber aus Effizienzgründen nicht wünschenswert ist (z. B. wegen häufiger Taskwechsel, siehe Abschnitt 4.4, Implementierung).

Die Ressourcenanforderungen einer Transaktion, d. h. die aus der Datenbank gelesenen und die in die Datenbank geschriebenen Objekte, sind bereits vor dem Start einer Applikation bekannt und können zur Laufzeit nicht geändert werden (canned transactions, siehe Abschnitt 3.2.1).

Alle Informationen zu einer Transaktion werden in einem Transaktionstupel zusammengefaßt:

Definition 6 (Transaktionstupel) *Jede Transaktion τ wird als ein Tupel $\tau = (R, W, g)$ geschrieben. Dabei gelten folgende Vereinbarungen:*

- R *ist die Menge der von der Transaktion τ gelesenen Datenobjekte, $R \subseteq \mathcal{D}$,*
- W *ist die Menge der von der Transaktion τ geschriebenen Datenobjekte, $W \subseteq \mathcal{D}$,*
- g *ist die Menge der Abbildungen g_j, die die von τ vorgenommene Zuordnung von R auf W beschreiben, $g = \{g_1, g_2, \ldots, g_m\}$, $m = |W|$. Es gilt:*

$$y_j = g_j(x_1, x_2, \ldots, x_p),\ \forall y_j \in W,\ j = 1 \ldots m,\ x_k \in R,\ p \leq |R|. \tag{4.1}$$

Hierbei ist zu beachten, daß ein Objekt in einer Transaktion τ bereits einmal gelesen sein muß, bevor es in eine Funktion g_j eingehen kann.

Die Gesamtheit aller l Transaktionen τ_i, die im Verlauf einer Realzeitanwendung ausgeführt werden, bezeichnen wir mit $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_l\}$, $l < \infty$.

Die Datenbank ist hauptspeicherbasiert, insbesondere aus Gründen der Vorhersagbarkeit, da so Zugriffe auf Sekundärspeicher als Quelle der Unvorhersagbarkeit ausgeschlossen werden können (siehe Abschnitt 2.2.1). Auf Grund der heute möglichen Speichergrößen und der niedrigen Kosten von Speicherelementen stellt dies in der betrachteten Anwendungsklasse keine Einschränkung hinsichtlich der Funktionalität und des Funktionsumfangs dar. Positiv ist zusätzlich der so erzielte Geschwindigkeitsgewinn.

4.2. Concurrency-Control Protokoll PRED-DF

Wie bereits in Abschnitt 3.2.2 begründet, existiert im Augenblick kein für die Anwendungsdomäne optimiertes Concurrency-Control Protokoll. Im Rahmen dieser Arbeit wird deshalb mit PRED-DF ein neues optimiertes, semantisches Protokoll entwickelt.

PRED-DF (predeclaration and data-flow analysis based concurrency-control protocol) beruht auf der vorherigen Deklaration der Transaktionsressourcen, ist Lock-basiert und generiert serialisierbare Schedules [Mün00a]. Dabei ist es Deadlock-frei. Als Basis seiner Scheduling-Entscheidungen nutzt es, neben dem augenblicklichen Systemzustand, zusätzliche Informationen, die auf Grund der vordefinierten Transaktionen über den Datenfluß in der Applikation gewonnen werden können (insbesondere zu Zyklen im Datenfluß), um so unterschiedlich restriktive Locking-Strategien zwischen verschiedenen Transaktionsgruppen verwenden zu können. Die Blockierzeiten von Datensätzen lassen sich dadurch minimieren, d. h. Sperren werden so kurz wie möglich gesetzt. Damit kann die Transaktionsparallelität auf der Datenbank maximiert und die Verweildauer von Transaktionen in der Datenbank minimiert werden. Die Analyse des Datenflusses wird bereits während der Softwareentwicklung durchgeführt und benötigt so zum Zeitpunkt der Programmausführung keine zusätzliche Rechenzeit. Das Laufzeitverhalten von PRED-DF ist vorhersagbar und die Integration in einen Realzeitnachweis über die gesamte Applikation ist somit möglich.

4.2.1. Vorüberlegungen

Konfliktdetektion In traditionellen Datenbanken wird versucht, durch niedrige Lock-Granularität[1] eine große Zahl an gleichzeitig rechenbereiten Transaktionen zu erreichen. Dadurch

[1] Niedrige Lock-Granularität bezeichnet das Sperren von Daten auf der Grundlage möglichst *kleiner* Einheiten.

können die Wartezeiten beim Zugriff einer Transaktion auf den im Verhältnis langsamen Sekundärspeicher optimal genutzt werden, um so einen möglichst hohen Durchsatz und eine möglichst hohe Performance zu erzielen.

In hauptspeicherbasierten Datenbanken entfällt bei der Transaktionsbearbeitung der Zugriff auf den Sekundärspeicher[2] und man stellt fest, daß für diesen Datenbanktyp hohe Lock-Granularitäten am besten geeignet sind [LC87][GMS92]. Damit entfällt auch die Notwendigkeit, Ressourcen dynamisch während der Transaktionslaufzeit zu akquirieren. Am besten für die Nutzung positiver Cache-Effekte und zur optimalen Ausnutzung der CPU-Zeit wäre, die gesamte Datenbank zu sperren und somit alle Transaktionen hart zu serialisieren (wenige Kontextwechsel, kein Abort von Transaktionen). Dies führt jedoch zu nicht tolerierbar langen Blockierzeiten, falls die im Konflikt befindlichen Transaktionen von unterschiedlicher Länge und von unterschiedlicher Priorität sind (Long Duration Transactions).

Ein möglicher Mittelweg ist es, den gesamten Lese- und Schreib-Datensatz einer Transaktion während ihrer Bearbeitung zu sperren [UB98] (PRED). Voraussetzung dafür ist, daß die Lese- und Schreibdatensätze bereits beim Transaktionsstart bekannt sind. Ein großer Vorteil dieses Verfahrens ist seine effiziente Implementierbarkeit und seine Vorhersagbarkeit. Auch bezüglich der Performance ist PRED anderen Protokollen überlegen (z. B. Priority-Abort, Priority-Inheritance und Optimistic-Wait-50). Das Problem der Long Duration Transactions besteht aber auch hier weiterhin.

Graphbasierte Verfahren sind zur Konfliktdetektion ungeeignet, da sie einen hohen Verwaltungsaufwand haben und ihre Vorteile vor allem bei niedriger Lock-Granularität besitzen.

Konfliktauflösung Die beiden Strategien, die zur Konfliktauflösung in hauptspeicherbasierten Realzeitdatenbanken in Frage kommen, sind „blocking“ und „abort“. In Tabelle 4.1 werden diese beiden Möglichkeiten einander gegenüber gestellt. Multiversioning und Änderungen in der Serialisierungsreihenfolge entfallen wegen zu hoher Kosten (Verwaltungsaufwand) und mangelnder Vorhersagbarkeit.

Im Konfliktfall niederpriore Transaktionen abzubrechen (abort) hat den Vorteil, daß stets die Transaktion mit höherer Priorität zuerst bearbeitet wird und daß keine Prioritätsinversion und keine Deadlocks auftreten können. Dem stehen mehrere Nachteile gegenüber: Bestimmte Transaktionen können in Realzeitsystemen nicht abgebrochen werden (siehe Abschnitt 3.2.1) und die Worst-Case Execution Time (WCET) einer Transaktion ist unter Umständen außerordentlich hoch [MF00]. Zusätzlich treten Effizienzeinbußen durch den Rechenzeitverlust beim Transaktionsabbruch auf. Das Blocking-Verfahren weist diese Nachteile nicht auf. Die größten Probleme sind hier der Umgang mit langen Transaktionen bei unterschiedlicher Priorisierung und die Gefahr von Prioritätsinversion und Deadlocks. Bei hoher Lock-Granularität entstehen so häufig Situationen, bei denen Deadlines wegen der langen Blockierung von Transaktionen nicht eingehalten werden können, obwohl ein möglicher, serialisierbarer Schedule existieren würde.

Auf Grund obiger Überlegungen kombiniert PRED-DF Locking zur Konfliktdetektion mit der Blockierung von Transaktionen zur Konfliktauflösung. Die Gefahr von Deadlocks wird durch die günstige Wahl der Sperren ausgeschlossen. Informationen, die aus der Datenflußanalyse gewonnen werden, werden dazu verwendet, zwei verschiedene Lock-Granularitäten zwischen Transaktionsgruppen einzusetzen und so Blockierzeiten und Kosten zu minimalisieren: Die Lock-Granularität ist entweder der gesamte Datensatz einer Transaktion oder es werden jeweils der Lese- und Schreib-Datensatz getrennt gesperrt. Auf Grund der Vorhersagbarkeit von

[2] Zugriffe auf Sekundärspeicher können allerdings z. B. für das Schreiben von Log- und Recovery-Files notwendig sein.

blocking	abort
Vorteile	
• effiziente Implementierung • optimierte Rechenzeitnutzung (Cache-Effekte)	• Transaktionen hoher Priorität werden zuerst bearbeitet • keine Prioritätsinversion • keine Deadlocks
Nachteile	
• Prioritätsinversion • Deadlocks • Probleme bei langen Transaktionen	• Rechenzeitverlust durch den Abbruch von Transaktionen • möglicherweise nicht begrenzte WCET einer Transaktion • Detektion des Transaktionsabbruchs • Transaktionen können u. U. nicht abgebrochen werden

Tabelle 4.1.: Vergleich der beiden Strategien „blocking“ und „abort“ zur Konfliktauflösung in Concurrency-Control Protokollen.

PRED-DF können Deadline-Verletzungen durch Blockierung während der Softwareentwicklung in der Verifikations-Phase aufgedeckt und behoben werden.

4.2.2. Transaktions-Konflikt-Graph

Der Ausgangspunkt zur Analyse des von den Transaktionen in einer Applikation getragenen Datenflusses ist der Transaktions-Konflikt-Graph TCG($\mathcal{T}$) [BSR80]. Dieser kann auf der Grundlage der Definitionen aus Abschnitt 4.1 wie folgt beschrieben werden:

Definition 7 (Transaktions-Konflikt-Graph) *Für eine Menge von Transaktionen $\mathcal{T}$ ist der Transaktions-Konflikt-Graph $TCG(\mathcal{T})$ wie folgt definiert:*

$$TCG(\mathcal{T}) := (V,\ E).$$

Dabei sind V und E folgendermaßen festgelegt:

- *V ist die Menge der Knoten. Für jedes Element $\tau_i \in \mathcal{T}$ gibt es in V je einen Knoten R_i und W_i, $V = \{R_1, W_1, \ldots, R_l, W_l\}$.*
- *E ist die Menge der Kanten. Für jede Transaktion $\tau_i \in \mathcal{T}$ enthält $TCG(\mathcal{T})$ eine Kante $e_i = \langle R_i, W_i \rangle$. Zusätzlich beinhaltet der Graph folgende Kanten ($i, j = 1 \ldots l$):*
  - *$\langle W_i, W_j \rangle$, falls $W_i \cap W_j \neq \emptyset$; $\tau_i,\ \tau_j \in \mathcal{T},\ i > j$*
  - *$\langle W_i, R_j \rangle$, falls $W_i \cap R_j \neq \emptyset$; $\tau_i,\ \tau_j \in \mathcal{T},\ i \neq j$*

Der Transaktions-Konflikt-Graph ist nicht gerichtet.

Aus einem abstrakten Blickwinkel gesehen, charakterisiert dieser Graph den Datenfluß, der durch die Transaktionen in der Applikation transportiert wird. Problematisch für die Konsistenz der Daten und damit für das Concurrency-Control Protokoll sind Zyklen im Datenfluß, d. h. im Transaktions-Konflikt-Graphen. Schedules, die die parallele, unkontrollierte Ausführung von Transaktionen in einem Zyklus beinhalten, müssen durch das CC-Protokoll ausgeschlossen werden. Gleichzeitig können aber für Transaktionsgruppen, die nur einen linearen, nicht zyklischen Datenfluß tragen, weniger starke Einschränkungen bezüglich der parallelen Ausführung von Transaktionen gemacht werden.

Aus diesem Grund wird jeder Transaktion τ eine Menge an Transaktionen zugeordnet, mit denen sie sich potentiell im Konflikt befindet (Kante im TCG), für die aber auf Grund der Ergebnisse der Datenflußanalyse weniger strenge Ausschlußkriterien im Locking-Verfahren gelten. Diese Menge heißt *Friend-Set* von τ und wird mit $\mathcal{T}_f(\tau)$ bezeichnet. Zur Bestimmung dieser Zuordnung wird zuerst die Untermenge der normalisierten Transaktionen definiert:

Definition 8 (normalisierte Transaktion) *Eine Transaktion τ wird als normalisiert bezeichnet, falls Lese- und Schreiboperationen so getrennt werden können, daß τ wie folgt geschrieben werden kann:*

$$\tau = r_1\, r_2 \ldots r_n\, w_1\, w_2 \ldots w_m.$$

Das heißt, Daten werden nur zu Beginn der Transaktion τ eingelesen, Schreiboperationen werde nur am Ende durchgeführt. Dazwischen liegen sämtliche in τ durchgeführte Berechnungen. Zusätzlich zu den gewöhnlichen Zuständen einer Transaktion lassen sich somit drei zusätzliche Bearbeitungsphasen festlegen[3], die in Abbildung 4.1 dargestellt sind:

- *read.* Alle Daten, die von einer Transaktion benötigt werden, werden eingelesen; diese Phase beginnt mit dem Start der Transaktion und endet mit dem letzten „read"-Kommando.
- *calculate.* Zwischen der „read"- und der „write"-Phase. Hier werden alle Berechnungen der Transaktion durchgeführt.
- *write.* Beginnt mit dem ersten „write"-Kommando und endet mit dem „commit" der Transaktion.

Der Transaktionszustand „scheduled" umfaßt den gesamten Zeitraum, während dessen sich die Transaktion im System befindet, d. h. vom Start bis zum Abbruch oder zum erfolgreichen Abschluß. Andernfalls befindet sich die Transaktion im Zustand „undefined".

Ausschlaggebend für Konfliktbetrachtungen sind nur Transaktionen, die im Transaktions-Konflikt-Graph durch Kanten verbunden sind. Eine Gruppe von Transaktionen, die alle durch einen Kantenzug verbunden sind, werden zu einer Konfliktmenge zusammengefaßt:

Definition 9 (Konfliktmenge) *Eine Teilmenge $\mathcal{T}_k \subseteq \mathcal{T}$ heißt Konfliktmenge, falls für alle $\tau_i, \tau_j \in \mathcal{T}_k$, $i \neq j$, τ_j von τ_i in $TCG(\mathcal{T}_k)$ erreichbar ist.*

Das heißt, eine Konfliktmenge ist eine zusammenhängende Teilmenge des Transaktions-Konflikt-Graphen TCG($\mathcal{T}$). Entsprechend ihren Eigenschaften kann eine Konfliktmenge näher charakterisiert werden:

[3] Diese Phasen sind nicht zu verwechseln mit anderen Phasendefinitionen, z. B. im Rahmen von validierungsbasierten Protokollen.

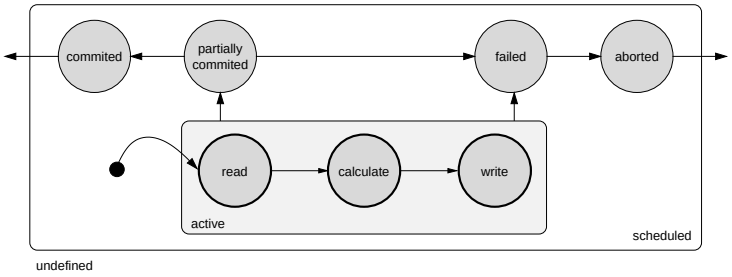


Abbildung 4.1.: Zustände und Bearbeitungsphasen einer normalisierten Transaktion. Im Zustand „active“ existieren zusätzlich die Bearbeitungsphasen „read“, „calculate“ und „write“.

Definition 10 (azyklische Konfliktmenge) *Eine Konfliktmenge $\mathcal{T}_k$ heißt azyklisch, falls der Transaktions-Konflikt-Graph TCG($\mathcal{T}_k$) azyklisch ist und keine Transaktion $\tau_i \in \mathcal{T}_k$ Element eines Zyklus in TCG($\mathcal{T}$) ist. Sie wird mit $\mathcal{T}_{ka}$ bezeichnet.*

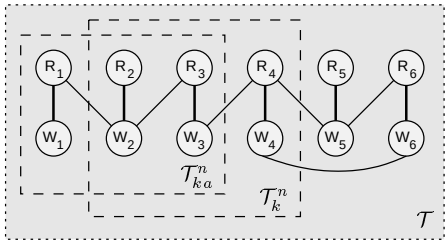
Definition 11 (normalisierte Konfliktmenge) *Eine Konfliktmenge $\mathcal{T}_k$, deren Elemente alle normalisiert sind, heißt normalisierte Konfliktmenge und wird mit $\mathcal{T}_k^n$ bezeichnet.*

Mit Hilfe der bisher vorgenommenen Definitionen kann im folgenden das Friend-Set einer normalisierten Transaktion τ definiert werden:

Definition 12 (Friend-Set) *Das Friend-Set einer Transaktion $\tau \in \mathcal{T}_{ak}^n$ sind alle normalisierten Nachbartransaktionen von τ in TCG($\mathcal{T}$). Die Menge dieser Transaktionen wird mit $\mathcal{T}_f(\tau)$ bezeichnet. Hierbei ist $\mathcal{T}_{ak}^n$ eine beliebige normalisierte, azyklische Konfliktmenge in $\mathcal{T}$. Eine Transaktion ist nicht Element ihres eigenen Friend-Sets, $\tau \notin \mathcal{T}_f(\tau)$. Es gilt: $\tau \in \mathcal{T}_f(\tau_i)$ für alle $\tau_i \in \mathcal{T}_f(\tau)$ (Symmetrie).*

Die vorgenommenen Definitionen erläutert Abbildung 4.2 anhand eines Beispiels. In a) ist ein Transaktions-Konflikt-Graph abgebildet. Es wird angenommen, daß alle Transaktionen $\tau_1 \ldots \tau_6$ normalisiert sind. Hierbei stellen $\mathcal{T}_{ka}^n$ und $\mathcal{T}_k^n$ jeweils gültige Konfliktmengen dar. $\mathcal{T}_{ka}^n$ ist zusätzlich azyklisch, $\mathcal{T}_k^n$ dagegen nicht, da τ_4 Element eines Zyklus in TCG($\mathcal{T}$) ist. In b) ist die aus $\mathcal{T}_{ka}^n$ für τ_1, τ_2 und τ_3 folgende Bildung von Friend-Sets abgebildet.

a)



b)

$$\mathcal{T}_f(\tau_1) = \{\tau_2\}$$
$$\mathcal{T}_f(\tau_2) = \{\tau_1, \tau_3\}$$
$$\mathcal{T}_f(\tau_3) = \{\tau_2, \tau_4\}$$
$$\mathcal{T}_f(\tau_4) = \{\tau_3\}$$

Abbildung 4.2.: Bildung von Friend-Sets. Abbildung a) zeigt einen Transaktions-Konflikt-Graphen und zwei mögliche Konfliktmengen. Abbildung b) zeigt die bezüglich $\mathcal{T}_{ka}^n$ folgende Aufteilung in Friend-Sets. Alle Transaktionen werden als normalisiert angenommen.

4.2.3. PRED-DF Protokolldefinition

Das Concurrency-Control Protokoll PRED-DF selbst unterteilt sich in zwei aufeinanderfolgende Schritte: Im ersten Schritt werden alle Informationen, die bereits offline gewonnen werden können, extrahiert und aufbereitet. Auf Grundlage dieser Information werden dann im zweiten Schritt zur Laufzeit die Scheduling-Entscheidungen getroffen.

- **Schritt 1 (offline):**
  - Bestimmung von R_i und W_i für alle $\tau_i \in \mathcal{T}$,
  - Bestimmung von $\mathcal{T}_f(\tau_i)$ für alle $\tau_i \in \mathcal{T}, i = 1 \ldots l$.
- **Schritt 2 (online):** Die Transaktionen werden zur Laufzeit entsprechend einem Locking-Protokoll gescheduled. Dieses baut auf der Kompatibilitätsmatrix K_{ij} auf, die in Abbildung 4.3 dargestellt ist. τ_i ist hierbei eine Transaktion, die den Lock auf einen Datensatz neu anfragt, τ_j besitzt den Lock bereits. Alle Sperren umfassen jeweils das gesamte Lese- und Schreibset einer Transaktion.

  Zusätzlich gelten folgende Regeln:

  1. $\tau_i \in \mathcal{T}_f(\tau_j)$. Gegenüber allen Transaktionen im Friend-Set einer Transaktion werden die Read-Locks bereits nach der Beendigung der Read-Phase aufgehoben, Write-Locks werden erst am Beginn der Write-Phase gesetzt.
  2. $\tau_i \notin \mathcal{T}_f(\tau_j)$. Lese- *und* Schreibsperren werden gleichzeitig am Anfang der Transaktion gesetzt und erst nach Beendigung der Transaktion wieder aufgehoben.

τ_i \ τ_j	r	w
r	√	×
w	×	×

√ Lock gewährt
× Lock verweigert

Abbildung 4.3.: Kompatibilitätsmatrix K_{ij} des Concurrency-Control Protokolls PRED-DF (shared read-lock, exclusive write-lock).

Eine Transaktion muß alle für sie notwendige Lese- und Schreibsperren besitzen, bevor sie mit der Bearbeitung fortfahren kann. Locks, die einmal zugeteilt wurden, werden später nicht mehr entzogen. Transaktionen werden durch die Datenbank nicht abgebrochen. Die Auflösung von Konflikten findet durch die Verzögerung der blockierten Transaktionen statt. Die parallele Ausführung von mehreren Instanzen derselben Transaktion ist ausgeschlossen, da eine Transaktion nicht Element ihres eigenen Friend-Sets sein kann.

Die Unterschiede in der Anforderung von Sperren bei PRED-DF bezüglich Transaktionen, die innerhalb und außerhalb des Friend-Sets einer Transaktion liegen, zeigt schematisch Abbildung 4.4. Zusätzlich ist der Verlauf der Sperrenanforderung bei PRED eingezeichnet. Falls eine Transaktion τ_y nicht im Friend-Set einer Transaktion τ_x liegt, ist für PRED und PRED-DF der zeitliche Verlauf der Sperrenanforderung identisch.

4.2.4. Bildung von Friend-Sets

Der Hauptvorteil des Protokolls PRED-DF liegt darin, die Lock-Zeiten für die Lese- und Schreibmenge einer Transaktion τ für alle Transaktionen $\tau_i \in \mathcal{T}_f(\tau)$ zu minimieren. Ziel muß es daher

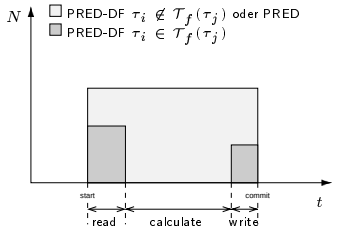


Abbildung 4.4.: Exemplarischer zeitlicher Verlauf der Zahl der von einer Transaktion τ_x bezüglich einer anderen Transaktion τ_y angeforderten Locks N. Falls $\tau_y \notin \mathcal{T}_f(\tau_x)$, ist der Verlauf der Sperrenanforderung von PRED und PRED-DF identisch.

sein, möglichst umfangreiche Friend-Sets zu definieren. Im folgenden wird daher ein effizientes Vorgehen zur Segmentierung des Transaktions-Konflikt-Graphen in Konfliktmengen und zur Bildung von Friend-Sets angegeben:

1. *Normalisierung der Transaktionen.* Transformation möglichst vieler Transaktionen einer Applikation in normalisierte Transaktionen. Die Lese- und Schreibdatensätze einer Transaktion sind bekannt und eine Konvertierung ist daher bis auf wenige Ausnahmen, z. B. falls sich eine Transaktion über mehrere Komponenten erstreckt, möglich.

2. *Segmentierung in Konfliktmengen.* Der Transaktions-Konflikt-Graph TCG($\mathcal{T}$) wird in seine Zusammenhangkomponenten zerlegt. Diese Zerlegung ist immer möglich. Man erhält w disjunkte Konfliktmengen $\mathcal{T}_{k,1}, \mathcal{T}_{k,2} \ldots \mathcal{T}_{k,w}$. Der Aufwand hierfür ist bei Verwendung von Adjazenzenlisten zur Speicherung der Graphinformationen zu TCG($\mathcal{T}$) von der Ordnung $\mathcal{O}(m+n)$. m ist die Zahl der Ecken, n die Zahl der Kanten des Graphen. In Anhang C.1 ist ein entsprechender Algorithmus angegeben.

   Würde nun zur Bestimmung der Friend-Sets lediglich jede Zusammenhangkomponente auf Zyklusfreiheit und Normalisiertheit hin überprüft werden, wäre die Zahl an normalisierten, azyklischen Konfliktmengen $\mathcal{T}^n_{ka,i},\ i = 1 \ldots t$ zur Bestimmung von Friend-Sets sehr gering. Aus diesem Grund werden die bisher bestimmten Zusammenhangkomponenten weiter aufgetrennt.

3. *Abtrennung azyklischer, normalisierter Teil-Konfliktmengen.* Jede gefundene Konfliktmenge wird auf Zyklen untersucht. Wird ein Zyklus gefunden, so werden alle an diesem Zyklus beteiligten Transaktionen in eigenen Konfliktmengen $\mathcal{T}_{kc,j}$ zusammengefaßt. In der verbleibenden Menge wird nun jeder Transaktion, die nicht normalisiert ist, eine eigene Konfliktmenge $\mathcal{T}^{\bar{n}}_{k,m}$ zugewiesen.

   In der verbleibenden Menge

   $$\mathcal{T}'_{k,i} = \mathcal{T}_{k,i} \setminus \left(\cup_j \mathcal{T}_{kc,j} \bigcup \cup_m \mathcal{T}^{\bar{n}}_{k,m} \right) \tag{4.2}$$

   werden nun jeweils erneut die Zusammenhangkomponenten $\mathcal{T}^n_{ka,i},\ i = t+1 \ldots u$ bestimmt. Diese sind nun alle azyklisch und normalisiert.

4. *Bestimmung von Friend-Sets.* Für alle Transaktionen in den azyklischen, normalisierten Konfliktmengen werden die Friend Sets entsprechend Definition 12 bestimmt.

Abbildung 4.5 veranschaulicht das hier dargestellte Vorgehen anhand eines einfachen Beispiels.

Es ist nicht möglich, wie man intuitiv vermuten könnte, bei normalisierten, zyklischen Konfliktmengen dadurch unter PRED-DF Serialisierbarkeit herzustellen, daß eine Transaktion des

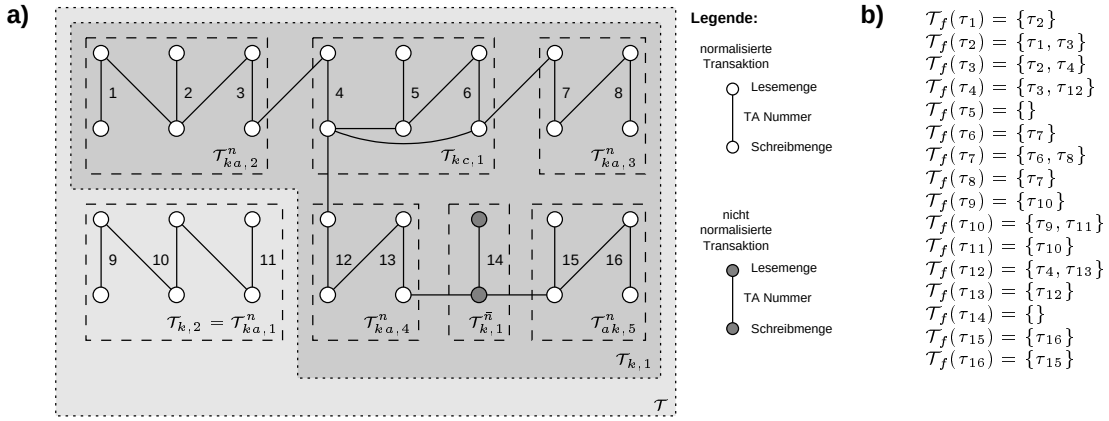


Abbildung 4.5.: Beispiel für die Aufteilung von $\mathcal{T}$ in Friend-Sets. a) zeigt die Segmentierung in die azyklischen, normalisierten Konfliktmengen $\mathcal{T}^n_{ka,i}$, $i = 1 \ldots 5$, nach dem vorgestellten Verfahren, b) gibt die sich daraus ergebenden Friend-Sets.

Zyklus aus allen Friend-Sets der Konfliktmenge entfernt und damit hart gegenüber allen anderen Transaktionen in dieser Konfliktmenge serialisiert wird. Abbildung 4.6 gibt dazu ein einfaches Gegenbeispiel: Teilbild a) zeigt einen möglichen Transaktions-Konflikt-Graphen und eine entsprechende Segmentierung in Friend-Sets. τ_4 ist hier aus dem Zyklus gelöst worden. Der folgende Schedule s für diese Konstellation wäre mit den Locking-Regeln von PRED-DF vereinbar[4]:

$$s = r_2(x)\, w_1(w)\, w_1(x)\, r_4(w)\, w_4(y)\, r_3(y)\, w_3(z)\, w_2(z)$$

Abbildung 4.6 b) zeigt den zu s gehörigen Konflikt-Graphen $G(s)$. Der Konflikt-Graph ist zyklisch, d. h. der Schedule ist nicht konflikt-serialisierbar (siehe Anhang B.1). Allerdings ist $\mathcal{T}_{k,1}$ nach Definition 10 nicht azyklisch und damit im Widerspruch zur Protokolldefinition von PRED-DF.

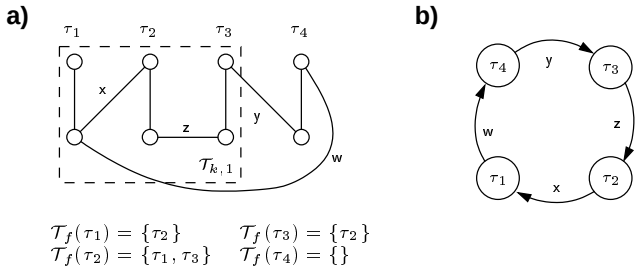


Abbildung 4.6.: Gegenbeispiel zur Auftrennung von Zyklen in normalisierten, zyklischen Konfliktmengen. a) zeigt einen Transaktions-Konflikt-Graphen und eine Aufteilung in Friend-Sets. b) zeigt den zyklischen Konflikt-Graphen $G(s)$ für den Schedule s.

4.2.5. Nachweis von Serialisierbarkeit und Deadlock-Freiheit

Serialisierbarkeit

Behauptung: Schedules unter PRED-DF sind konflikt-serialisierbar.

[4] $r_2(x)$ beschreibt in dieser Notation den lesenden Zugriff der Transaktion τ_2 auf das Datum x.

Beweis:

1. Es gilt: Schedule s konflikt-serialisierbar $\Leftrightarrow$ $G(s)$ azyklisch (siehe Anhang B.1).

   Damit ist zu beweisen: Der Konflikt-Graph $G(s)$ ist für alle möglichen Schedules s unter PRED-DF azyklisch.

2. Der Beweis erfolgt vorerst für Schedules in *einer* beliebigen Konfliktmenge $\mathcal{T}^n_{ka,i}$ bzw. $\mathcal{T}_{kc,i}$ entsprechend der Aufteilung von $\mathcal{T}$ in Abschnitt 4.2.4. Betrachtet wird ein beliebiger Schedule s. Fallunterscheidung bezüglich der Zyklizität der Konfliktmenge:

   a) *Konfliktmenge zyklisch.* In diesem Fall geht PRED-DF in 2PL über. Die Lock-Granularität ist der gesamte Transaktionsdatensatz. 2PL ist konflikt-serialisierbar [VGH93].

   b) *Konfliktmenge azyklisch.* Ist die Konfliktmenge nicht normalisiert, so ist PRED-DF äquivalent zu 2PL, d. h. konflikt-serialisierbar.

      Ist die Konfliktmenge normalisiert, so muß eine Fallunterscheidung bezüglich der potentiellen Zykluslänge l in $G(s)$ getroffen werden, da der Transaktions-Konflikt-Graph nicht gerichtet ist:

      - $l = 1$. Ausgeschlossen, da parallele Ausführung derselben Transaktion nicht möglich ist.
      - $l = 2$. An einem Zyklus der Länge $l = 2$ in $G(s)$ können nur genau zwei Transaktionen τ_1 und τ_2 beteiligt sein. Diese sind auf Grund der Annahme, daß TCG azyklisch ist, durch höchstens eine Kante e im Transaktions-Konflikt-Graph verbunden. Daraus folgt, daß $G(s)$ nicht zyklisch ist.

        Der Beweis erfolgt exemplarisch anhand der Kante $e = \langle \mathrm{R}_1, \mathrm{W}_2 \rangle$ in Abbildung 4.7. Für einen Zyklus in $G(s)$ muß folgendes erfüllt sein:

        $$\begin{aligned} r_1(x) &< w_2(x) \quad \wedge \\ w_2(y) &< r_1(y), \quad x, y \in R_1 \cap W_2 \end{aligned} \tag{4.3}$$

        „$p_i < p_j$“ bedeutet in diesem Zusammenhang $s = p_1 \ldots p_i \ldots p_j \ldots p_n$, d. h. „$p_i$ liegt im Schedule vor p_j“.

        Falls der Schedule s mit $r_1(x)$ startet, so folgt mit (4.3) und aus den Regeln von PRED-DF: $r_1(y) < \mathrm{unlock}_1(x, y)$, dies ist ein Widerspruch zu (4.3): $w_2(y) < r_1(y) < \mathrm{unlock}_1(y)$. Analoges gilt, falls der Schedule mit $w_2(y)$ startet. Daher ist $G(s)$ azyklisch. Der Beweis für andere mögliche Kanten erfolgt äquivalent.

      - $l > 2$. Der Transaktions-Konflikt-Graph ist azyklisch, d. h. es gibt keinen Weg $t = (\tau_1, \tau_2, \ldots \tau_n) \in \mathrm{TCG}$, für den gilt: $\tau_1 \equiv \tau_n$. Auf Grund der Definition von TCG folgt hieraus: es gibt keine Operationen p_i, die untereinander in Konflikt stehen und für die gilt: $p_i(x_i) \in \tau_i$ und $p_{i+1}(x_i) \in \tau_{i+1}$, $x_i \in \mathcal{D}$, $i = 1 \ldots n-1$ mit $\tau_1 \equiv \tau_n$. Genau dies ist aber Voraussetzung für einen Zyklus in G, d. h. $G(s)$ azyklisch.

3. Bei Schedules, die sich über mehrere Konfliktmengen $\mathcal{T}^n_{ka,i}$, $\mathcal{T}_{kc,i}$ und $\mathcal{T}^{\bar{n}}_{k,i}$ erstrecken, gilt folgendes (ohne Einschränkung wird angenommen, daß alle Mengen in der selben Zusammenhangkomponente liegen):

- Ein Zyklus im Transaktions-Konflikt-Graph liegt immer vollständig in einer Konfliktmenge $\mathcal{T}_{kc,i}$. Für diesen ist $G(s)$ sicher nicht zyklisch (Äquivalenz zu 2PL). Zyklen können aus diesem Grund in $G(s)$ auch nicht über mehrere Konfliktmengen geschlossen werden (wie dies z. B. in Abbildung 4.6 geschieht), da der TCG außerhalb $\mathcal{T}_{kc,i}$ für diese Zusammenhangkomponente nicht zyklisch ist.
- An der Schnittstelle von zwei Konfliktmengen können keine Zyklen in $G(s)$ induziert werden. Der Beweis hierfür ist äquivalent zu 2.

Damit folgt: $G(s)$ unter PRED-DF ist stets azyklisch. □

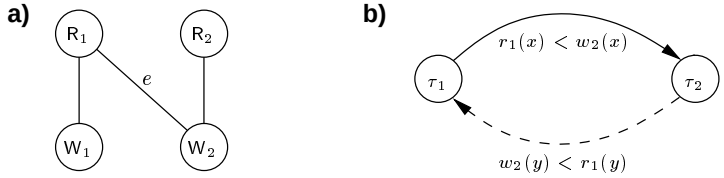


Abbildung 4.7.: a) Transaktions-Konflikt-Graph der Transaktionen τ_1 und τ_2. Konflikt zwischen R_1 und W_2. b) Mögliches Szenario für einen Zyklus im Konfliktgraphen $G(s)$ (siehe Text).

Deadlock-Freiheit

Behauptung: PRED-DF ist Deadlock-frei.

Beweis:

1. Zu beweisen ist, daß der Wait-Graph W für alle Schedules unter PRED-DF stets azyklisch ist (siehe hierzu Anhang B.2).

   Betrachtet wird ein Schedule auf einer beliebigen Konfliktmenge $\mathcal{T}^n_{ka,i}$ bzw. $\mathcal{T}_{kc,i}$ entsprechend der Aufteilung von $\mathcal{T}$ in Abschnitt 4.2.4. Fallunterscheidung bezüglich Zyklizität der Konfliktmenge:

   a) Konfliktmenge *zyklisch*. Im diesem Fall ist PRED-DF äquivalent zu 2PL. Es werden allerdings alle notwendigen Ressourcen auf einmal atomar angefordert und gemeinsam wieder freigegeben. Daraus folgt nach [CES71] Deadlock-Freiheit.

   b) Konfliktmenge *azyklisch*. Falls nicht alle Transaktionen normalisiert sind, ist PRED-DF äquivalent zu 2PL, d. h. Deadlock-frei.

      In allen anderen Fällen (normalisierte Konfliktmenge): Fallunterscheidung nach potentieller Zykluslänge l in W:

      - $l = 2$. Da TCG nicht zyklisch folgt: Es existiert höchstens eine Kante in TCG, die τ_1 und τ_2 verbindet (siehe Beweis zur Serialisierbarkeit). Auf Grund der Normalisierung von τ_1 und τ_2 ergibt sich, daß höchstens eine gemeinsam genutzte Ressource besteht, d. h. keine Deadlocks [CES71].
      - $l > 2$. Analog zum Beweis der Serialisierbarkeit ergibt sich, daß auf Grund der Azyklizität des Transaktions-Konflikt-Graphen W stets azyklisch ist.

2. Schedules über mehrere Konfliktmengen. Auch hier gilt, daß Zyklen über mehrere Mengen nicht geschlossen werden können und an der Grenzfläche kein Zyklus in W induziert werden kann.

Damit folgt, daß W stets azyklisch ist. □

4.2.6. Sensor-Transaktionen

Einen Sonderfall bilden Transaktionen, die im System dazu dienen, die Ergebnisse der Sensorauslese in der Datenbank abzulegen (Sensor-Transaktionen). Sensor-Transaktionen besitzen einen leeren Lese-Datensatz und im System existieren nur Transaktionen, die sich lediglich über ihre Lese-Datensätze mit den Sensor-Transaktionen im Konflikt befinden. Durch diese Zugriffe der Applikation ergibt sich in der Regel eine Konfiguration, wie sie in Abbildung 4.8 a) dargestellt ist.

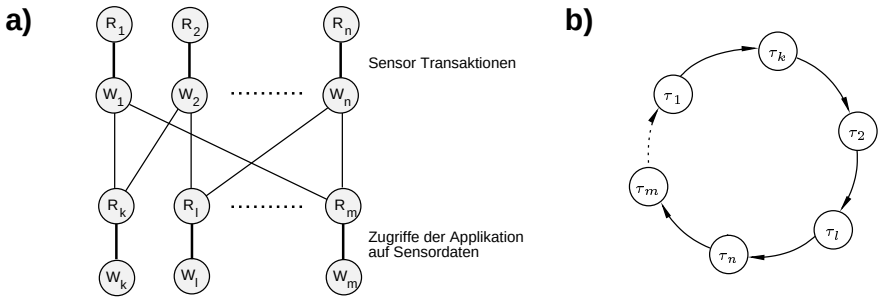


Abbildung 4.8.: a) Sensor-Transaktionen und Zugriffe der Applikation auf die Sensordaten. Die Transaktionen $\tau_1 \ldots \tau_n$ legen Sensorwerte in der Datenbank ab. Die Transaktionen $\tau_k \ldots \tau_m$ lesen die Datenwerte und verarbeiten sie weiter. b) zeigt den Verlauf eines potentiellen Zyklus im Graphen $G(s)$.

Verallgemeinern läßt sich diese Situation folgendermaßen:

Behauptung: Gibt es im Transaktions-Konflikt-Graphen TCG($\mathcal{T}$) einen Zyklus, an dem jede darin enthaltene Transaktion entweder *nur* mit ihrem Lese- oder *nur* mit ihrem Schreib-Datensatz beteiligt ist, so kann dieser Zyklus für die Bestimmung der Friend-Sets vernachlässigt werden, da er unter PRED-DF keine Zyklen in $G(s)$ induzieren kann.

Beweis: Da von jeder Transaktion an einem potentiellen Zyklus nur jeweils eine einzige Ressource beteilgt ist (entweder der Lese- oder der Schreib-Datensatz), definiert ein solcher Zyklus gleichzeitig eine Reihenfolge der Transaktionen. Nach Abbildung 4.8 b) gilt dann ohne Beschränkung der Allgemeinheit:

$$\tau_1 > \tau_k > \tau_2 > \ldots > \tau_m > \tau_1. \tag{4.4}$$

Das heißt, Operationen der Transaktion τ_1 müßten sowohl vor als auch nach der Transaktion τ_k ausgeführt werden. Dies ist unter PRED-DF ein Widerspruch zur Annahme, da der Lese- oder Schreib-Datensatz einer Transaktion immer als eine einzige Einheit gesperrt werden. □

4.3. Aktive Datenbankfunktionalität

Dieser Abschnitt beschreibt das Modell, auf dem der aktive Teil der in dieser Arbeit definierten Datenbank basiert. Abschnitt 4.3.1 befaßt sich dabei mit dem zugrunde liegenden formalen Wissensmodell der aktiven Regeln. Die Formalisierung erlaubt es, die Abhängigkeiten der Objekte in der Datenbank, die durch das Regelwerk entstehen, zu analysieren (Analysegraph). Aus

den Zeitbedingungen für die einzelnen Datenobjekte (temporale Konsistenz) und für die Regelausführung können so die Zeitbedingungen für die Ausgangsdaten, die Abbildung auf die Implementierung und die Deadlines der Tasks, die für die Ausführung des Regelwerkes verantwortlich sind (siehe Abschnitt 4.3.2, Ausführungsmodell), abgeleitet werden. Basierend auf dem Analysegraphen ist zudem eine Optimierung der Regelbearbeitung möglich (siehe Abschnitt 4.3.3). Die Formalisierung erlaubt eine eindeutige Darstellung der Regeln und stellt so außerdem sicher, daß die Entwicklung und Spezifikation des Regelwerkes einfach durch Werkzeuge unterstützt werden kann.

4.3.1. Wissensmodell

Das Wissensmodell faßt die Informationen über die aktiven Inhalte der Datenbank zusammen. Diese sind das Ergebnis der Anforderungsspezifikation und müssen bei der Systementwicklung vom Designer festgelegt werden. Das Wissensmodell gliedert sich in die *Regelbasis* $\mathcal{R}'$ und die *Konsistenzinformation* $\mathcal{K}$ und die für die Informationsbearbeitung festgelegten *Zeitanforderungen*. Die Regelbasis enthält die vom Benutzer frei definierbaren Regeln. Sie sind jeweils mit einem einzelnen Datenobjekt verbunden. Die Konsistenzinformation ist das Wissen über die dauerhaften Abhängigkeiten der Daten untereinander, die von der Datenbank sicherzustellen sind. Sowohl für die Abarbeitung der Konsistenzinformation als auch für die Regelausführung können vom Systemdesigner Zeitschranken festgelegt werden, die für die korrekte Funktion der Applikation eingehalten werden müssen (Zeitanforderungen).

Regelbasis

Die Gesamtheit aller frei definierbaren ECA-Regeln ρ_{ik} in der aktiven Datenbank wird zur Regelbasis $\mathcal{R}'$,

$$\mathcal{R}' := \{\rho_{11}, \rho_{12}, \ldots, \rho_{ik} \ldots\}, \tag{4.5}$$

zusammengefaßt. Mit einem Datum $x_i \in \mathcal{D}$ können hierbei mehrere Regeln verknüpft sein. Eine einzelne ECA-Regel ρ_{ik} setzt sich aus den folgenden einzelnen Komponenten zusammen:

$$\rho_{ik} := (E_{\mathrm{db},i}, c_{ik}, A_{ik}). \tag{4.6}$$

Es gelten die folgenden Vereinbarungen:

$E_{\mathrm{db},i}$ ist ein Element aus der Menge der *Ereignisse in* der Datenbank, das heißt:

$$E_{\mathrm{db},i} = w(x_i),\ x_i \in \mathcal{D}. \tag{4.7}$$

c_{ik} ist die *Bedingung*, d. h. eine Funktion $c_{ik} : S \times \mathcal{D}^n \to \mathbb{B}$ die den augenblicklichen Zustand des Systems bzw. der Datenbank abprüft. S bezeichnet hierbei den Zustand der einbettenden Applikation. c_{ik} ist folgendermaßen definiert:

$$c_{ik} : (S, x_1, x_2, \ldots, x_n) \mapsto r_{ik} = c_{ik}(S, x_1, x_2, \ldots, x_n),\ r_{ik} \in \mathbb{B}. \tag{4.8}$$

Dabei ist jeder Bedingung c_{ik} eine Transaktion τ_{ik}^{c} zugeordnet, die den Zugriff auf die Datenbank zur Bewertung des Datenbankzustandes kapselt. Beide sind in C_{ik} zusammengefaßt:

$$C_{ik} := (c_{ik}, \tau_{ik}^{c}). \tag{4.9}$$

Ist keine Bedingung angegeben, so ist per definitionem $c_{ik} := t$.

A_{ik} ist die *Aktion* der ECA-Regel. Sie wird ausgeführt, falls das Ereignis $E_{\mathrm{db},i}$ eintritt und die dazugehörige Bedingung c_{ik} wahr ist. A_{ik} ist entweder die Ausführung einer Transaktion auf der Datenbank oder ein internes Ereignis der einbettenden Applikation $E_{\mathrm{ap},ik}$, das aus der Datenbank heraus ausgelöst werden kann:

$$A_{ik} := (\, \tau_{ik} \mid E_{\mathrm{ap},ik} \,). \tag{4.10}$$

Die internen Ereignisse $E_{\mathrm{ap},ik}$ dienen dazu, die Bearbeitung eines Tasks in der Applikation anzustoßen (siehe Abschnitt 5.1.2).

Es ist möglich, Regeln zur Programmlaufzeit zur Regelbasis hinzuzufügen und diese wieder zu entfernen, z. B. um nur unter bestimmten Umständen über einen Systemzustand informiert zu werden. Dies ist auch in der für diese Arbeit durchgeführten prototypischen Implementierung der aktiven Datenbank möglich. Die in variablen Regeln auftretenden Datenbankzugriffe und Transaktionen und die internen Ereignisse der Applikation, die möglicherweise von der Datenbank ausgelöst werden, müssen für die Realzeitanalyse und für die Bestimmung von Worst-Case Execution Times bekannt sein. Im weiteren Verlauf der Arbeit kann ohne Beeinträchtigung der Allgemeinheit eine statische Regelbasis vorausgesetzt werden, die dem maximal ausgebauten Regelsatz des dynamischen Falls entspricht (siehe Abschnitt 5.2.2). Unter Umständen können noch zusätzlich verschiedene Szenarien berücksichtigt werden, falls sich unterschiedliche Regelkonfigurationen gegenseitig ausschließen.

Konsistenzinformation

Wie bereits in der Anforderungsanalyse in Abschnitt 3.1 dargelegt, ist es nicht möglich, alle Datenobjekte $x_i \in \mathcal{D}$ so zu definieren, daß keine logischen Abhängigkeiten der Daten untereinander bestehen. Diese können aber über die aktive Funktionalität der Datenbank automatisch angeglichen werden. Die Abhängigkeiten müssen dem Entwickler beim Entwurf der Datenbasis für eine Applikation bekannt sein. Mit dem Wissen über diese Abhängigkeiten kann für jedes Datenobjekt $x_i \in \mathcal{D}$ eine Abbildung h_i definiert werden, die dessen Abhängigkeit von den anderen Objekten der Datenbank beschreibt. Die Konsistenzinformation ist über die Laufzeit der Applikation nicht variabel. Es gilt:

$$x_i := h_i(\mathcal{M}_i),\ \mathcal{M}_i \subseteq \mathcal{D}. \tag{4.11}$$

Die Datenbankoperationen zur Konsistenzsicherung müssen, wie alle anderen Zugriff auf die Datenbank auch, innerhalb einer Transaktion ausgeführt werden. Jede Funktion h_i zur Aktualisierung des Objektes x_i wird daher in einer Transaktion τ_i^k gekapselt:

$$\tau_i^k := (\mathcal{M}_i, x_i, h_i),\ \forall x_i \in \mathcal{D}. \tag{4.12}$$

Die Gesamtheit aller Transaktionen τ_i^k, die der Konsistenzsicherung dienen, wird zur *Konsistenzinformation* $\mathcal{K} := \{\tau_1^k, \tau_2^k, \ldots \tau_n^k\}$ zusammengefaßt.

Abbildung der Konsistenzinformation auf Regeln Um eine einheitliche Behandlung des Wissens über die Konsistenzbeziehungen der Daten und die Regeln in der Regelbasis zu ermöglichen, wird die Konsistenzinformation $\mathcal{K}$ auf ECA-Regeln abgebildet.

Ändert sich ein Datum x_i in der Datenbank, so sind alle Daten anzugleichen, die von x_i abhängig sind. Das heißt, es sind alle Transaktionen τ_k^k auszuführen, für die gilt:

$$\tau_k^k \in \{\tau_l^k \mid x_i \in \mathcal{M}_l\} =: \mathcal{K}_{x_i}. \tag{4.13}$$

Die Konsistenzinformation $\mathcal{K}$ einer Applikation kann also mit den frei definierbaren Regeln $\mathcal{R}'$ folgendermaßen zu einer gemeinsamen Regelbasis $\mathcal{R}$ zusammengefaßt werden:

$$\mathcal{R} := \mathcal{R}' \cup \mathcal{R}_k, \tag{4.14}$$

$$\mathcal{R}_k := \{\rho_{ik} \mid \rho_{ik} := (w(x_i), t, \tau_l^k), x_i \in \mathcal{D},\ \tau_l^k \in \mathcal{K}_{x_i}\}. \tag{4.15}$$

Analysegraph Die Informationen der gemeinsamen Regelbasis $\mathcal{R}$ können auf einen Graphen abgebildet werden, der alle funktionalen Abhängigkeiten der Datenobjekte untereinander visualisiert:

Definition 13 (Analysegraph) *Der Analysegraph $A(\mathcal{D}, \mathcal{R})$ gibt die durch die Regelbasis $\mathcal{R}'$ und die Konsistenzinformation $\mathcal{K}$ festgelegten, funktionalen Verknüpfungen der Datenobjekte in der Datenbank $\mathcal{D}$ wieder. Der Graph ist folgendermaßen definiert:*

$$A(\mathcal{D}, \mathcal{R}) := (V, E). \tag{4.16}$$

Es gelten die folgenden Definitionen:

- V *ist die Menge der Knoten und identisch mit der Menge der Datenobjekte x_i in der Datenbank, $V := \mathcal{D}$.*
- E *ist die Menge der gerichteten Kanten k_{ij}. Diese sind folgendermaßen definiert:*

$$k_{ij} := (x_i, x_j),\ \forall \rho_{ik} \in \mathcal{R}, x_j \in W_{\tau_{ik}} \text{ falls } A_{ik} = \tau_{ik}. \tag{4.17}$$

Falls notwendig, kann der Analysegraph $A(\mathcal{D}, \mathcal{R})$ auch auf die beiden Teilbereiche der $\mathcal{R}'$ und $\mathcal{K}$ der Regelbasis $\mathcal{R}$ eingeschränkt werden.

Abbildung 4.9 zeigt die graphische Darstellung eines Analysegraphen. Die Zugriffe der Applikation sind zur besseren Übersicht durch gestrichelte Pfeile angedeutet. Sie sind nicht Element des Analysegraphen.

Mit Hilfe des Analysegraphen $A(\mathcal{D}, \mathcal{R})$ lassen sich zu jedem Datenobjekt $x_i \in \mathcal{D}$ diejenigen Umweltdaten ermitteln, von denen x_i abhängt und die letztendlich eine Neuberechnung von x_i bewirken können:

Definition 14 (Ausgangsdaten) *Die Ausgangsdaten $\mathcal{A}_i$ eines Datums $x_i \in \mathcal{D}$ sind alle Daten $\mathcal{A}_i = \{x_1, x_2, \ldots, x_n\}$ des Umweltabbildes, von denen x_i in $A(\mathcal{D}, \mathcal{R})$ erreichbar ist. Ist x_i ein Datum des Umweltabbildes, so ist es mit seinen Ausgangsdaten identisch.*

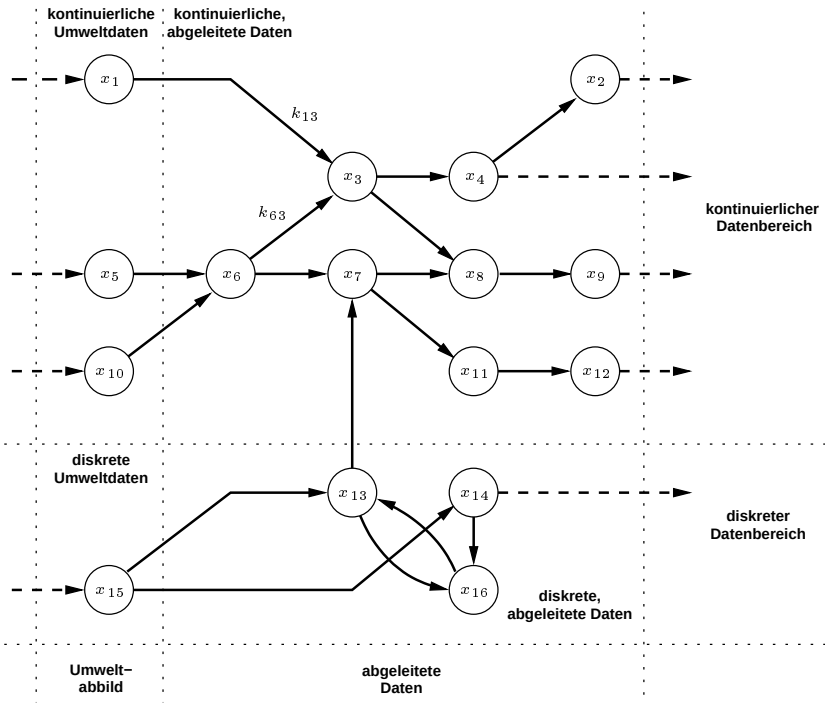


Abbildung 4.9.: Beispiel für einen Analysegraphen $A(\mathcal{D}, \mathcal{R})$ zur Visualisierung der Abhängigkeitsbeziehungen zwischen den Objekten der Datenbank. Gestrichelte Pfeile symbolisieren lesende bzw. schreibende Zugriffe der Applikation, sie sind nicht Bestandteil von $A(\mathcal{D}, \mathcal{R})$.

Betrachtet man den Analysegraphen in Abbildung 4.9, so sind z. B. die Datenobjekte $\mathcal{A}_2 = \{x_1, x_5, x_{10}\}$ die Ausgangsdaten von x_2.

Auf Grund der Trennung der Umweltdaten in kontinuierliche und diskrete Datenobjekte kann der gesamte Datenbestand einer Applikation entsprechend der Abhängigkeit von diskreten bzw. kontinuierlichen Ausgangsdaten in einen kontinuierlichen und einen diskreten Datenbereich unterteilt werden. Ein Datum, das von mindestens einem kontinuierlichen Datum abhängt, ist dabei immer dem kontinuierlichen Datenbereich zugeordnet. Diese Unterteilung der Daten in kontinuierliche und diskrete Daten bzw. Umweltdaten und abgeleitete Daten ist in Abbildung 4.9 zusätzlich eingezeichnet.

Zeitanforderungen

Zeitanforderungen können für die temporale Konsistenz und den Jitter von kontinuierlichen Datenobjekten gestellt werden. Es können Anforderungen an die Zeitspanne festgelegt werden, die nach dem Setzen eines diskreten Umweltdatums bis zur Wiederherstellung der vollständigen logischen Konsistenz innerhalb der Datenbank verstreichen darf. Zusätzlich kann die Zeit, die bis zur Abarbeitung einer Regel vergehen kann, definiert werden:

- *Temporale Konsistenz kontinuierlicher Daten.* Für alle kontinuierlichen Datenobjekte, auf die aus der Applikation heraus zugegriffen wird, ist das Setzen von Zeitbedingungen für die Aktualität deren kontinuierlicher Ausgangsdaten und damit auch auf die Zeit für die Abarbeitung der Konsistenzinformation in der folgenden Form möglich:
  - $a_k(x_i) \leq T_{ik},\ \forall x_i \in \mathcal{A}_k,$

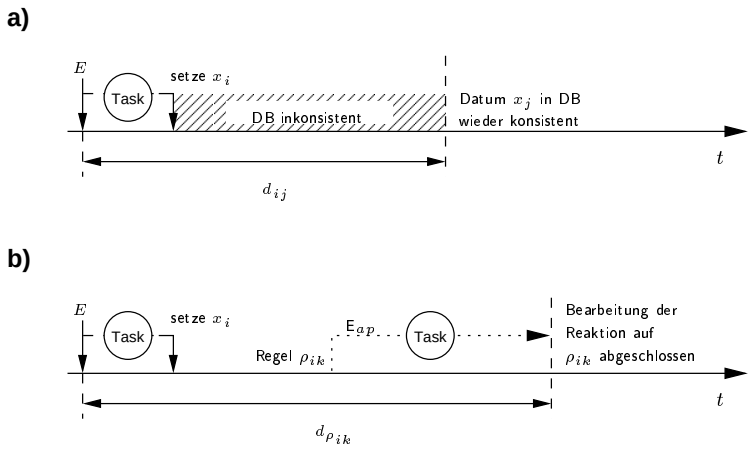


Abbildung 4.10.: Zur Definition von Zeitbedingungen a) für die Abarbeitung der Konsistenzinformation bei diskreten Umweltdaten und b) bei der Bearbeitung von Regeln.

- $|\, a_k(x_i) - a_k(x_j)\,| \leq T'_k(x_i, x_j),\ \forall x_i, x_j \in \mathcal{A}_k$.

- *Jitter kontinuierlicher Umweltdaten.* Der Jitter $j_k(x_i)$ eines kontinuierlichen Umweltdatums x_i legt fest, um welche Zeitspanne aus „Sicht“ des Datums x_k der Aufnahmezeitpunkt $t_s(x_i)$ durch den zugehörigen Sensor vom eigentlich spezifizierten, idealen Zeitpunkt $t_{s,\text{ideal}}(x_i)$ abweichen darf:

$$j_k(x_i) = |t_s(x_i) - t_{s,\text{ideal}}(x_i)|. \tag{4.18}$$

- *Konsistenz diskreter, abgeleiteter Datenobjekte.* Das Setzen eines diskreten Umweltdatums x_i wird stets durch ein Ereignis E in der Umwelt ausgelöst, das die Bearbeitung eines Tasks und einer darin eingebetteten Transaktion startet. Es ist der Zeitraum d_{ij} zu spezifizieren, der zwischen dem Auftreten von E und dem Anpassen eine Datums x_j in der Datenbank vergeht, das von $x_i \in \mathcal{A}_j$ abhängt und auf das von der Applikation zugegriffen werden kann (siehe Abbildung 4.10 a).

- *Regeln, die mit internen Ereignissen der Applikation verknüpft sind.* Die Definition einer Zeitanforderung für eine Regel besteht bei diskreten Datenobjekten darin, die maximale Zeitspanne $d_{\rho_{ik}}$ festzulegen, die zwischen dem auslösenden Ereignis in der Umwelt E und dem Abschluß der Reaktion des Systems maximal verstreichen darf (siehe hierzu Abbildung 4.10 b). Bei kontinuierlichen Datenobjekten wird festgelegt, welches Alter $T_{\rho_{ik}}(x_i)$ ein Umweltdatum x_i, das ursprünglich eine Regel ausgelöst hat, zu dem Zeitpunkt maximal haben darf, zu dem die Reaktion der Applikation auf das Setzen des Datums x_i abgeschlossen ist.

4.3.2. Abarbeitungsmodell

Das Abarbeitungsmodell definiert, wie und mit welcher Deadline die Informationen des Wissensmodells bearbeitet und in die Implementierung umgesetzt werden müssen, um die logische und temporale Konsistenz bzw. die Erfüllung aller Zeitanforderungen an die abgeleiteten Datenobjekte und an die Regeln sicherzustellen.

Im folgenden Abschnitt wird zuerst der Kopplungsmodus Transaktion - Regelbearbeitung behandelt. Die darauf folgenden Abschnitte behandeln dann ein allgemeines Verfahren, um die

absolute temporale Konsistenz und den Jitter kontinuierlicher Umweltdaten zu realisieren, und um die rechtzeitige Bearbeitung der Regeln im kontinuierlichen und diskreten Datenbankanteil sicherzustellen. Im Anschluß daran werden Möglichkeiten zur Optimierung dieses Verfahrens dargestellt.

Kopplungsmodus

Transaktionen können durch den Benutzer, nicht aber durch die Datenbank abgebrochen werden. Um Probleme beim Zurückrollen von Transaktionen mit eventuell irreversiblen Aktionen aus Regeln heraus zu vermeiden, wird sowohl die Auswertung einer Regel als auch die Ausführung der zugeordneten Aktion im Modus *sequential causallay dependent* durchgeführt. Das heißt, alle mit einem Datum verknüpften Regeln werden *nach* dem erfolgreichen „commit“ der auslösenden Transaktion bearbeitet (siehe Abschnitt 2.2.2). Innerhalb einer Transaktion sind damit Regeln, die mit dem Datenbestand dieser Transaktion selbst verknüpft sind, nicht wirksam. Dies ist notwendig, um – trotz Regeln mit „nach außen“ sichtbaren Aktionen (real actions) – jederzeit ein Zurückrollen der Transaktion durch den Benutzer zu ermöglichen.

Zyklen und mehrfacher Datenfluß

Durch die Konsistenzbeziehungen der Datenobjekte und durch die zusätzlichen Abhängigkeiten der Daten untereinander, die in der Regelbasis festgelegt sind, können in den Abhängigkeitsrechnungen Zyklen entstehen und es kann der Fall eintreten, daß ein Datum über mehrere unterschiedliche Abhängigkeitsketten mit einem anderen Datum der Datenbank verknüpft ist. Diese beiden Fälle sind, um die Korrektheit der Datenbank sicherzustellen, gesondert zu betrachten.

Zyklen Zyklen im Analysegraphen $A(\mathcal{D}, \mathcal{R})$ führen dazu, daß bei sukzessiver Bearbeitung aller Regeln diese Rechnung nicht terminiert, da stets wieder die Regel am Beginn einer Abhängigkeitsrechnung ausgelöst wird. Das Auftreten solcher Zyklen in der Datenbankdefinition ergibt sich in der Regel automatisch aus der wechselseitgen Abhängigkeit der verschiedenen Datenobjekte und kann daher im Design nicht vermieden werden.

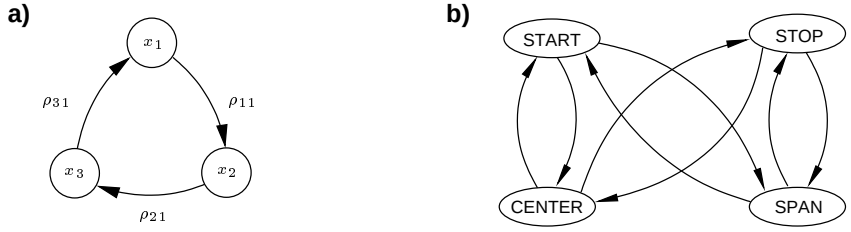


Abbildung 4.11.: a) Zyklus im Analysegraphen und damit im Ausgleich der Datenabhängigkeiten. b) Beispiel zyklischer Datenabhängigkeiten für den Fall einer parametrierbaren, graphischen Meßwertdarstellung.

Ein einfaches Beispiel gibt Abbildung 4.11 a). Abbildung 4.11 b) zeigt den Fall einer vom Benutzer anhand von `CENTER/SPAN` und `START/STOP` parametrierbaren, graphischen Meßwertdarstellung. Können jeweils beide Parameterpaare vom Anwender geändert werden, sind zyklische Abhängigkeiten nicht zu umgehen.

Zur Vermeidung von Endlosschleifen bei der Datenbearbeitung sind zwei Strategien denkbar: i) Ein Datum wird nur dann in der Datenbank gesetzt, falls sich sein Wert geändert hat. Dann

ist allerdings nicht mehr sichergestellt, daß immer alle Ereignisse zur Laufzeit wie spezifiziert ausgelöst werden. ii) Die Endlosschleife wird durch Entfernen einer bestimmten Regel der Datenbank unterbrochen. In diesem Fall ist eine vollständige Konsistenzrechnung für alle Datenobjekte nicht mehr sichergestellt. Würde man in Abbildung 4.11 a) beispielsweise ρ_{11} entfernen, so würden beim Setzen von x_1 fälschlicherweise weder x_2, noch x_3 neu berechnet werden.

Die Unterbrechung von Zyklen muß daher zur Laufzeit geschehen. Dazu müssen in der Implementierung Mechanismen vorgesehen werden, die zyklische Mehrfachberechnung von Daten erkennen und unterdrücken (siehe Abschnitt 4.4.3). Des weiteren muß durch die Analyse der funktionalen Abhängigkeiten sichergestellt werden, daß nach dem einmaligen Durchlaufen eines Zyklus sich für ein Datum stets dieselben Inhalte ergeben (Confluence) [AWH92].

Mehrfacher Datenfluß Ist ein Datum x_j innerhalb von $A(\mathcal{D}, \mathcal{R})$ von einem anderen Datum x_i auf unterschiedlichen, zumindestens teilweise disjunkten Wegen erreichbar, so ist, wie im Fall von Zyklen, zu überprüfen, ob die auf den verschiedenen Wegen erzielten Ergebnisse für x_j stets identisch sind (Confluence). Ein Beispiel für mehrfachen Datenfluß gibt Abbildung 4.9. Dort ist x_8 von x_6 über x_3 und x_7 erreichbar.

Bearbeitung kontinuierlicher Umweltdaten

Bei der Bearbeitung der kontinuierlichen Datenobjekte ist zum einen die temporale Konsistenz der Umweltdaten sicherzustellen, das heißt, es muß vom Anwendungsentwickler durch die Implementierung von Tasks mit geeigneter Periode und Deadline dafür gesorgt werden, daß die Umweltdaten ausreichend aktuell in der Datenbank zur Verfügung stehen. Zum anderen ist durch die Implementierung die rechtzeitige Bearbeitung der Regeln und damit die temporale Konsistenz der abgeleiteten Daten zu gewährleisten.

Temporale Konsistenz Für die kontinuierlichen Umweltdaten x_i können aus den Zeitanforderungen, die für eine Applikation spezifiziert wurden, die folgenden Grenzwerte abgeleitet werden:

1. $T_{\max}(x_i) = \min\left[\, T_{ik}, T_{\rho_{lm}}(x_i) \,\right],\ \forall x_k, \rho_{lm}$ (Alter),
2. $j_{\max}(x_i) = \min\left[\, j_k(x_i) \,\right],\ \forall x_k$ (Jitter),
3. $T'_{\max}(x_i, x_j) = \min\left[\, T'_k(x_i, x_j) \,\right],\ \forall x_k$ (Dispersion).

Im Abarbeitungsmodell bzw. in der Implementierung wird jedem kontinuierlichen Umweltdatum x_i ein separater Task T_i zugeordnet. Dieser liest die Sensoren aus, die dem kontinuierlichen Umweltdatum x_i zugeordnet sind, faßt deren Ergebnisse zusammen und legt sie über die Transaktion τ_i^k im Datum x_i ab. Um die temporale Konsistenz des Datums x_i zu gewährleisten, wird der Task T_i periodisch mit dem Intervall Δt_i und der Deadline d_i bearbeitet. Δt_i und d_i müssen die folgenden Bedingungen erfüllen[5]:

$$\Delta t_i + d_i \leq T_{\max}(x_i), \tag{4.19}$$

$$d_i \leq \Delta t_i. \tag{4.20}$$

[5] Dabei wird hier und in den folgenden Herleitungen davon ausgegangen, daß die Rechenzeit vom Start der Task bis zur Auslese der Sensoren vernachlässigt werden kann. In jedem Fall ist aber, falls die Ungleichungen erfüllt sind, die temporale Konsistenz sichergestellt.

Die genaue Aufteilung von $T_{\max}(x_i)$ auf die beiden Parameter ist nicht festgelegt. Eine mögliche Wahl für diese beiden Parameter ist beispielsweise $\Delta t_i = d_i = \frac{1}{2} T_{\max}(x_i)$. Sie kann mit Hilfe der Ergebnisse aus der Realzeitanalyse (Abschnitt 5.2.2) optimiert werden.

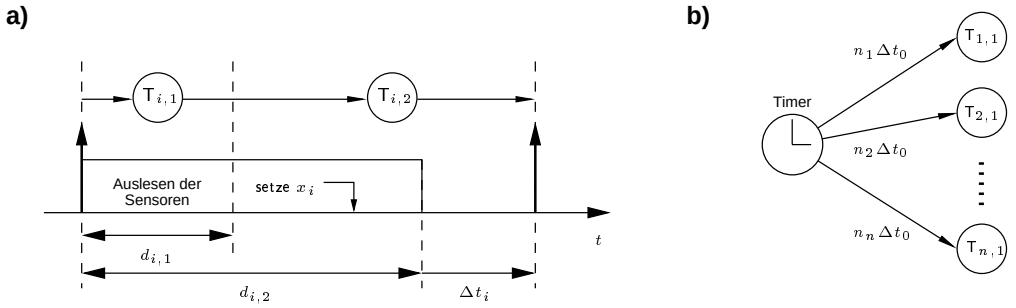


Abbildung 4.12.: a) Wahl der Deadlines zur Realisierung von $j_{\max}(x_i) < d_i$. b) Synchronisierung der Sensorauslese bei der Auffrischung von kontinuierlichen Umweltdaten zur Realisierung von Dispersionsrelationen.

Jitter Ist $j_{\max}(x_i) < d_i$, muß zur Erfüllung der Zeitanforderungen T_i, wie in Abbildung 4.12 a) dargestellt, in zwei Tasks $T_{i,1}$ und $T_{i,2}$ aufgeteilt werden. Im ersten Task werden lediglich die Sensoren ausgelesen. Im zweiten Task werden die Sensordaten verarbeitet und das Ergebnis x_i in der Datenbank gesetzt. Die Abtastfrequenz Δt_i der Tasks bleibt dabei unverändert. Für die Deadlines der beiden Tasks gilt:

$$\begin{aligned} d_{i,2} &= d_i, \\ d_{i,1} &\leq j_{\max}(x_i), \\ d_{i,1} &< d_{i,2}. \end{aligned} \tag{4.21}$$

Dispersion Sind Dispersionsrelationen zwischen zwei verschiedenen Umweltdaten x_i und x_j gegeben, müssen die Auslesevorgänge der einzelnen Daten zeitlich synchronisiert, d. h. die Auslesefrequenzen einander angeglichen und die auslesenden Tasks T_i bzw. $T_{i,1}$ gemeinsam gestartet werden (siehe Abbildung 4.12 b). Die maximal zulässigen Deadlines d_i und d_j sind dann zusätzlich zu (4.19) durch die folgende Gleichung nach oben beschränkt:

$$\max[\, d_i, d_j \,] < T'_{\max}(x_i, x_j). \tag{4.22}$$

Auf eine Synchronisierung der Auslese zweier unterschiedlicher Datenobjekte kann verzichtet werden, falls gilt:

$$\max\,[\, a_{\max}(x_i), a_{\max}(x_j) \,] < T'_{\max}(x_i, x_j). \tag{4.23}$$

Sind Dispersionsrelationen für mehrere unterschiedliche Umweltdaten gegeben, muß innerhalb eines solchen Systems abhängiger Daten die Gleichung (4.22) dementsprechend paarweise für diese Daten erfüllt sein.

Falls zwar die Dispersion zwischen zwei Datenobjekten eingeschränkt ist, die Zeitbeschränkungen an das Alter der beiden Daten aber stark unterschiedlich sind, können die Startzeitpunkte auch ganzzahlige Vielfache einer minimalen Zeiteinheit Δt_0 sein (siehe Abbildung 4.12 b).

Regelbearbeitung bei kontinuierlichen Ausgangsdaten

Die rechtzeitige und effiziente Abarbeitung von Regeln bzw. von Konsistenzinformation bei kontinuierlichen Datenobjekten stellt, besonders bei großer Tiefe der Abhängigkeiten in der Datenbank, ein komplexes Optimierungsproblem dar, das bisher auch in der Literatur nicht ausreichend behandelt wird. Das häufig angewendete Verfahren, asynchrone, periodische Rechenprozesse zur Konsistenzsicherung zu verwenden [ABRW93][GHS95], ist vor allem auf Grund der großen Zahl der hierfür notwendigen Tasks und der daraus resultierend häufigen Taskwechsel für das hier betrachtete Anwendungsgebiet nicht geeignet.

In dieser Arbeit wird daher vorerst ein allgemeines Vorgehen zur Bearbeitung der Datenbankregeln vorgestellt, das die rechtzeitige Abarbeitung aller Zusammenhänge und damit temporale Konsistenz der Daten garantiert. Die Bearbeitung der Abhängigkeiten in der Datenbank erfolgt nach den folgenden Regeln:

1. Nach dem Setzen eines Datums x_i wird am Ende der Transaktion die Kette aller mit diesem Datum verknüpften und der in der Folge davon kaskadierend getriggerten Regeln abgearbeitet. Die neue Information pflanzt sich so in die Datenbank hinein fort. Zyklen in der Abhängigkeitsberechnung (siehe vorne) werden zur Laufzeit von der Datenbank erkannt und nach dem ersten Durchlauf abgebrochen.

2. Alle Berechnungen werden im Kontext des Tasks T_i bzw. $\mathrm{T}_{i,2}$ mit der Deadline d_i bzw. $d_{i,2}$ durchgeführt.

Wendet man dieses Verfahren auf das in Abbildung 4.9 dargestellte System an, werden zum Beispiel nach dem Setzen des Datums x_1 – unter der Annahme, daß $j_{\max}(x_1)$ nicht definiert ist – die Datenobjekte x_3, x_4, x_2, x_8 und x_9 im Kontext des Tasks T_1 mit der Deadline d_1 aktualisiert.

Das dargestellte Vorgehen liefert vorhersagbare Ergebnisse und ist einfach automatisierbar, allerdings ist die darin vorgenommene pauschale Zuordnung einer Deadline zu einem Datum bezüglich der Rechenzeitausnutzung nicht optimal. In Abschnitt 4.3.3 werden daher Optimierungspotentiale für häufiger auftretende Situationen vorgestellt, die vom Anwender eingesetzt werden können, falls der Rechenzeitbedarf zu hoch ist. Sie setzen zusätzliches Wissen über die Applikation voraus und gehen zu Lasten der Allgemeingültigkeit und der Dynamik der Regelbasis. Zudem sind Eingriffe in die Wissensbasis und die Implementierung notwendig und es wird eine größere Anzahl an Tasks für die Implementierung benötigt. Die Angabe eines geschlossenen Algorithmus zur Optimierung ist nicht möglich, da schon für die Zuweisung von Δt_i und d_i bei mehreren Transaktionen ein solcher nicht existiert (siehe Abschnitt 5.2.2). Der Anwender kann diese Methoden jedoch, basierend auf den Ergebnissen einer Realzeitanalyse, gezielt an problematischen Stellen der Applikation zur Transformation der Datenbasis einsetzen.

Diskrete Datenobjekte

Werden innerhalb eines Task T in der Transaktion τ_l diskrete Datenobjekte in der Datenbank gesetzt, so werden alle Regeln, die direkt oder indirekt mit den Daten dieser Transaktion verknüpft sind, am Ende von τ_l im Kontext des Tasks T iterativ ausgewertet und bearbeitet. Auch hier werden Zyklen in der Abhängigkeitsberechnung zur Laufzeit von der Datenbank erkannt und nach dem ersten Durchlauf abgebrochen.

Die Deadline d für die Ausgleichsrechnung ergibt sich aus den Zeitanforderungen für die aktive

Datenbank folgendermaßen:

$$d = \min_{i,k}[\, d_{\rho_{ik}}\,], \forall \rho_{ik} \text{ mit } x_n \in \mathcal{A}_i \wedge x_n \in \mathrm{W}_l. \tag{4.24}$$

Gleichzeitig muß d die folgende Ungleichung bezüglich der Deadlines des einbettenden Tasks d_j und des im Taskpräzedenzsystem nachfolgenden Tasks d_{j+1} erfüllen:

$$d_j \leq d \leq d_{j+1}. \tag{4.25}$$

4.3.3. Optimierungspotentiale

Dieser Abschnitt erläutert Optimierungsmöglichkeiten für das weiter oben vorgestellte allgemeine Verfahren zur Bearbeitung der Datenbankregeln. Eine exemplarische Untersuchung des Zeitverhaltens im Hinblick auf einen Nachweis der Realzeitfähigkeit findet sich in Abschnitt 5.2.2.

Fusion von kontinuierlichen Datenflüssen

Sehr häufig werden im kontinuierlichen Anteil der Datenbank in einem Datum x_1 verschiedene Datenströme mit unterschiedlichen Zeitanforderungen über die Konsistenzinformation miteinander verknüpft. Im einfachsten Fall werden die Daten von zwei unterschiedlichen Sensoren fusioniert. Abbildung 4.13 a) gibt ein Beispiel.

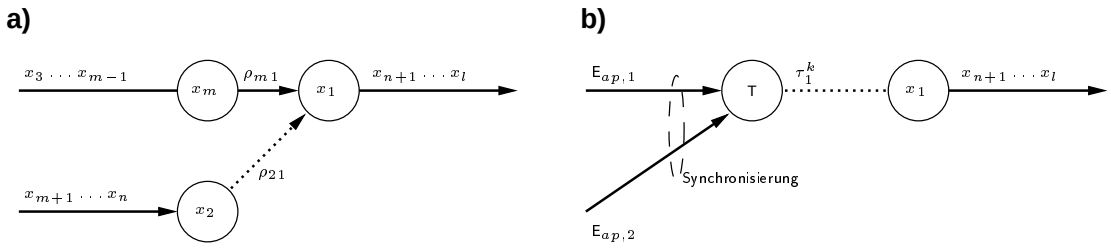


Abbildung 4.13.: Fusion zweier kontinuierlicher Datenströme in einem Datenobjekt x_1. a) Optimierung der Implementierung durch die Entnahme der Regel ρ_{21} aus der Regelbasis im 1. Fall, $T_{\min,1} \ll T_{\min,2}$. b) Optimierung durch einen zusätzlichen Task im 2. Fall.

Legt man nach Abbildung 4.13 a) die beiden Definitionen

$$T_{\min,1} = \min_{\substack{i=3\ldots m,\\ j=(n+1)\ldots l}} [\, T_{ij}\,] \text{ und} \tag{4.26}$$

$$T_{\min,2} = \min_{\substack{i=(m+1)\ldots n,\\ j=(n+1)\ldots l}} [\, T_{ij}\,] \tag{4.27}$$

zu Grunde, so sind die folgenden zwei Fälle zu unterscheiden:

1. Fall: $T_{\min,1} \ll T_{\min,2}$.

Um hier eine effizientere Implementierung zu erreichen und überflüssige Berechnungen von Abhängigkeiten zu sparen, kann in der Regel auf die Weiterrechnung der Abhängigkeiten

aus dem Zweig mit den geringeren Zeitanforderungen heraus verzichtet werden. Die Werte von T_{ij} müssen dann wie folgt angepaßt werden:

$$T_{ij} \longrightarrow T_{ij} - T_{\min,1}, \tag{4.28}$$

$$i = (m+1) \ldots n,\ j = (n+1) \ldots l. \tag{4.29}$$

Im Beispiel der Abbildung 4.13 a) heißt das, daß die Regel ρ_{21} aus dem Regelsatz der Konsistenzinformation entnommen werden kann.

Soll in einem solchen Fall die mittlere Auslastung der CPU, trotz Verschärfung der Bedingungen für die temporale Konsistenz, sinken, muß für den Rechenzeitgewinn Δc, der durch das Weglassen von Regel ρ_{21} auftritt, folgende Ungleichung gelten ($\bar{c}$ bezeichnet hierbei die mitteler Rechenzeit, die für die Berechnung der Regeln aus dem „Ast“ $x_{m+1} \ldots x_n$, benötigt wird):

$$\bar{c}\,\frac{T_{\min,1}}{T_{\min,2}} < \Delta c. \tag{4.30}$$

2. Fall: Gleichung (4.30) ist nicht erfüllt.

In diesem Fall macht eine Verkleinerung von $T_{\min,2}$ keinen Sinn und es ist besser, den Sensor-Update für die beiden in x_1 zusammentreffenden Äste zeitgleich mit der Deadline d zu starten und die Weiterverarbeitung bei x_1 zu synchronisieren. Das heißt, die Regelbasis wird wie folgt umgeformt (siehe Abbildung 4.13 b):

$$\rho_{m1} \rightarrow \rho'_{m1} = (w(x_m), t, E_{\mathrm{ap},1}), \tag{4.31}$$

$$\rho_{21} \rightarrow \rho'_{21} = (w(x_2), t, E_{\mathrm{ap},2}), \tag{4.32}$$

In der Implementierung wird zusätzlich Task T mit der Deadline d eingeführt, der die beiden Ereignisse $\mathrm{E}_{\mathrm{ap},1}$ und $\mathrm{E}_{\mathrm{ap},2}$ synchronisiert und die Transaktion τ_1^k inklusive aller folgenden Regeln bearbeitet.

Aufspaltung eines kontinuierlichen Datenflusses

Der genau umgekehrte Fall liegt vor, falls von einem kontinuierlichem Datum x_1 Datenströme mit unterschiedlichen Zeitanforderungen ausgehen. Dies ist in Abbildung 4.14 für das Datum x_1 schematisch dargestellt.

Gilt in diesem Fall

$$T_{\min,1} = \min_{\substack{i=4\ldots m,\\ j=(m+1)\ldots n}} [T_{ij}], \tag{4.33}$$

$$T_{\min,2} = \min_{\substack{i=4\ldots m,\\ j=(n+1)\ldots l}} [T_{ij}], \tag{4.34}$$

$$T_{\min,1} < T_{min,2}, \tag{4.35}$$

kann der Zweig mit den niedrigeren Zeitanforderungen abgespalten und inklusive aller folgenden, rekursiv zu bearbeitenden Regeln einem neuen Task T zugeteilt werden. Dazu muß die Regel ρ_{12} wie folgt umformuliert werden:

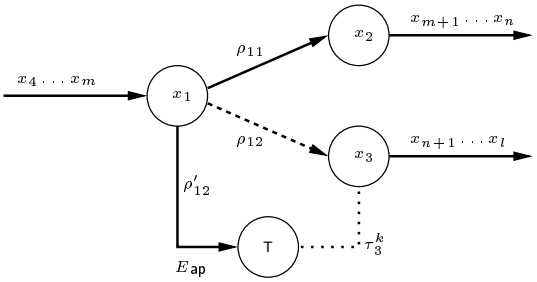


Abbildung 4.14.: Aufspaltung eines kontinuierlichen Datenflusses in zwei neue Datenflüsse mit unterschiedlichen Zeitanforderungen.

$$\rho_{12} \longrightarrow \rho'_{12} = (x(x_1), w, E_{\mathrm{ap}}). \tag{4.36}$$

Der Task T, gestartet durch E_{ap}, kann dann ab der Transaktion τ_3^k mit einer geringeren Deadline d arbeiten. Dabei muß gelten:

$$d \leq \min[\,\Delta t_i\,],\ i = 4 \ldots m. \tag{4.37}$$

Zusätzlich besteht die Möglichkeit, falls es die Zeitanforderungen erlauben ($\Delta t_2 > n\ \Delta t_1$), im Task T nur auf jedes n-te Ereignis E_{ap} zu reagieren, und so die Rechenlast weiter zu verringern.

Komplexere Situationen lassen sich durch die Kombination der beiden dargestellten Fälle behandeln. Abbildung 4.15 zeigt ein Beispiel.

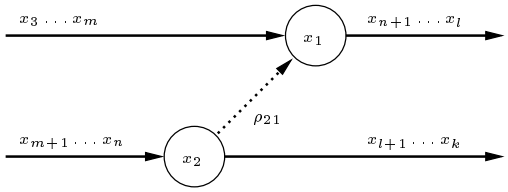


Abbildung 4.15.: Informationsaustausch zwischen unterschiedlichen Datenflüssen.

Gilt hier

$$T_{\mathrm{min},1} = \min_{\substack{i=3\ldots m,\\ j=(n+1)\ldots l}} [\,T_{ij}\,], \tag{4.38}$$

$$T_{\mathrm{min},2} = \min_{\substack{i=(m+1)\ldots n,\\ j=(n+1)\ldots l}} [\,T_{ij}\,], \tag{4.39}$$

$$T_{\mathrm{min},1} \ll T_{min,2}, \tag{4.40}$$

kann die Regel ρ_{21} aus der Regelbasis entfernt werden. Die Zeitbedingungen für $x_{m+1} \ldots x_n$ müssen nach Formel (4.28) angepaßt werden. Ein zusätzlicher Task T ist in diesem Fall nicht notwendig, da dieser in dem Task aufgeht, der für die Bearbeitung des Datums x_1 verantwortlich ist.

Unterschiedliche Zeitanforderungen bezüglich eines Ausgangsdatums

In der Regel sind von einem Umweltdatum x_1 mehrere Datenobjekte der Datenbank, auf die aus der Applikation heraus zugegriffen wird, abhängig. Diese können jeweils unterschiedliche

Zeitanforderungen haben (siehe Abbildung 4.16).

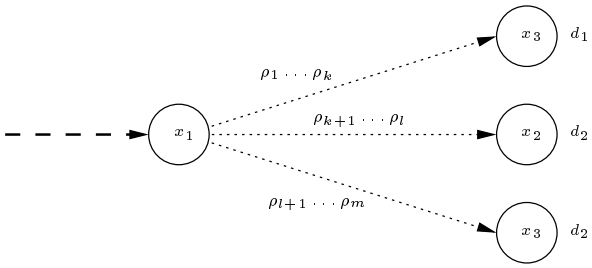


Abbildung 4.16.: Unterschiedliche Zeitschranken für die von einem Ausgangsdatum ausgehende Informationsausbreitung in der Datenbank. Es gilt $d_1 \leq d_2 \leq d_3$.

Um in der Implementierung eine Entspannung der Deadlines zu erreichen, kann die Bearbeitung der Regeln für das Datum x_1 folgendermaßen sortiert werden (die Deadline des ausführenden Tasks wird dabei entsprechend schrittweise angepaßt):

$$(d = d_1) \;\rightarrow\; \rho_1 \ldots \rho_k \;\rightarrow\; (d = d_2) \;\rightarrow\; \rho_{k+1} \ldots \rho_l \;\rightarrow\; (d = d_3) \;\rightarrow\; \rho_{l+1} \ldots \rho_m. \tag{4.41}$$

Hierbei ist darauf zu achten, daß bei kontinuierlichen Daten die Deadlines die folgende Bedingung nicht verletzen:

$$d_j \leq \Delta t_1. \tag{4.42}$$

Befindet sich der ausführenden Task in einem Präzedenzsystem, muß die längste Deadline, die bei der Bearbeitung der Konsistenzinformation auftreten kann, stets kleiner als die Deadline des im Präzedenzsystem nachfolgenden Tasks sein.

4.4. Implementierung

4.4.1. Allgemeines

Das Concurrency-Control Protokoll PRED-DF und die Funktionen der Datenbank, die für die aktive Komponente der Datenbank notwendig sind, wurde im Rahmen dieser Arbeit zur Verifikation der grundlegenden Funktionsprinzipien prototypisch implementiert. Dies geschah im Zuge der Entwicklung einer aktiven Realzeitdatenbank für eingebettete System innerhalb des von der Bayerischen Forschungsstiftung geförderten Forschungsprojektes FORSOFT II, Teilprojekt HRS. Die Entwicklung erfolgte für ein Ein-Prozessor System und wurde, nach objektorientierten Gesichtspunkten strukturiert, in der Programmiersprache C durchgeführt, da die Programmiersprache C++ augenblicklich von vielen Realzeitbetriebssystemen noch nicht ausreichend unterstützt wird.

Abbildung 4.17 gibt einen Überblick über die Systemarchitektur.

Um bei der Implementierung eine möglichst hohe Effizienz und einen geringen Overhead zu erzielen, wurde auf die Implementierung einer eigenständigen Datenbank nach dem traditionellen Client-Server Ansatz verzichtet, da insbesondere der Anteil an im Verhältnis zur Tasklaufzeit kurzen Zugriffe auf die Datenbank im betrachteten Anwendungsbereich überproportional groß ist (z. B. Update-Transaktionen auf Grund neuer Sensordaten). Statt dessen wird die

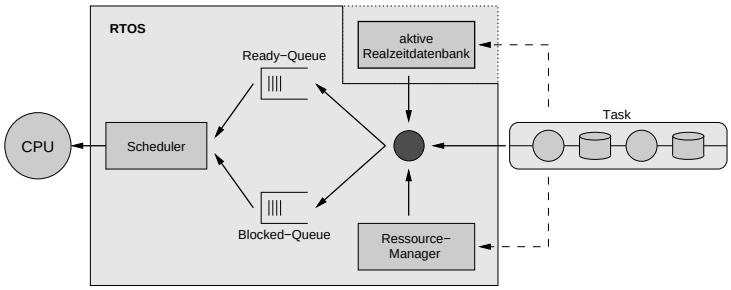


Abbildung 4.17.: Überblick über die Systemarchitektur der prototypischen Implementierung.

Abarbeitung der Transaktionen in die Applikationstasks selbst verlegt. Unnötig häufige und teure Taskwechsel zwischen Applikation und Datenbankmanager werden so vermieden. Das Realzeit-Betriebssystem dient bei diesem Ansatz gleichzeitig implizit auch als Transaktions-Managementsystem. Die Transaktionen erben die Priorität des einbettenden Tasks. Von der Datenbank wird entsprechend dem Concurrency-Control Protokoll der Zustand der Tasks, die eine Transaktion ausführen, manipuliert. Die Datenbank ist als Library implementiert und wird zur Applikation gebunden. Die Daten werden in einem gemeinsam genutzten Speicherbereich abgelegt und über zentrale Datenstrukturen der Datenbank verwaltet. Datenbankkommandos sind Funktionsaufrufe.

4.4.2. Concurrency-Control Protokoll PRED-DF

Entsprechend der Klassifizierung von Transaktionen in Abschnitt 4.2.2 werden dem Benutzer für normalisierte Transaktionen neben `TA_start` und `TA_commit` zwei zusätzliche Datenbankkommandos zur Verfügung gestellt: `TA_freeReadSet` und `TA_lockWriteSet`. Wie in Abbildung 4.18 dargestellt, dient `TA_freeReadSet` dazu, den Lock des Lese-Datensatz einer Transaktion bezüglich den Transaktionen im Friend-Set aufzuheben, `TA_lockWriteSet` sperrt den Schreib-Datensatz auch für im Friend-Set enthaltene Transaktionen. Da Logging und Undo-Mechanismen nicht Gegenstand dieser Arbeit sind, wird der Abort von Transaktionen hier nicht näher behandelt.

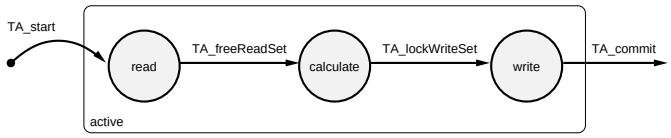


Abbildung 4.18.: Datenbankkommandos und Zustandsübergänge beim Wechseln der Lebenszyklus-Phasen einer normalisierten Transaktion im Concurrency-Control Protokoll PRED-DF.

Die Implementierung des Locking-Mechanismus selbst beruht auf einem Konfliktzähler und zwei Konfliktlisten, die jeder einzelnen Transaktion zugeordnet sind. Diese werden, gemeinsam mit anderen Informationen, die von PRED-DF und der Datenbank benötigt werden, im Transaktions-Kontrollblock `TA_CTRLBLK` verwaltet. Dessen wichtigste Elemente sind in Tabelle 4.2 zusammengefaßt.

Der Konfliktzähler `conflictCounter` einer Transaktion τ gibt darüber Auskunft, durch wieviele andere, aktuell im System befindliche Transaktionen τ blockiert wird (scheduled transactions, siehe Abbildung 4.1). Das heißt, wieviele andere Transaktionen Datenobjekte, die von τ benötigt werden, im Augenblick gerade nutzen oder diese auf Grund ihrer höheren Priorität vor τ zugeteilt

TA_CTRLBLK	
conflictCounter	Anzahl der Transaktionen im System, die τ augenblicklich blockieren
unlockAfterFreeReadSet	Liste aller Transaktionen, die von τ blockiert werden und die von τ bei TA_freeReadSet entsperrt werden müssen
unlockAfterCommit	Liste aller Transaktionen, die von τ blockiert werden und die von τ bei TA_commit entsperrt werden müssen
status	„scheduled“ oder „undefined“
prio	Priorität der Transaktion
executingStatus	Status der Ressourcenbearbeitung (siehe Text)
originalPrio	ursprüngliche Priorität der Transaktion
afterFreeReadSetPrio	Priorität der Transaktion nach verlassen der „write“-Phase
blockedBy	Liste aller Transaktionen mit kleinerer Priorität als τ, die τ blockieren

Tabelle 4.2.: Tabelle der wichtigsten Elemente des Transaktions-Kontrollblocks TA_CTRLBLK einer Transaktion τ.

bekommen werden. Eine Transaktion kann erst mit der Bearbeitung beginnen bzw. fortfahren, wenn sie alle zur Ausführung notwendigen Datenobjekte besitzt, d. h. der conflictCounter identisch Null ist. Der Wert des Konfliktzählers einer Transaktion wird nach jeder Änderung überprüft und dementsprechend der Zustand des Tasks, der die betreffende Transaktion enthält, durch die Datenbank manipuliert (von „ready“ nach „blocked“ oder umgekehrt). Die beiden Konfliktlisten unlockAfterFreeRead und unlockAfterCommit einer Transaktion τ enthalten alle Transaktionen, die von τ blockiert werden und bei TA_freeReadSet bzw. TA_commit von dieser wieder freigegeben werden müssen. Das heißt, der Konfliktzähler aller in diesen Listen enthaltenen, anderen Transaktionen wird bei der Ausführung des entsprechenden Datenbankkommandos um eins dekrementiert.

Beim Start einer Transaktion τ laufen in der Datenbank folgende Schritte ab (siehe Abbildung 4.19):

1. Mit TA_start wird die Transaktion zur Menge der im System befindlichen Transaktionen hinzugefügt (setzte τ „scheduled“).

2. Daraufhin werden alle anderen bereits im System befindlichen Transaktionen, die potentiell beim Transaktionsstart mit τ in Konflikt geraten können (Menge $\mathcal{C}_1$), auf eine aktuelle Überschneidung hin überprüft. Dies sind alle Transaktionen, die sich nicht im Friend-Set der Transaktion τ befinden oder die in einem RW-Konflikt mit dieser stehen und sich bereits in der „write“-Phase befinden. Entsprechend dem Verhältnis der Prioritäten und dem augenblicklichen Ausführungszustand der beiden Transaktionen wird der Konfliktzähler einer Transaktion um eins erhöht und diese in die entsprechende Konflikt-Liste der anderen Transaktion eingereiht. Dabei gibt der Ausführungszustand einer Transaktion („executing“, „executing_rw“) an, ob sie augenblicklich auf die Zuteilung von Ressourcen wartet, oder ob die Transaktion im Moment bereits zugeteilte Ressourcen bearbeitet. „Executing“ bezieht sich dabei auf die gesamte Transaktion, „executing_rw“ auf die Bearbeitungsphasen „read“ und „write“. Auf diese Weise ist sichergestellt, daß einer Transaktion, die bereits mit der Bearbeitung ihrer Daten begonnen hat, diese nicht wieder entzogen werden.

3. Zum Abschluß des Transaktionsstarts wird überprüft, ob die neue Transaktion τ alle notwendigen Ressourcen erhalten hat. Ist dies der Fall, so kann die Transaktion mit der

Bearbeitung fortfahren. Falls nicht, wird der einbettende Task in die „blocked“-Queue des Betriebssystems eingereiht. Wird dann später während der Programmausführung der Konfliktzähler der Transaktion τ zu Null, d. h. stehen alle Ressourcen zur Verfügung, wird der einbettende Task wieder in die „ready“-Queue des Betriebssystems umgesetzt.

Alle anderen Datenbankkommandos arbeiten ähnlich. Für einen genauen Überblick sind die von PRED-DF mit `TA_start`, `TA_freeReadSet`, `TA_lockWriteSet` und `TA_commit` durchgeführten Kontrollschritte in Abbildung 4.19 in Flußdiagrammen graphisch dargestellt.

Dieses Vorgehen entspricht der gegenseitigen Blockierung von Transaktionen an gemeinsam genutzten Ressourcen. Dies sind hier die konkurrierend von unterschiedlichen Transaktionen genutzten Elemente oder Gruppen von Elementen in der Datenbank. Die Warteschlangen an den einzelnen Daten sind nach Prioritäten geordnet. Einmal zugeteilte Ressourcen, für die bereits mit der Bearbeitung begonnen wurde, werden nicht wieder entzogen. Abbildung 4.20 erläutert dies noch einmal anhand eines einfachen Beispiels.

Insgesamt ist der zusätzliche Aufwand von PRED-DF gegenüber PRED zur Laufzeit sehr gering, da alle zeitintensiven Operationen vorab durchgeführt werden können. Für Transaktionen des selben Friend-Sets ist der Transaktions-Konfliktgraph nicht zyklisch. Daher erhöht sich die Zahl der zu betrachtenden Transaktionskonflikte und damit die Zahl der Listenoperationen insgesamt beim Übergang von PRED zu PRED-DF nicht. Der zusätzliche Aufwand reduziert sich also im wesentlichen auf die wenigen, zusätzlichen if-Anweisungen, die in Abbildung 4.19 grau hinterlegt sind.

Priority Inheritance Protokoll

Zwei Transaktionen, die konkurrierend sich überschneidende Datensätze bearbeiten, können sich gegenseitig blockieren. Dabei ist es möglich, daß eine Transaktion mit hoher Priorität durch Transaktionen mit niedriger Priorität blockiert wird. Um diese Problematik zu entschärfen, wurde in die hier vorgestellte Implementierung zusätzlich das Priority Inheritance Protokoll integriert (siehe Abschnitt 5.2.1). Dazu werden im Transaktionskontrollblock die ursprüngliche Priorität einer Transaktion (`originalPrio`) und die Priorität nach dem Freigeben des Read-Sets (`afterFreeReadSetPrio`) festgehalten, um im Falle der Prioritätsinversion die Prioritäten der ausführenden Tasks richtig verwalten zu können. Zur korrekten Handhabung transitiver Blockierung wird für jede Transaktion zusätzlich eine Liste aller Transaktionen gespeichert (`blockedBy`), die diese blockieren und ein niedrigere Priorität als sie selbst haben.

Test

Die hier dargestellte Implementierung des Protokolls PRED-DF wurde in dieser Arbeit in der folgenden Systemkonstellation getestet:

- Betriebssystem: RTEMS 3.6.0[6] mit Erweiterungen für EDF-Scheduling [Wro97] und nachrichtenbasiertes EDF-Scheduling [PMK+00],
- Hardware: Galileo Board mit MIPS Orion R4600 Prozessor, Taktfrequenz 50 Mhz.

Die für diese Realisierung gemessenen Ausführungszeiten der einzelnen Abschnitte des Concurrency-Control Protokoll PRED-DF sind in Tabelle 4.3 zusammengefaßt. Die aufgeführten Werte sind jeweils der Mittelwert über 50 Wiederholungen derselben Messung.

[6] Real-Time Executive for Multiprocessor Systems, `http://www.rtems.com`

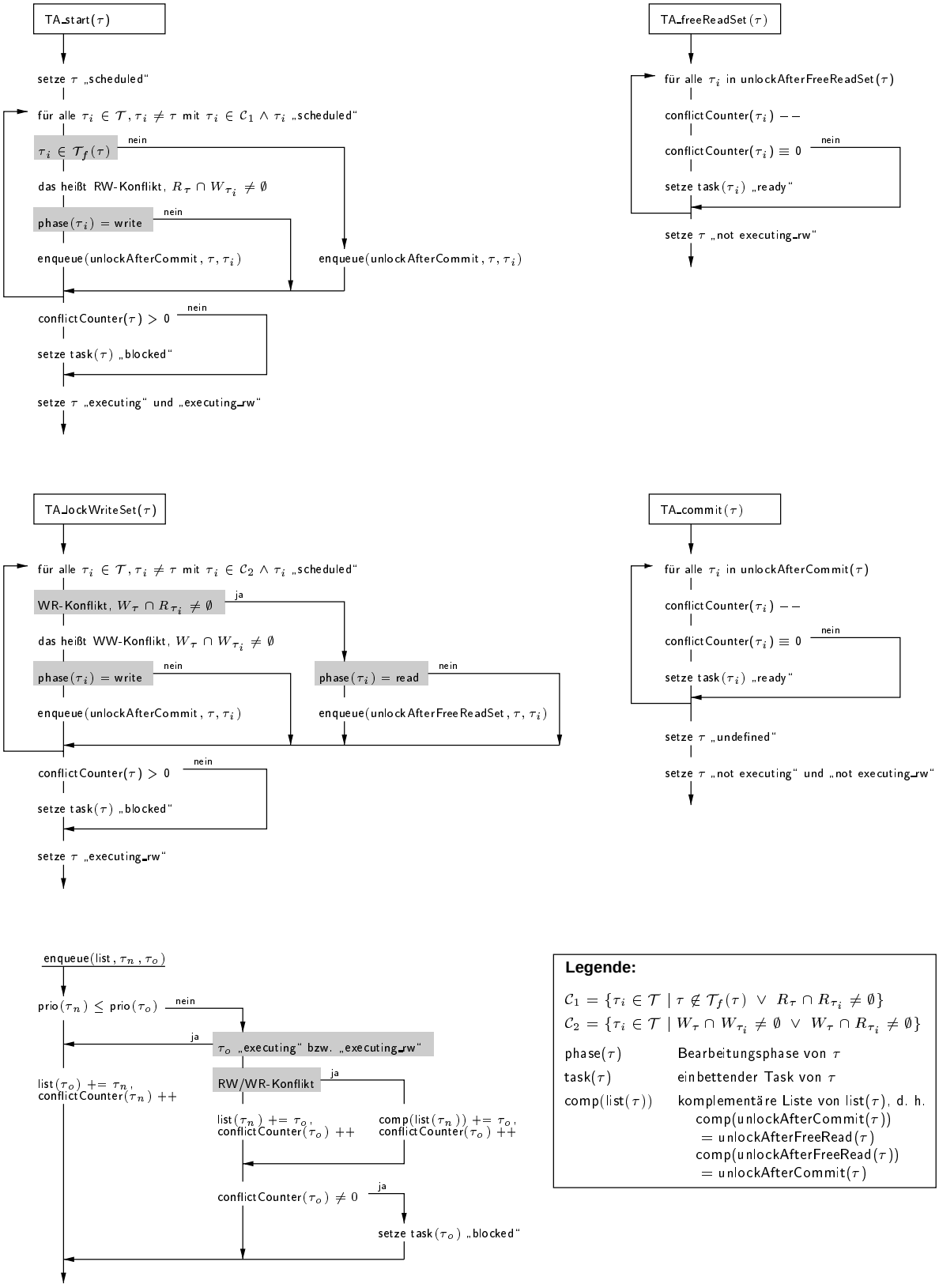


Abbildung 4.19.: Flußdiagramme von PRED-DF für die Datenbankkommandos `TA_start`, `TA_freeReadSet`, `TA_LockWriteSet` und `TA_commit`. Zusätzlich ist das Vorgehen beim Einreihen einer Transaktion in eine entsprechende Konfliktliste mit `enqueue(list,` τ_n`,` τ_o`)` dargestellt. Grau hinterlegt ist der zusätzliche Aufwand von PRED-DF gegenüber PRED zur Laufzeit.

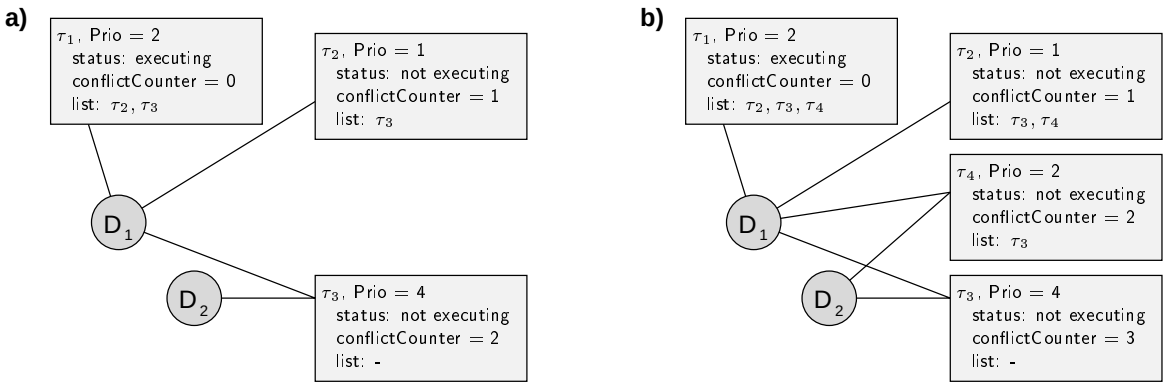


Abbildung 4.20.: Synchronisation verschiedener Transaktionen mit unterschiedlichen Prioritäten an den gemeinsam genutzten Datenelementen D_1 und D_2 in der hier vorgestellten Implementierung von PRED-DF. Abbildung a) stellt die Ausgangssituation dar. Abbildung b) gibt den Systemzustand nach dem Hinzufügen von Transaktion τ_4.

`TA_start`	
kein Konflikt	20 μs
zusätzliche Zeit/Konflikt	4 μs
`TA_freeReadSet`	
kein Konflikt	0.2 μs
zusätzliche Zeit/Konflikt	3 μs
`TA_lockWriteSet`	
kein Konflikt	1 μs
zusätzliche Zeit/Konflikt	4 μs
`TA_commit`	
kein Konflikt	13 μs
zusätzliche Zeit/Konflikt	4 μs

Tabelle 4.3.: Ausführungszeit der wichtigsten Datenbankkommandos. Die Zeiten beziehen sich ausschließlich auf den Aufwand des Concurrency-Control Protokolls. Sie sind jeweils der Mittelwert aus 50 Messungen.

Wird zusätzlich das Priority Inheritance Protokoll verwendet, so erhöht sich die benötigte CPU-Zeit zusätzlich. Pro gegenseitigem Ausschluß ergibt sich durch den zusätzlichen Verwaltungsaufwand bei `TA_start` und `TA_lockWriteSet` eine Erhöhung um 1.2 μs und bei `TA_commit` bzw. `TA_freeReadSet` eine Erhöhung um 0.6 μs.

Eine detaillierte Evaluierung und Bewertung des Protokolls aus der Sicht harter Realzeitsysteme und deren Realzeitnachweis findet sich in Kapitel 5.

Alternativ zu der hier vorgestellten Implementierung wäre es prinzipiell auch denkbar, den gegenseitigen Ausschluß von Transaktionen auch z. B. über Semaphore für gemeinsam genutzte Datenbereiche zu realisieren. Dies hätte dann einen Vorteil, wenn Datenbereiche gemeinsam von vielen Transaktionen genutzt werden („hot spots“). Das Problem einer solchen Implementierung wäre jedoch, daß bei Semaphoren die individuelle Information über den Stand im Lebenszyklus für eine bestimmte Transaktion, die ein Datum gerade bearbeitet, nicht berücksichtigt wird. Genau diese Information wird aber für PRED-DF genutzt. Ein solche Implementierung ist also nicht möglich.

4.4.3. Aktive Datenbankfunktionalität

Ereignisse bzw. Regeln in der aktiven Datenbank sind im Modell, das in dieser Arbeit entwickelt wurde, stets mit einem einzelnen Datenobjekt verknüpft. Gemeinsam mit dem Inhalt des Datums selbst werden aus diesem Grund die Informationen zum aktiven Datenbankverhalten in einer Struktur `dataObject` gespeichert:

```
typdef struct {
      void                   *data;
      transactionListElement *consistencyTransactions;
      ruleListElement        *ruleList;
      kontextListElement     *kontextList;
} dataObject;
```

In der Struktur `dataObject` ist `data` ein Zeiger auf den in diesem Datenelement gespeicherten Datensatz. Der Pointer `consistencyTransactions` zeigt auf eine einfach verkettete Liste von denjenigen Transaktionen $\mathcal{K}_{x_i}$, die nach dem Schreiben dieses Datums x_i zur Sicherung der Konsistenz in der Datenbank ausgeführt werden müssen. Die doppelt verkettete Liste `ruleList` enthält alle dynamischen Regeln für dieses Datenelement, das heißt, all diejenigen Transaktionen bzw. Ereignisse der Applikation, die, falls die jeweils zugeordnete Bedingung wahr ist, nach dem Schreiben des Datums ausgeführt bzw. ausgelöst werden. Der Pointer `kontextList` wird im nachfolgenden Abschnitt erläutert.

Beim „commit“ werden für alle Daten im Schreibset W_i einer Transaktion τ_i alle Elemente der beiden Listen `consistencyTransactions` und `ruleList` bearbeitet. Lösen dabei Regeln, die mit den Daten im Schreibset W_i von τ_i verknüpft sind, kaskadierend wiederum neue Transaktionen aus usf., so werden diese alle sukzessive rekursiv bearbeitet.

Abbruch zyklischer Abhängigkeiten

In solchen Systemen von kaskadierend getriggerten Transaktionen können endlose Zyklen entstehen (Abschnitt 4.3.2). In der durchgeführten Implementierung werden diese erkannt und abgebrochen. Dazu muß während der Bearbeitung einer Regelkaskade bekannt sein, welche Datenobjekte in diesem Zusammenhang schon einmal behandelt wurden. Um dies effizient realisieren zu können, wird jede Abhängigkeitsrechnung in einem definierten Kontext mit einem eindeutig identifizierbaren, einmaligen Schlüssel `id` durchgeführt. Ein Kontext speichert in einer einfach verketteten Liste von `kontextElement`en alle Datenelemente `dataObject`, die innerhalb einer Ausgleichsrechnung von Transaktionen bearbeitet wurden:

```
typdef struct {
      int              id;
      dataObject       *dataObject;
      kontextElement   *nextKontextElement;
} kontextElement;
```

Beim Abschluß einer Transaktion durch den Benutzer wird ein neuer Kontext erzeugt und der Bezug darauf innerhalb einer Abarbeitungskaskade stufenweise nach unten weitergegeben. Werden die im vorherigen Abschnitt dargestellten Optimierungsmöglichkeiten genutzt und in der Datenbank zusätzliche Tasks eingeführt, so muß der jeweils gültige Kontext gemeinsam mit dem Ereignis dem zusätzlichen Task weitergegeben werden. Während der Bearbeitung der Datenbankregeln fügt jede Transaktion dem aktuellen Kontext die von ihr bearbeiteten, d. h.

geschriebenen Datenelemente hinzu. Parallel dazu wird in jedem Datum `dataObject`, das innerhalb einer Ausgleichsrechnung bearbeitet wird, die `id` des Kontextes der aktuellen Bearbeitung in der Liste `kontextList` gespeichert. Wird im Rahmen der Regelbearbeitung versucht, dasselbe Datum ein zweites Mal im selben Kontext zu schreiben, ist der zu diesem Zeitpunkt gültige Kontext bereits in der `kontextList` dieses Datums gespeichert. In diesem Fall wird dieser Zweig der rekursiven Bearbeitung der Regelkaskade abgebrochen.

Ist letztlich die Regelbearbeitung zu einer Benutzer-Transaktion komplett beendet, wird der dazugehörige Kontext aus dem System gelöscht. Das heißt, in jedem Datenelement `dataObject`, das in der Liste von `kontextElement`en eines Kontextes enthalten ist, wird der Verweis auf die `id` dieses Kontextes aus der Liste `kontextList` entfernt. Zum Schluß wird die Kontext-Liste selbst aufgelöst. In Abbildung 4.21 ist das prinzipielle Vorgehen anhand eines beispielhaften Zyklus der Länge drei dargestellt.

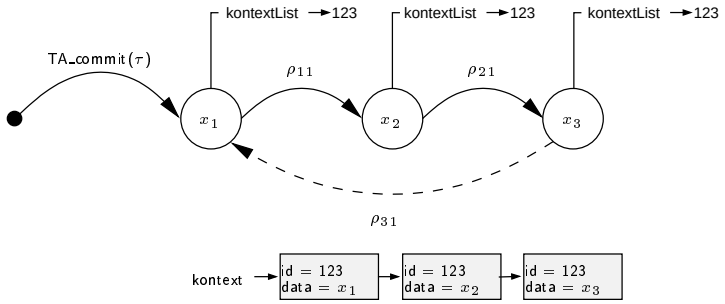


Abbildung 4.21.: Beispiel zum Abbruch unendlicher Zyklen bei der Bearbeitung von Regelkaskaden in der Datenbank. Im dargestellten Kontext `123` wir die zur Regel ρ_{31} gehörende Transaktion nicht mehr ausgeführt.

In der hier durchgeführten Beispielimplementierung verlängert sich durch die Verwendung der aktiven Datenbankfunktionalität die für `TA_commit` benötigte CPU-Zeit um 6 μs, zuzüglich der Zeit, die für die Auswertung der Bedingung und die Ausführung der entsprechenden Aktionen dieser Regel benötigt wird.

Der große Vorteil, die Kontext-Id mit den einzelnen Datenobjekten zu verknüpfen und zusätzlich die behandelten Daten im Kontext zu speichern, ist, daß zur Überprüfung auf Zyklen nur jeweils die im Verhältnis kurzen Kontextlisten der einzelnen Datenobjekte durchsucht werden müssen. Es ist nicht notwendig, jedes mal neu eine Liste aller bisher behandelten Daten zu durchsuchen. Durch das Führen der Datenliste im Kontext, können die Kontextkennungen am Ende der Rechnung sehr einfach wieder aus dem System entfernt werden. Der geringfügige Nachteil dieses Verfahrens ist, daß mehrere unterschiedliche – im Prinzip redundante – Listen geführt werden müssen.

Dynamische Regeln

Für ein einzelnes Datenelement `dataObject` ist der Inhalt der Liste `consistencyTransactions` konstant. Dieser ist durch die Konfiguration bereits beim Systemstart definiert. Die Elemente `ruleListElement`

```
typdef struct {
      void                (*condition)();
      void                (*transaction)();
      semaID              *semaId;
      ruleListElement     *nextRuleListElement;
      ruleListElement     *previuosRuleListElement;
} ruleListElement;
```

der doppelt verketteten Liste `ruleList` sind dagegen variabel und können zur Laufzeit der Applikation durch den Benutzer dynamisch geändert werden. Die Funktion `(*condition)()` enthält hier die für diese Regel zu evaluierende Bedingung. Ist diese wahr, so wird entweder die Transaktion `(*transaction)()` ausgeführt, oder das zugeordnete Ereignis ausgelöst, das heißt die Semaphore mit der Kennung `semaID` gesetzt.

Tabelle 4.4 gibt abschließend noch einmal einen Überblick über die wichtigsten Datenbankbefehle, die dem Anwender in der vorliegenden Beispielimplementierung zur Verfügung stehen.

Definition des Datenbankinhaltes
`dataObject* RTDB_createNewDataObject()`
`int RTDB_addRuleConsistency(dataObject* data, void (*transaction)())`
Datenbankkonfiguration und Initialisierung
`void RTDB_config(void)`
`void RTDB_init(void)`
Transaktionsausführung
`TA_CTRLBLK* TA_start(int id, rtems_id taskId, kontextElement** pointerToKontext)`
`int TA_freeReadSet(TA_CTRLBLK* transaction)`
`int TA_lockWriteSet(TA_CTRLBLK* transaction)`
`int TA_commit(TA_CTRLBLK* transaction, kontextElement** pointerToKontext)`
Verwaltung dynamischer Regeln
`int RTDB_addRule(dataObject* data, void (*condition)(), void (*transaction)(), semaId *semaId)`
`int RTDB_removeRule(dataObject* data, void (*transaction)(), semaId *semaId)`

Tabelle 4.4.: Überblick über alle Datenbankbefehle, die dem Anwender in der vorliegenden prototypischen Implementierung zur Verfügung stehen.

Die Worst-Case Execution Time der Datenbankommandos in dieser Implementierung kann bestimmt werden, falls die WCETs der einzelnen Teilschritte bekannt ist. Sie ist dann nur noch von der Länge der zu bearbeitenden Listen abhängig. Im Fall von `TA_start()` steigt die Ausführungszeit zum Beispiel linear mit der Zahl der Transaktionen, die sich gerade im System befinden und mit denen sich die Transaktion im Konflikt befindet.

5. Nachweis der Realzeitfähigkeit

Dieses Kapitel befaßt sich mit der Analyse des Realzeitverhaltens einer Datenbank, die das in dieser Arbeit definierte Modell einer aktiven Realzeitdatenbank zu Grunde legt. Dabei wurde Wert darauf gelegt, die Integration des Concurrency-Control Protokolls und der aktiven Funktionalität in das Analysemodell der Applikation allgemein zu halten, um die Übertragbarkeit auch auf andere Verfahren zum Realzeitnachweis sicherzustellen.

Abschnitt 5.1 gibt zuerst einen knappen Überblick über das hier verwendete Verfahren zur Realzeitanalyse. In Abschnitt 5.2.1 wird mit Hilfe der vorgestellten Techniken das Concurrency-Control Protokoll PRED-DF analysiert. Abschnitt 5.2.2 befaßt sich mit der Integration der aktiven Funktionalität in das Analysemodell der Applikation und analysiert deren Auswirkungen auf das Realzeitverhalten der Datenbank.

5.1. Systembeschreibung

5.1.1. Umwelt

Die in dieser Arbeit verwendete Umweltspezifikation basiert auf der Beschreibungstechnik der *Ereignisströme* und der *Ereignisabhängigkeitsmatrizen*. Diese Methode wurde 1993 von Gresser entwickelt [Gre93a][Gre93b] und stellt eine wichtige Erweiterung zur häufig verwendeten Spezifikation von Umweltereignissen durch den minimal auftretenden Abstand zwischen zwei Ereignissen dar. Zudem können zeitliche Korrelationen zwischen Ereignissen aus verschiedenen Quellen einfach und genau spezifiziert werden.

Ereignisströme

Durch einen Ereignisstrom werden die kürzesten abgeschlossenen Zeitintervalle $a_0, a_1, a_2, \ldots$ angegeben, in denen 1, 2, 3, ... Ereignisse eines bestimmten Typs l auftreten können[1]. Bildlich gesprochen dürfen, falls man beispielsweise ein Zeitfenster der Länge a_1 über einer Aufzeichnung einer beliebigen möglichen Ereignisfolge über der Zeit verschiebt, stets höchstens zwei Ereignisse innerhalb des Zeitfensters a_1 liegen. Entsprechend dürfen innerhalb eines Zeitfensters der Länge a_2 immer höchstens drei Ereignisse liegen usf. Das minimale Zeitfenster, in dem ein Ereignis auftreten kann, ist stets Null. Die Zusammenfassung aller Intervallangaben bezeichnet man als *Ereignisstrom*. Konstruiert man eine Ereignisfolge, die jeweils die kürzesten, nach der Ereignisstromspezifikation möglichen Abstände realisiert, so heißt diese Folge *Worst-Case Ereignisfolge*.

Um eine endliche Darstellung auch von unendlichen, periodischen Ereignisfolgen zu ermöglichen, kann für solche Ereignisfolgen zusätzlich die Zykluszeit z spezifiziert werden. Diese gibt an, mit welchem minimalen Zeitabstand sich Ereignisfolgen innerhalb eines Ereignisstroms wiederholen

[1] Zeiten werden in dieser Arbeit stets dimensionslos angegeben. Sie sind das Vielfache einer beliebigen Zeiteinheit.

können. Tritt ein Ereignis nur einmal auf, so ist $z = \infty$. Hierbei bezeichnet „∞“ nicht Unendlich im eigentlichen mathematischen Sinne, sondern einen Zeitraum, der wesentlich länger als der gesamte, denkbare Beobachtungszeitraum für die betrachtete Applikation ist.

Gresser gibt in seiner Arbeit eine Notation zur Beschreibung von Ereignisströmen an. Der besseren Übersichtlichkeit wegen wird hier allerdings eine alternative Notation verwendet, die sich an [Blo99] orientiert. Ein Ereignisstrom für einen Ereignistyp l ist die Menge ES^l von λ Tupeln T_i^l, $i = 0 \ldots (\lambda - 1) < \infty$. Jedes Tupel T_i^l besteht aus μ_i Intervallzeiten und einer Zykluszeit z_i:

$$
\begin{aligned}
\mathrm{ES}^l &= \{\mathrm{T}_0^l, \mathrm{T}_2^l \ldots, \mathrm{T}_{\lambda-1}^l\} \\
&= \{(a_{0,0}^l, \ldots, a_{0,\mu_1-1}^l, z_0^l), \ldots, (a_{\lambda-1,0}^l, \ldots, a_{\lambda-1,\mu_{\lambda-1}}^l, z_{\lambda-1}^l)\} \qquad (5.1) \\
&\text{mit} \\
z_i^l &\neq z_j^l \quad \forall\, i, j \in \{0 \ldots (\lambda - 1)\}, i \neq j \quad \text{und} \quad z_i^l \neq 0 \\
a_{i,j}^l &\leq a_{i,j+1}^l \quad \text{und} \quad a_{i,j}^l \in \mathbb{N}_0
\end{aligned}
$$

In dieser Notation bezeichnet $a_{i,j}^l$ den minimalen zeitlichen Abstand von $(j+1)$ Ereignissen des Typs l, der durch das Tupel T_i^l definiert wird. z_i^l gibt die Zykluszeit des i-ten Tupels an.

Besteht ein Ereignisstrom nur aus einem einzigen Tupel, $\mathrm{ES}^l = (a_0, a_1, \ldots, a_{\mu-1}, z)$, so wird er als *homogen* bezeichnet, enthält er mehrere Ereignistupel, so heißt er *inhomogen*. Inhomogene Ereignisströme mit endlichen Zykluszeiten können durch Ereignisvervielfachung in homogene Ereignisströme mit einer neuen Zykluszeit transformiert werden. Die neue Zykluszeit ist dabei das kleinste gemeinsame Vielfache der alten Zykluszeiten:

$$\{(0, \infty), (0, 2), (0, 3)\} = \{(0, \infty), (0, 0, 2, 3, 4, 6)\}.$$

Das minimale Intervall, in dem *ein* Ereignis auftreten kann, ist stets Null, d. h. in einem homogenen Ereignisstrom gilt stets $a_{0,0} = 0$. In inhomogenen Ereignisströmen muß es mindestens ein Element $a_{i,0} = 0$ geben. In allen anderen Tupeln darf auch $a_{i,0} \geq 0$ gelten. Dies wird als *Offset* bezeichnet. Würde man einen homogenen Ereignisstrom auf nur ein Tupel $(0, z)$ reduzieren, so entspräche dies wieder der Angabe des minimalen Abstands zwischen zwei Ereignissen.

Ereignisfunktion Aus einem gegebenen Ereignisstrom ES^l kann die Ereignisfunktion $\mathrm{E}_a^l(t)$ berechnet werden. Sie gibt die maximal mögliche Anzahl an Ereignissen des Typs l innerhalb des abgeschlossenen Zeitintervalls t an und wird folgendermaßen berechnet:

$$
\mathrm{E}_a^l(t) = \sum_{i=0}^{\lambda-1} \sum_{j=0}^{\mu_i-1} \begin{cases} 0 : t < a_{i,j}^l \\ \left\lfloor \frac{t - a_{i,j}^l}{z_i} + 1 \right\rfloor : t \geq a_{i,j}^l \wedge z_i < \infty \\ 1 : t \geq a_{i,j}^l \wedge z_i = \infty \end{cases} \qquad (5.2)
$$

Für ein rechtsseitig offenes Zeitintervall t stellt sich die Gleichung für die Ereignisfunktion $\mathrm{E}_r^l(t)$ wie folgt dar:

$$\mathrm{E}_r^l(t) = \sum_{i=0}^{\lambda-1} \sum_{j=0}^{\mu_i-1} \begin{cases} 0 : t \leq a_{i,j}^l \\ \left\lfloor \frac{t-a_{i,j}^l}{z_i} + 1 \right\rfloor : t > a_{i,j} \wedge z_i < \infty \\ 1 : t > a_{i,j} \wedge z_i = \infty \end{cases} \tag{5.3}$$

Die bisherigen Definitionen werden im folgenden anhand eines kurzen Beispiels veranschaulicht. Gegeben sei dafür der folgende homogene Ereignisstrom:

$$\mathrm{ES}^{ex} = \{(0, 1, 3, 7)\}$$

Abbildung 5.1 a) zeigt eine zu diesem Ereignisstrom gehörige Worst-Case Ereignisfolge. Abbildung 5.1 b) gibt die zugeordnete Ereignisfunktion $\mathrm{E}_a^{ex}(t)$.

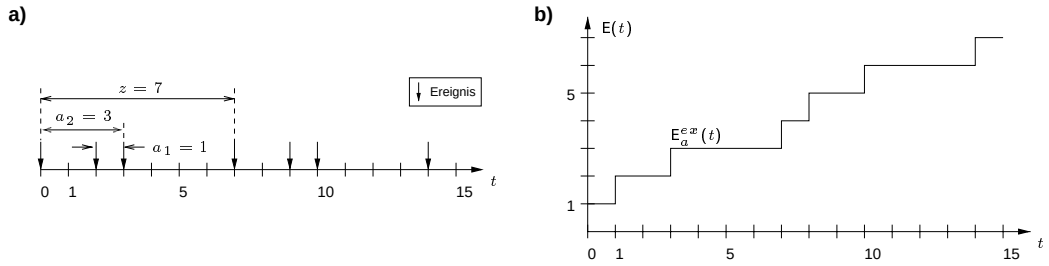


Abbildung 5.1.: Beispiel für die Umweltmodellierung mit Hilfe eines Ereignisstroms. a) zeigt die Ereignisse der Worst-Case Ereignisfolge für den Ereignisstrom ES^{ex}. b) gibt die zugeordnete Ereignisfunktion $\mathrm{E}_a^{ex}(t)$.

Gültigkeit von Ereignisströmen Auf Grund der Definition der Ereignisströme kann der Wert der Ereignisfunktion in einem beliebigen Zeitintervall nicht stärker zunehmen als der Wert der Ereignisfunktion für dieses Intervall. Das heißt, es muß stets folgende Ungleichung erfüllt sein:

$$\mathrm{E}_a(t_2) - \mathrm{E}_a(t_1) \leq \mathrm{E}_a(t_2 - t_1) \quad \text{mit} \quad t_2 > t_1 \tag{5.4}$$

Dies hat seinen Grund darin, daß die einzelnen Tupelelemente in aufsteigender Reihenfolge jeweils den kürzesten Abstand angeben, in dem eine bestimmte Anzahl an Ereignissen auftreten kann („zeitlich nach vorne konzentriert“).

Für homogene Ereignisströme kann diese Bedingung mit dem folgenden Gleichungssystem enger gefaßt werden:

$$\begin{aligned} a_0 &= 0 \\ a_1 &= \min(a_1 - a_0, a_2 - a_1, \ldots, z + a_0 - a_{\mu-1}) \\ a_2 &= \min(a_2 - a_0, a_3 - a_1, \ldots, z + a_1 - a_{\mu-1}) \\ &\vdots \\ a_{\mu-1} &= \min(a_{\mu-1} - a_0, z + a_0 - a_1, \ldots, z + a_{\mu-2} - a_{\mu-1}) \end{aligned}$$

Konkatenation Verschmilzt man zwei gültige Ereignisströme aus unterschiedlichen Quellen zu einem Ereignisstrom, so entsteht wiederum ein gültiger Ereignisstrom. Die Konkatenation ist wie folgt definiert:

$$\mathrm{ES}_{\mathrm{konk}} = \mathrm{ES}_1 \circ \mathrm{ES}_2 := \mathrm{ES}_1 \cup \mathrm{ES}_2 \tag{5.5}$$

Ereignisabhängigkeitsmatrix (EDM)

Ereignisströme beschreiben das zeitliche Verhalten von Ereignissen eines bestimmten Typs. Falls zusätzliche Informationen über die zeitliche Korrelation von Ereignissen unterschiedlichen Typs vorliegen (z. B. ein minimaler zeitlicher Abstand), so können diese Informationen in einer Ereignisabhängigkeitsmatrix EDM niedergelegt werden:

$$\mathrm{EDM} = \begin{pmatrix} e_{11} & e_{12} & \cdots & e_{1n} \\ \vdots & \ldots & \ddots & \vdots \\ e_{n1} & e_{n2} & \cdots & e_{nn} \end{pmatrix} \tag{5.6}$$

Das Element e_{ik} beschreibt hierbei den minimalen zeitlichen Abstand zwischen zwei Ereignissen der Ereignisströme ES^i und ES^k. Die genauen Beziehungen sind in Abbildung 5.2 noch einmal veranschaulicht.

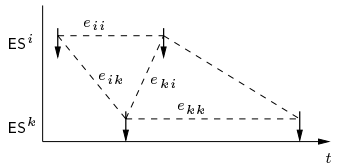


Abbildung 5.2.: Spezifikation der zeitlichen Beziehung zwischen unterschiedlichen Ereignisströmen mit Hilfe der Ereignisabhängigkeitsmatrix EDM.

Die Diagonalelemente von EDM sind die bereits bekannten Elemente der Ereignisströme, die den minimalen Abstand zweier Ereignisse beschreiben. Falls keine zusätzliche Information über eine zeitliche Korrelation der Ereignisströme vorliegt, so ist $e_{ik} := 0$ zu definieren. Dies entspricht unabhängigen Ereignisströmen.

Sowohl die Ereignisströme als auch die Ereignisabhängigkeitsmatrizen sind das Ergebnis einer Analyse des physikalischen Verhaltens des einbettenden technischen Prozesses und müssen dem Systementwickler bekannt sein und werden in dieser Arbeit als gegeben vorausgesetzt.

5.1.2. Applikation

Das Modell der gesamten Applikation setzt sich aus den Modellen für die Datenbank und für die einbettende Anwendung zusammen. Das Modell der Datenbank wurde bereits in Kapitel 4 ausführlich diskutiert. Der folgende Abschnitt konzentriert sich daher auf die Erläuterung der Mittel, die für die Abbildung der einbettenden Applikation verwendet werden.

Die Applikation wird durch einen annotierten gerichteten Graphen spezifiziert. Ein kleiner Ausschnitt aus solch einer Spezifikation ist in Abbildung 5.3 dargestellt. Anhand dieser Abbildung werden im folgenden dessen wesentlichen Elemente erläutert.

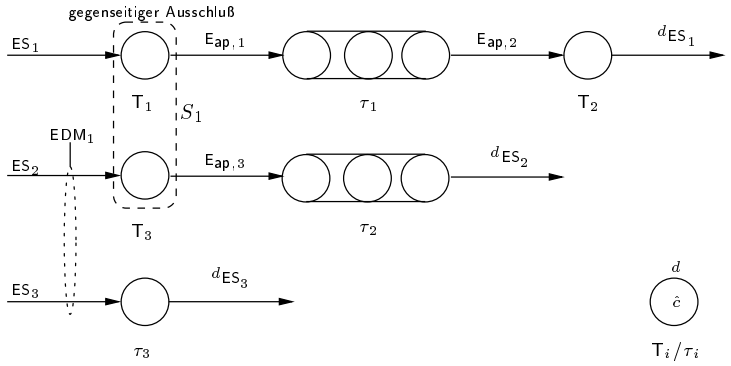


Abbildung 5.3.: Graphbasiertes Modell der Applikationsstruktur und Einbettung der Datenbank bzw. der Transaktionen in das Analysemodell.

1. *Taskknoten.* Die Taskknoten T_i sind die Bearbeitungseinheiten einer Applikation. Sie sind Rechenabschnitte ohne Betriebssystemaufrufe. Solche Aufrufe sind nur am Anfang und am Ende eines Tasks zulässig, d. h. auch alle Ressourcen eines Taskknotens sind gegebenenfalls am Anfang zu belegen und am Ende wieder freizugeben. Taskknoten werden beim Programmstart erzeugt und existieren über die gesamte Programmlaufzeit. Dynamische Generierung ist nicht möglich. Taskknoten können identisch mit Betriebssystem-Tasks oder ein Teil davon sein. Die maximale und minimale Rechenzeit eines Taskknotens T_i sind bekannt $(\hat{c}_i, \check{c}_i)$. Jeder Task hat eine individuelle Priorität, bzw. Deadline $d_{T,i}$.

   Die Bearbeitung einer Task wird entweder direkt durch einen Ereignisstrom oder indirekt durch den in einem Präzedenzsystem vorhergehenden Task ausgelöst. Das auslösende interne Ereignis innerhalb einer Applikation wird in diesem Fall als $E_{ap,i}$ bezeichnet.

   Gegenseitiger Ausschluß von zwei oder mehr Taskknoten kann, wie in Abbildung 5.3 dargestellt, ebenfalls spezifiziert werden (S_1).

2. *Transaktionen.* Jeder Betriebssystem-Task kann eine oder mehrere Transaktionen enthalten. Für das Analysemodell der Applikation werden der umgebende Task und die eingebetteten Transaktionen auf einzelne separate Taskknoten abgebildet. Normalisierte Transaktionen bestehen dann aus drei Taskknoten; nicht normalisierte Transaktionen bestehen aus einem Taskknoten. Die zusammenhängenden Taskknoten einer normalisierten Transaktion sind besonders gekennzeichnet (siehe Abbildung 5.3).

   Die maximalen und minimalen Rechenzeiten, die spezifische Deadline $d_{\tau,i}$ und das Lese- bzw. Schreibset R_i bzw. W_i sind für jede Transaktion bekannt.

3. *Scheduling.* Für die Zuteilung von Rechenzeit an die einzelnen Taskknoten wird EDF-Scheduling angenommen.

4. *Präzedenzsysteme.* Präzedenzbeziehungen definieren die Abarbeitungsreihenfolge der unterschiedlichen Taskknoten, die durch den Kontrollfluß bzw. durch die Zugehörigkeit zu einer einzigen Transaktion vorgegeben wird. Wird daher der Task T_1 immer vor dem Task T_2 bearbeitet $(T_1 \succ T_2)$, so wird das durch eine gerichtete Kante (T_1, T_2) symbolisiert. Mehrere Taskknoten, die durch einen Kantenzug verbunden sind und daher nacheinander bearbeitet werden, bilden ein Präzedenzsystem. Normalerweise wird ein Präzedenzsystem durch ein externes Ereignis angestoßen und bearbeitet dann dieses Ereignis bis zur Rückgabe der Antwort bzw. Reaktion an die Umwelt. Zwischen den einzelnen Knoten eines Präzedenzsystems werden Warteschlangen angenommen (asynchrone Kommunikation).

Anhand seines Präzedenzsystems kann die Startzeit $\hat{s}_{b,i}$ eines Knotens T_i berechnet werden. Dies ist die Summe aller maximalen Rechenzeiten $\hat{c}_i$ der Taskknoten, die im Präzedenzsystem des betrachteten Tasks liegen und deren Deadlines kleiner sind, als die Deadline des betrachteten Tasks:

$$\hat{s}_{b,i} = \sum_{j,\ d_j < d_i} \hat{c}_j \tag{5.7}$$

Die Summe aller Rechenzeiten, die in einem Präzedenzsystem nach T_i liegen, wird mit $\hat{s}_{a,i}$ bezeichnet. Sie berechnet sich analog zu obiger Formel. Ebenfalls analog sind $\check{s}_{b,i}$ und $\check{s}_{a,i}$ jeweils für die minimalen Ausführungszeiten definiert.

5. *Ereignis-Abhängigkeiten.* Abhängigkeiten zwischen unterschiedlichen Ereignisströmen werden durch die Angabe der entsprechenden Ereignisabhängigkeitsmatrix definiert.

6. *Deadlines.* Deadlines werden immer im Bezug auf ein Ereignis bzw. einen Ereignisstrom definiert. Das heißt, man spezifiziert den maximalen Zeitraum, der nach dem Eintreten eines Ereignisses bis zur Fertigstellung eines Taskknotens verstreichen darf.

   In der Regel haben alle Taskknoten im selben Präzedenzsystem dieselbe Deadline [PMK+00]. Das entspricht der Angabe der maximal zulässigen Reaktionszeit d_{ES_i} auf ein Ereignis in ES_i. Wird das Ergebnis eines Tasks an einer anderen Stelle der Applikation früher benötigt, so kann die Deadline entsprechend kleiner sein. Die Deadline eines Task ist jedoch nie kleiner als die seines Vorgängers.

Die Informationen über die Struktur der Anwendung, die notwendig sind, den Graphen aufzubauen, können entweder aus dem Programmcode direkt extrahiert werden, oder sie sind Produkt des Spezifikationsprozesses im Rahmen der Softwareentwicklung.

5.2. Realzeitnachweis

Mittels der in Abschnitt 5.1.1 definierten Ereignisfunktionen E_a^l und E_a^l können zusammen mit den Angaben aus dem Analysemodell der Applikation die Rechenzeitanforderungsfunktionen $C_a(I)$ und $C_r(I)$ definiert werden, n ist die Zahl der unterschiedlichen Ereignisströme:

$$C_a(I) = \sum_{l=1}^{n} E_a^l(I - d_l)\ \hat{c}_l \tag{5.8}$$

$$C_r(I) = \sum_{i=l}^{n} E_r^l(I - d_l)\ \hat{c}_l \tag{5.9}$$

$C_a(I)$ gibt die maximal mögliche Summe aller Rechenzeiten an, die innerhalb eines beliebigen, abgeschlossenen Zeitintervalls I angefordert werden können und auch beendet sein müssen. $C_r(I)$ ist äquivalent für ein rechtsseitig offenes Intervall definiert.

Für unabhängige Ereignisströme und unabhängige Tasks (d. h. keine Präzedenzen, kein gegenseitiger Ausschluß etc.) läßt sich zeigen, daß eine Applikation alle ihre Deadline einhält, falls für ein beliebiges Intervall I stets gilt:

$$C_a(I) \leq I \tag{5.10}$$

Anschaulich gesprochen heißt das, daß $C_a(I)$ stets unterhalb der Winkelhalbierenden liegen muß. Abbildung 5.4 zeigt das Vorgehen anhand eines sehr einfachen Beispiels.

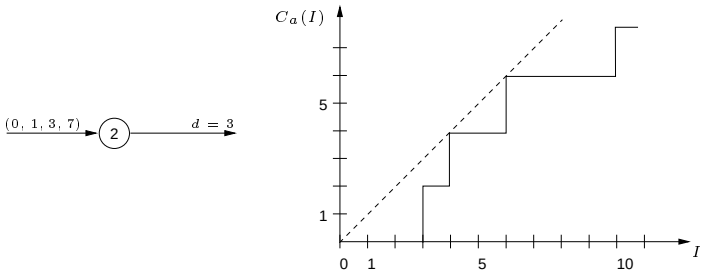


Abbildung 5.4.: Prinzipielles Vorgehen beim Nachweis der Realzeitfähigkeit mittels der Rechenzeitanforderungsfunktion $C_a(I)$.

Soweit die Analyse von Systemen, in denen sich Tasks auf Grund von wechselseitigem Ausschluß gegenseitig blockieren können, benötigt wird, wird diese später vorgestellt.

Präzedenzbeziehungen

Sind unterschiedliche Tasks über Präzedenzrelationen miteinander verknüpft, sind sie nicht mehr unabhängig voneinander. Allerdings muß die Deadline eines im Präzedenzsystem nachfolgenden Tasks stets größer oder gleich der Deadline aller seiner Vorgänger sein. Durch die Zuteilung der Tasks nach ansteigender Deadline wird die Einhaltung der Präzedenzrelationen sichergestellt. Auf Grund dessen und auf Grund der Entkopplung der Tasks durch asynchrone Taskkommunikation können sie für die Realzeitanalyse als unabhängige Tasks angenommen werden, die alle durch den selben Ereignissstrom angeregt werden [Gre93b].

Abhängige Anforderungen

Die exakte Analyse abhängiger Rechenzeitanforderungen, die über eine Ereignisabhängigkeitsmatrix untereinander verknüpft sind, ist eine NP-vollständige Problemstellung. Zur näherungsweisen Lösung sind in [Gre93b] iterative Verfahren und Ansätze für Näherungslösungen angegeben. Insbesondere bei der Analyse von Situationen mit gegenseitigem Ausschluß von Tasks (Prioritätsinversion) kann allerdings durch die Einbeziehung von Informationen über Ereignisabhängigkeiten die zusätzlich zu berücksichtigende Rechenzeit geringer angesetzt werden (siehe Abschnitt 5.2.1).

5.2.1. Betrachtung von PRED-DF

Abbildung auf das Analysemodell des Realzeitnachweises

Das in dieser Arbeit entwickelte Concurrency-Control Protokoll PRED-DF kann sehr einfach in alle Verfahren zum Realzeitnachweis integriert werden, die es erlauben, den gegenseitigen Ausschluß von Tasks zu behandeln (siehe Abschnitt 2.1.2). Aus der Perspektive der Verifikationsverfahren sind die Lese- und Schreibdatensätze der Transaktionen Ressourcen, die entsprechend der Kompatibilitätsmatrix K_{ij} (Abbildung 4.3) nur jeweils von einer Transaktion genutzt werden dürfen. Daher können entweder ganze Transaktionen oder, falls eine Transaktion im Friend-Set der anderen liegt, Teile davon nicht parallel abgearbeitet werden. Sie sind daher im

Realzeitnachweis wie Tasks zu behandeln, die sich auf Grund konkurrierender Ressourcennutzung gegenseitig ausschließen.

Die Abbildung von PRED-DF auf das hier vorgestellte Applikationsmodell zur Realzeitanalyse zeigt Abbildung 5.5 anhand eines einfachen Beispiels.

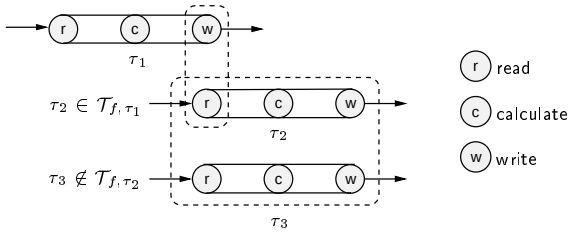


Abbildung 5.5.: Abbildung von PRED-DF auf das Analysemodell des Realzeitnachweises. Transaktionen mit sich überschneidenden Datensätzen schließen sich entweder völlig oder nur während des Lese- oder Schreibvorgangs gegenseitig aus.

Evaluierung und Einordnung

Es existiert eine Vielzahl an vergleichenden Untersuchungen zu unterschiedlichen CC-Verfahren. Hier wird in der Regel jedoch nur die Performance in Form der *Success Ratio S* bestimmt, d. h. der über die Zeit gemittelte Anteil an Transaktionen im Gesamtsystem, der unter den gegebenen Randbedingungen seine Deadlines einhält [LL93][LS96][AGM92].

Solche Untersuchungen sind allerdings vor allem für weiche Echtzeitsysteme von Bedeutung. Als alleinige Metrik zur Bewertung von CC-Protokollen für harte Echtzeitsysteme sind sie nicht ausreichend, da sie lediglich eine Aussage über die mittlere Erfolgsquote einer Transaktion zulassen. Sie sagen nichts über das Verhalten der Protokolle in harten Realzeitsystemen, d. h. die Vorhersagbarkeit der Transaktionsdurchführung, die Wechselwirkung der Datenbank mit dem Scheduling des Realzeit-Betriebssystems und den effizienten Umgang mit der Ressource CPU aus. Zur Evaluierung des Concurrency-Control Protokolls PRED-DF erfolgt deshalb hier, neben der Untersuchung der Systemperformance, die Einordnung anhand der in Kapitel 3 herausgearbeiteten Forderungen *Richtigkeit, Rechtzeitigkeit, Optimalität* und *Effizienz*. Ein weiteres wichtiges Bewertungskriterium für den Einsatz in harten Realzeitsystemen ist, welche Genauigkeit das jeweilige CC-Protokoll beim Nachweis der Realzeitfähigkeit zuläßt, d. h. insbesondere, wie pessimistisch die Annahmen sind, die in der Realzeitanalyse angewendet werden müssen (*Overestimation*).

Richtigkeit und Rechtzeitigkeit Das hier entwickelte Datenbankprotokoll PRED-DF ermöglicht die Zusammenfassung von Datenbankoperationen in Transaktionen und garantiert bei deren paralleler Ausführung auf der Datenbank *Richtigkeit* in Form von *Konflikt-Serialisierbarkeit*. Gleichzeitig ist das Laufzeitverhalten von PRED-DF vorhersagbar und einfach in sehr viele Algorithmen zur Verifikation von Realzeitanforderungen integrierbar. Transaktionen sind priorisierbar und werden entsprechend ihrer Priorität bearbeitet. Die Garantie von *Rechtzeitigkeit* ist somit möglich.

Optimalität Der Begriff der *Optimalität* charakterisiert, inwieweit die Schedulingentscheidungen des Realzeit-Betriebssystems durch die Blockierung von Transaktionen durch die Datenbank zur Sicherung der Datenkonsistenz beeinflußt werden. Die Länge dieser Blockierung hat direkten

Einfluß auf die Qualität des Realzeitnachweises (siehe unten). Ein Maß für die Optimalität ist die Blockierzeit $\overline{t_b}$, die Transaktionen im Mittel durch alle anderen Transaktionen im System auf Grund des CC-Verfahrens der Datenbank erfahren. In dieser Arbeit wird für die Optimalität folgender Zusammenhang definiert ($\overline{l}$ bezeichnet die mittlere Laxity einer Transaktion):

$$\text{Optimalität } O := \left(1 - \frac{\overline{t_b}}{\overline{l}}\right) \tag{5.11}$$

Das heißt, die Optimalität ist eins ($O = 1$) bei keiner Blockierung und Null ($O = 0$), falls die Blockierung im Mittel den gesamten zeitlichen Spielraum ausschöpft, der zur Bearbeitung einer Transaktion zur Verfügung steht. Steigt die mittlere Blockierungszeit darüber hinaus weiter an, wird O negativ. Nimmt man im Durchschnitt eine über die Zeit gleichmäßige Belastung der Datenbank mit Transaktionen an, die einen mittleren zeitlichen Abstand von Δt haben und die im Durchschnitt ihren Datensatz für die Zeit $\overline{t_l}$ sperren, so läßt sich vereinfacht folgender funktionaler Zusammenhang für die Optimalität ansetzen:

$$O = \left(1 - P_{\mathrm{k}} \frac{\overline{t_l}}{2} \frac{1}{\overline{l}}\right) = \left(1 - C \frac{\overline{t_l}^2}{2\Delta t} \frac{1}{\overline{l}}\right) \tag{5.12}$$

P_{k} bezeichnet hierbei die Wahrscheinlichkeit dafür, daß eine Transaktion durch eine andere Transaktion, mit der sie sich im Konflikt befindet, blockiert wird. Diese Wahrscheinlichkeit ist proportional zum Verhältnis $\overline{t_l}/\Delta t$.

Die Optimalität steigt also quadratisch mit der Abnahme der Locking-Zeit $\overline{t_l}$ der Transaktionen. Eine Verkürzung der Locking-Zeiten beeinflußt die Optimalität daher überproportional positiv.

Dieses Verhalten konnte auch in der Simulation sehr gut verifiziert werden. Das hierfür in dieser Arbeit entwickelte Simulationsprogramm besteht aus einem Workload-Generator, einem Scheduler und einem Logging Mechanismus und gestattet es, das Verhalten eines Concurrency-Control Protokolls unter unterschiedlichen Bedingungen zu testen [Mün00b]. Der Workload-Generator erzeugt hierfür über Poisson-verteilte Transaktionen eine einstellbare Last. Die Rechenzeit c_τ einer Transaktion wird doppelt-exponentiell um einen Mittelwert verteilt angenommen. Für die Deadline einer Transaktion τ wird folgender Zusammenhang festgelegt (s bezeichnet den Slack-Faktor, $dexp_{\Delta x}()$ ist die doppelt-exponentielle Verteilung):

$$d = t_{\mathrm{global}} + c_\tau + dexp_{\Delta x}(s\ c_\tau). \tag{5.13}$$

Der Scheduler simuliert die Bearbeitung der Transaktion und die Zuteilung der CPU-Rechenzeit durch ein Realzeit-Betriebssystem (in dieser Arbeit EDF-Scheduling), der Logging-Mechanismus schreibt die Ergebnisse der Simulation (Blockierzeit, Deadlineverletzung etc.) für die spätere Auswertung in ein Datenfile.

Eine Darstellung der Ergebnisse für PRED-DF bei unterschiedlichen Systemauslastungen zeigt Abbildung 5.6. Aufgetragen ist die mittlere Blockierzeit $\overline{t_b}$ einer Transaktion und die Optimalität über den Blocking-Faktor b. Dieser beschreibt, für welchen Anteil der gesamten Transaktionslaufzeit $\overline{c_\tau}$ der Datensatz einer Transaktion gesperrt ist: $b = \overline{t_l}/\overline{c_\tau}$. Ein Blocking-Faktor von $b = 1$ entspricht einem gegenseitigen Ausschluß zweier Transaktionen und damit dem Protokoll PRED. Zusätzlich dargestellt ist jeweils ein in b quadratischer Fit an die Simulationsdaten. Die Simulation wurde bei einer mittleren Transaktionslänge $\overline{c_\tau} = 50$ und bei einer mittleren Laxity $\overline{l} = 250$ durchgeführt. Die Meßergebnisse sind der Mittelwert von jeweils 10^4 Messungen.

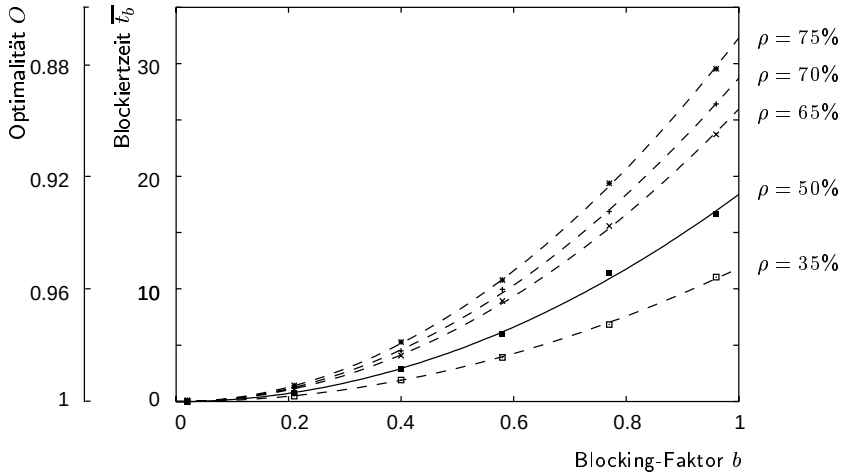


Abbildung 5.6.: Abhängigkeit der Optimalität und der mittleren Blockierzeit $\overline{t_b}$ einer Transaktion vom Blocking-Faktor b bei unterschiedlicher mittlerer Datenbankauslastung ρ. Ein Blocking-Faktor von $b = 1$ entspricht dem Protokoll PRED.

Effizienz Die *Effizienz* beschreibt, wie groß im Mittel für eine Transaktion der Zeitanteil t_{rwc} an der gesamten von einer Transaktion verbrauchten CPU-Zeit t_{gesamt} ist, der wirklich für Lese- und Schreiboperationen und für Berechnungen verwendet wurde:

$$\text{Effizienz } E := \frac{t_{\text{rwc}}}{t_{\text{gesamt}}} \tag{5.14}$$

Abbildung 5.7 gibt eine qualitative Einordnung verschiedener CC-Protokolle bezüglich dieser beiden Kriterien anhand der Literatur [AGM92][UB98]. Die Optimalität steigt mit sinkender Effizienz, da eine niedrigere Blockierungsrate eine höhere Zahl an Aborts/Restarts von Transaktionen bedingt. Durch seinen geringfügig gegenüber PRED erhöhten Verwaltungsaufwand liegt die Effizienz von PRED-DF etwas unter der von PRED. Sie liegt allerdings weiterhin über der Effizienz von Protokollen, die einzelne Datenelemente sperren, den Abbruch von Transaktionen zur Konfliktlösung verwenden, oder die aufwendige Verfahren zur Konsistenzsicherung einsetzten. Da keine Transaktionen abgebrochen werden, variiert die Effizienz von PRED-DF nicht mit der Konflikthäufigkeit im System. Die Optimalität von PRED-DF variiert mit dem Anteil der Transaktionen in azyklischen, normalisierten Konfliktmengen. Sie ist jedoch sicher genauso gut oder besser als die Optimalität von PRED.

Performance Mittels Simulationsrechnungen wurde zusätzlich die Performance des Protokolls PRED-DF untersucht. Abbildung 5.8 zeigt das Ergebnis. Dargestellt ist die Success Ratio S,

$$S := \frac{\text{innerhalb ihrer Deadline abgeschlossene TAs}}{\text{Gesamtzahl aller TAs}}, \tag{5.15}$$

in Abhängigkeit des Blocking Faktors b und dem Slack-Faktor s beziehungsweise der Transaktionslänge c_τ. Die Meßwerte sind jeweils das arithmetische Mittel von 10^4 Messungen bei einer mittleren Datenbankauslastung von $\rho = 60\%$. Ein Blocking-Faktor $b = 1$ entspricht dem Protokoll PRED.

Gut zu erkennen ist, daß die Erfolgsquote insbesondere von kurzen Transaktionen und von Transaktionen mit knapper Deadline, d. h. mit niedrigem Slack Faktor s, unter PRED stark

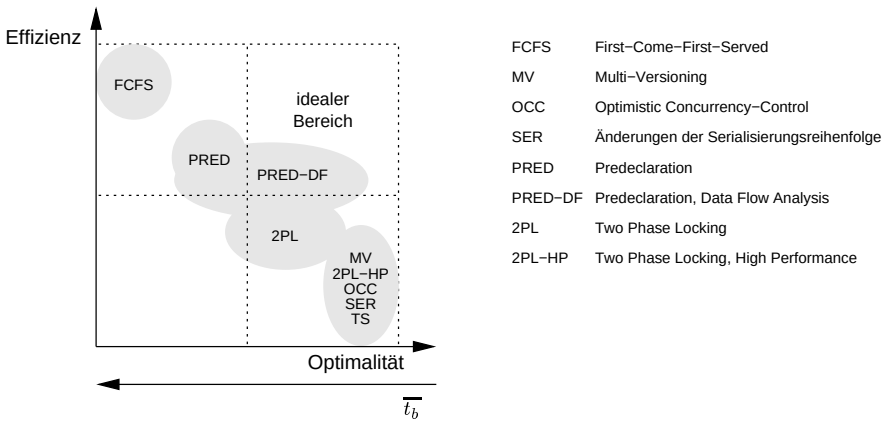


Abbildung 5.7.: Qualitative Einordnung von unterschiedlichen Concurrency-Control Protokollen bezüglich Optimalität und Effizienz. Nähere Details zu den einzelnen Protokollen finden sich in Abschnitt 2.2.1.

zurückgeht. Auf Grund der Verkürzung der Blockierzeiten unter PRED-DF ist S hier wesentlich höher.

Overestimation durch gegenseitige Blockierung

Der Nachweis der Realzeitfähigkeit einer Applikation erfordert es, stets den schlechtesten möglichen Fall zu berücksichtigen. Die Anforderungen an die benötigte Rechenzeit sind daher in der Regel äußerst pessimistisch kalkuliert und viel größer als im Durchschnitt benötigt (*Overestimation*). Wichtige Einflußgrößen hierbei sind vor allem die Worst-Case Execution Time von Transaktionen bzw. Tasks und die Bereiche gegenseitigen Ausschlusses.

Das oberste Ziel in einer Datenbank ist es, Serialisierbarkeit sicherzustellen. Gegenseitiger Ausschluß von Transaktionen ist aber nicht die unausweichliche Folge der Nutzung gemeinsamer Ressourcen, da die Daten der Datenbank prinzipiell replizierbar sind. Man hat daher stets die Möglichkeit, zwischen einer Realisierung mit größerer Blockierzeit und in der Regel höherer Effizienz einerseits und einer Implementierung mit niedrigerer Blockierzeit und niedrigerer Effizienz andererseits zu wählen. Beide Realisierungsvarianten haben unterschiedliche Auswirkungen auf die Kalkulation des Rechenzeitbedarfes. PRED-DF ist auf die Minimierung der Blockierzeiten hin optimiert. Zur Einordnung von PRED-DF bezüglich des Kriteriums „Overestimation“ und um die Vorteile kürzerer Blockierzeiten aufzuzeigen, werden im folgenden die Protokolle 2PL-HP/2PL-PA, PRED und PRED-DF diesbezüglich einander gegenübergestellt.

2PL-HP Die Worst-Case Execution Time bezeichnet immer die für eine Transaktion maximal benötigte CPU Zeit und nicht die maximale Antwortzeit. Falls eine Transaktion mit einer anderen Transaktion in Konflikt gerät, so wird bei diesem Protokoll die Transaktion mit der niedrigeren Priorität abgebrochen und später wieder neu aufgesetzt. Daher erhöht sich die Worst-Case Execution Time $\hat{c}_k$ einer abgebrochenen Transaktion τ_k um die bereits für diese Transaktion eingesetzte CPU-Zeit auf die *effektive Worst-Case Execution Time* $\hat{c}_{\mathrm{eff},k}$.

Das in dieser Arbeit verwendete Verfahren zur Verifikation von Realzeitanforderungen beruht auf der Annahme, daß alle Deadlines eingehalten werden und überprüft dann auf dieser Grundlage, ob die in einem bestimmten Zeitraum angeforderte Rechenzeit die zur Verfügung stehende

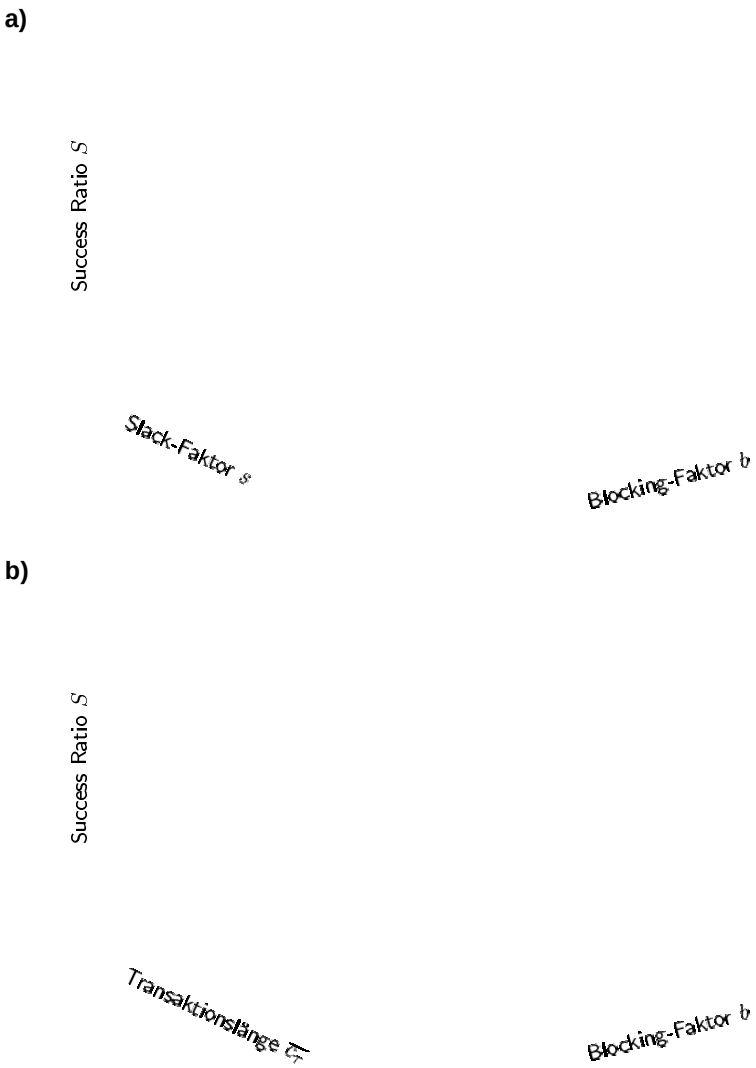


Abbildung 5.8.: Abhängigkeit der Erfolgsquote S einer Transaktion vom Blocking-Faktor b und dem Slack-Faktor s (Teilbild a) beziehungsweise der Transaktionslänge $\overline{c_\tau}$ (Teilbild b). Dargestellt ist das Ergebnis einer Simulation mit 10^4 Messungen und $\rho = 60\%$.

Rechenzeit übersteigt, $C(I) \leq I$. Daher kann die effektive Worst-Case Execution Time $\hat{c}_{\text{eff},k}$ einer Transaktion τ_k entsprechend Abbildung 5.9 folgendermaßen berechnet werden [MF00]:

$$\begin{aligned} \hat{c}_{\text{eff},k} &= \hat{c}_k + \sum_{j=1}^{L_k} \underbrace{(t_{ir,j} - t_{0,j})}_{< \hat{c}_k} \\ &\leq (1 + L_k)\, \hat{c}_k \end{aligned} \tag{5.16}$$

L_k ist die maximal mögliche Zahl an Unterbrechungen eines beliebigen τ_k durch andere Transaktionen τ_l, die mit τ_k auf Grund sich überschneidender Datenbereiche potentiell in Konflikt stehen. Diese werden in der Menge $\mathcal{C}_k$ zusammengefaßt. Mit Hilfe der jeweiligen Ereignisfunktionen ergibt sich für L_k:

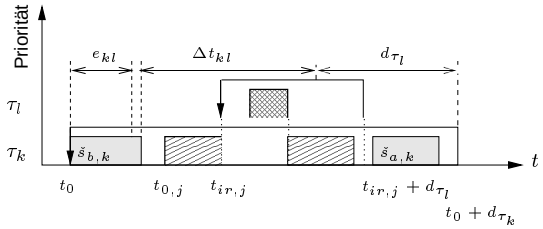


Abbildung 5.9.: Unterbrechung einer Transaktion τ_k durch eine Transaktion τ_l mit höherer Priorität beim CC-Protokoll 2PL-HP. Berechnung der Zeitspanne möglicher Unterbrechungen Δt_{kl}.

$$L_k = \sum_{\tau_l \in \mathcal{C}_k} E_r^l(\Delta t_{kl}) \tag{5.17}$$

E_r^l bezeichnet hier die der Transaktion τ_l zugeordnete Ereignisfunktion, Δt_{kl} den Zeitraum, in dem bei Berücksichtigung der Prioritäten und der Einhaltung aller Deadlines, ein Unterbrechung der Transaktion τ_k durch τ_l überhaupt stattfinden kann. Nach Abbildung 5.9 ist Δt_{kl} durch das folgende Gleichungssystem bestimmt:

$$\begin{aligned} t_{\mathrm{ir}} &> t_0 + \check{s}_{b,k} \\ t_{\mathrm{ir}} &< t_0 + d_{\tau_k} - \check{s}_{a,k} \\ t_{\mathrm{ir}} &< t_0 + d_{\tau_k} - d_{\tau_l} \\ t_{\mathrm{ir}} &> t_0 + e_{kl} \end{aligned}$$

Daraus berechnet sich Δt_{kl} zu:

$$\Delta t_{kl} = \max[0;\ d_{\tau_k} - \max(\check{s}_{b,k};\ e_{kl}) - \max(d_{\tau_l};\ \check{s}_{a,k})] \tag{5.18}$$

Statt $\hat{c}_k$ ist somit $\hat{c}_{\mathrm{eff},k}$ zum Zeitpunkt d_{τ_k} in der Rechenzeitanforderungsfunktion $C_a(I)$ zu berücksichtigen. Die effektive Worst-Case Execution Time ist im Konfliktfall immer mindestens doppelt so groß, wie der ursprüngliche Betrag. Die Effizienz sinkt bei häufigen Unterbrechungen stark.

PRED, PRED-DF Im Falle der Konfliktlösung durch Blockierung muß soviel zusätzliche Rechenzeit vorgehalten werden, daß höherpriore Tasks auch bei der Blockierung durch niederpriore Tasks und deren Verdrängung immer ihre Deadlines einhalten.

Falls eine Transaktion τ_y von einer Transaktion τ_x blockiert werden kann, läßt sich mittels der folgenden Erweiterungen der Arbeiten von Gresser die zusätzlich zu veranschlagende Rechenzeit kalkulieren. $\mathcal{V}$ bezeichnet hierbei die Menge aller Rechenzeitanforderungen, die τ_x verdrängen können, d. h. die eine Deadline $d < d_x$ haben und die noch nicht abgearbeitet sind. $\mathcal{P}$ bezeichnet die Menge aller Rechenzeitanforderungen mit $d = d_x$, die nicht bereits in $\max(\hat{s}_{b,x};\ e_{xy})$ berücksichtigt wurden – mit Ausnahme der blockierenden Transaktion τ_x selbst:

$$\mathcal{P}_x = \{\tau_i \neq \tau_x \mid d_{\tau_i} = d_{\tau_x} \wedge \tau_i \not\succ \tau_x\} \tag{5.19}$$

Es gilt folgende Herleitung:

$$
\begin{aligned}
C_a(d_y) + \hat{c}_x + \sum_{\mathcal{V}} \hat{c}_i &\leq d_y \\
C_r(d_x) + \hat{c}_x - \max(\hat{s}_{b,x};\ e_{xy}) &\leq d_y \\
C_a(d_x) - \max(\hat{s}_{b,x};\ e_{xy}) - \sum_{\mathcal{P}_x} \hat{c}_i &\leq d_y + d_x - d_x \\
\underbrace{C_a(d_x) + d_x - \max(\hat{s}_{b,x};\ e_{xy}) - \sum_{\mathcal{P}_x} \hat{c}_i - d_y}_{c_v} &\leq d_x
\end{aligned}
$$

Das heißt, die zusätzlich notwendige Rechenzeit kann, analog zu der Bedingung $C_a(I) \leq I$, durch einen virtuellen Task der Rechenzeit c_v und mit der Deadline d_x berücksichtigt werden:

$$
c_v = \max\left[0;\ d_x - \max(\hat{s}_{b,x};\ e_{xy}) - \sum_{\mathcal{P}_x} \hat{c}_i - d_y\right];\ d_v = d_x \tag{5.20}
$$

Verkürzt sich also die Rechenzeit der Blockierung von $\hat{c}_x$ auf $\hat{c}'_x$, so verringert sich c_v um denselben Betrag auf c'_v. Falls also $\tau_x \in \mathcal{T}_f(\tau_y)$, so verringert sich unter PRED-DF c_v im Vergleich zu PRED um die WCET der Read- bzw. Write-Phase und der Calculation-Phase der blockierenden Transaktion:

$$
c_v - c'_v = \hat{c}_{x,c} + \hat{c}_{x,(r|w)} \tag{5.21}
$$

Priority Inheritance Protokoll (PIP) Der durch die Verdrängung der blockierenden Transaktion bedingte, problematische Anstieg der Rechenzeit des virtuellen Tasks (Prioritätsinversion) kann durch Protokolle mit *Prioritätsvererbung* verhindert werden (Priority Inheritance) [SRL90]. Diese schließen Verdrängung im wesentlichen dadurch aus, daß ein blockierender Rechenprozeß auf die Priorität des blockierenden Rechenprozesses angehoben wird.

Im Falle der Prioritätsvererbung ist es für die Einhaltung aller Deadlines ausreichend, wenn sich die blockierende Transaktion τ_x als unabhängige Transaktion mit der Deadline d'_x der blockierten Transaktion einplanen läßt:

$$
d'_x = d_y \tag{5.22}
$$

Auch hier ist die zusätzlich einzuplanende Rechenzeit umso kürzer, je kleiner die Zeiten gegenseitigen Ausschlusses sind.

Tabelle 5.1 faßt die Resultate des Vergleichs zusammen. Die besten Ergebnisse liefert auf Grund der in der Regel wesentlich geringeren Blockierzeit der höherprioren Transaktionen PRED-DF in Verbindung mit PIP. 2PL-HP verbessert zwar die Situation für hochpriore Transaktionen, ist aber auf Grund des starken Rechenzeitanstieges für niederpriore Transaktionen problematisch.

Abbildung 5.10 macht den Unterschied der einzelnen Protokolle noch einmal anhand eines einfachen Beispiels deutlich. Abbildung a) zeigt die Situation, falls PRED-DF als CC-Protokoll verwendet wird. Wird PRED oder 2PL-HP verwendet, so schließen sich τ_1 und τ_2 gegenseitig komplett aus. Die Länge des virtuellen Tasks beträgt $c_{v,\,\text{PRED}} = 14$ im Falle von PRED und $c_{v,\,\text{PRED-DF}} = 5$ im Falle von PRED-DF, die effektive Rechenzeit der Transaktion τ_1 bei der

Protokoll		maximale Blockierzeit	zusätzliche Rechenzeit	Deadline
2PL-HP		0	$L_k\ \hat{c}_x$ $L_k \geq 0$	d_x
PRED		$\hat{c}_x + \sum_{\mathcal{V}} \hat{c}_i$	c_v	d_x
PRED-DF	$\tau_x \notin \mathcal{T}_f(\tau_y)$	$\hat{c}_x + \sum_{\mathcal{V}} \hat{c}_i$	c_v	d_x
	$\tau_x \in \mathcal{T}_f(\tau_y)$	$(\hat{c}_{x,r}$ oder $\hat{c}_{x,w}) + \sum_{\mathcal{V}} \hat{c}_i$	$c'_v \leq c_v$	d_x
PRED + PIP		$\hat{c}_x$	0	$d'(\hat{c}_x) = d_y$
PRED-DF + PIP	$\tau_x \notin \mathcal{T}_f(\tau_y)$	$\hat{c}_x$	0	$d'(\hat{c}_x) = d_y$
	$\tau_x \in \mathcal{T}_f(\tau_y)$	$\hat{c}_{x,r}$ oder $\hat{c}_{x,w}$		

Tabelle 5.1.: Vergleich der CC-Protokolle 2PL-HP, PRED und PRED-DF. Angenommen ist ein Konflikt zwischen den Transaktionen τ_x und τ_y, $d_x > d_y$. Die Tabelle zeigt jeweils die maximale Blockierzeit von τ_y, die in der Realzeitanalyse zusätzlich zu berücksichtigende Rechenzeit und die dafür geltende Deadline bzw. die Vorverlegung von Deadlines im Fall von PIP.

Verwendung von 2PL-HP $\hat{c}_{\text{eff},\tau_1} = 11 \cdot (1+1) = 22$. Aus der Abbildung ist leicht zu erkennen, daß in diesem Beispiel nur im Falle von PRED-DF die Deadlines immer eingehalten werden können. Der Rechenzeitgewinn PRED – PRED-DF ist für die Verwendung von PIP beispielhaft eingezeichnet.

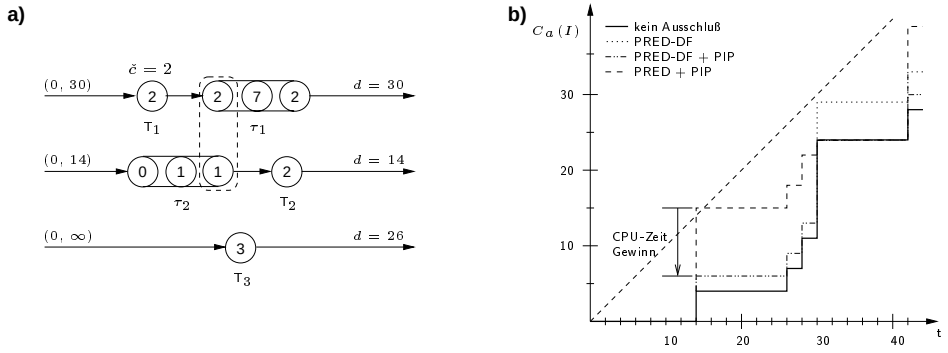


Abbildung 5.10.: Nachweis der Realzeitfähigkeit bei Konfliktauflösung unter PRED und PRED-DF. a) zeigt ein einfaches Beispiel, b) die dazugehörigen Rechenzeitanforderungsfunktionen $C_a(I)$.

Das Priority Inheritance Protokoll steht als Option in Verbindung mit PRED-DF im Datenbankprototypen dieser Arbeit zur Verfügung.

Priority Ceiling Protokoll (PCP) Durch die Verwendung von Priority Inheritance ist es möglich, eine obere Schranke für die Blockierzeiten anzugeben. Es bleiben jedoch zwei Probleme bestehen:

1. PIP ist nicht Deadlock-frei,
2. PIP schließt transitive Blockierungen nicht aus. Abhilfe bietet hier das Priority Ceiling Protokoll [SRL90].

In Verbindung mit dem CC-Protokoll PRED-DF bietet PCP allerdings keine weiteren Vorteile und wird in dieser Arbeit nicht weiter behandelt. Zum einen bedingt das PCP einen zusätzlichen Verwaltungsaufwand und es kann, wie in Abbildung 5.11 a) gezeigt, zusätzliche, unerwünschte Blockierungen induzieren. Zum anderen ist PRED-DF ohnehin Deadlock-frei. Außerdem können

transitive Blockierungen nur bis zur Tiefe zwei auftreten, da auf Grund der Tatsache, daß Transaktionen höchstens aus drei Phasen bestehen, die Anzahl der hinzukommenden Bereiche begrenzt ist (siehe Abbildung 5.11 b). Hinzukommende Bereiche sind kritische Bereiche, die innerhalb eines anderen Bereichs gegenseitigen Ausschlusses angefordert werden. Sie sind die Voraussetzung für die Entstehung transitiver Blockierungen [Gre93a].

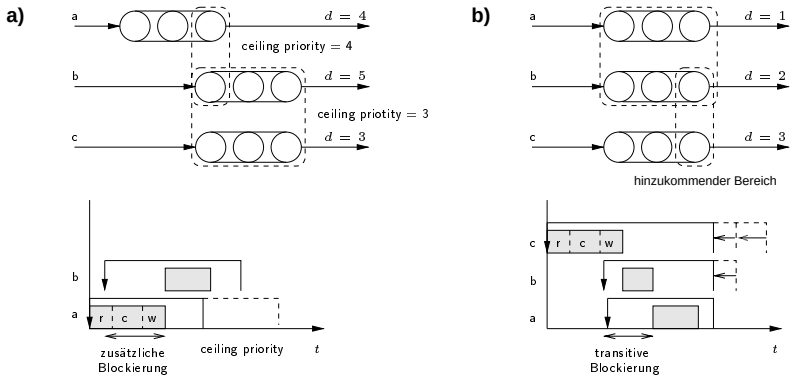


Abbildung 5.11.: a) Unerwünschte, zusätzliche Blockierungen bei der Verwendung von PCP in Verbindung mit PRED-DF, b) transitive Blockierung bei der Verwendung von PRED-DF in Verbindung mit PIP.

Integration von Informationen des Analysemodells und des Transaktions-Konflikt-Graphen

Durch die Kombination des Wissens aus dem Analysemodell des Realzeitnachweises und aus dem Transaktions-Konflikt-Graphen können zusätzliche Informationen gewonnen werden. Insbesondere kann die Abarbeitungsreihenfolge von Transaktionen z. B. aus deren Lage in den Präzedenzsystemen bestimmt werden. Dies erlaubt es, in bestimmten Fällen durch das Aufbrechen von Zyklen im TCG($\mathcal{T}$) größere Friend-Sets für eine Transaktion zu definieren und damit einen höheren Grad an Parallelität und kürzere Blockierzeiten in der Datenbank zu erzielen.

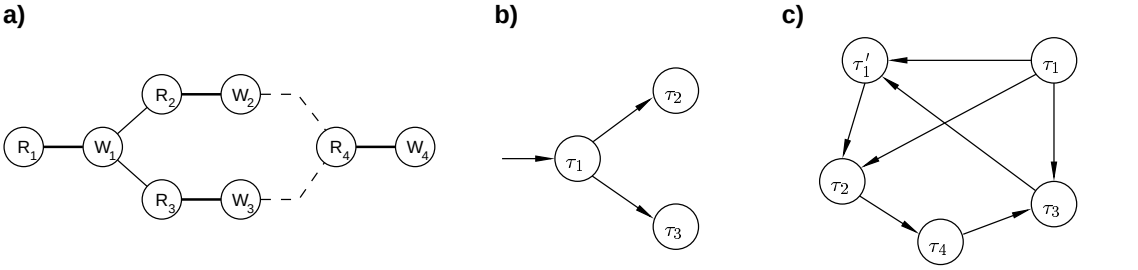


Abbildung 5.12.: Integration der Informationen a) aus dem Transaktions-Konflikt-Graphen einer Applikation und b) dem Analysemodell des Realzeitnachweises für dieselbe Applikation. c) zeigt ein Beispiel für $G(s)$, falls Bedingung (5.24) nicht erfüllt ist.

Es gilt: Existiert im TCG($\mathcal{T}$) ein Zyklus Z, so ist dieser für die Bildung von Friend-Sets nicht zu berücksichtigen, falls die folgenden beiden Bedingungen erfüllt sind:

- Es existieren drei Transaktionen $\tau_1, \tau_2, \tau_3 \in Z$, für die nach dem Analysemodell gilt:

$$\begin{aligned} \tau_1 &\succ \tau_2, \\ \tau_1 &\succ \tau_3. \end{aligned} \tag{5.23}$$

- Außerdem muß für die Abarbeitung der einzelnen Instanzen einer Transaktion die folgendne Bedingung erfüllt sein:

$$\begin{aligned} d_2, d_3 < a_1;\ d_1 \leq d_2, d_3 \quad \text{oder} \\ d_1 = d_2 = d_3. \end{aligned} \tag{5.24}$$

Hier ist durch (5.23) sichergestellt, daß für eine Instanz der Transaktion τ_1 der Graphen $G(s)$ immer nur solche Kanten enthalten kann, die von τ_1 weg gerichtet sind. Der Schluß eines Zyklus über τ_1 in $G(s)$ ist damit nicht möglich. (5.24) stellt sicher, daß bei mehreren, kurz hintereinander auftretenden Ereignissen, die die Bearbeitung von τ_1 auslösen, nicht ein Zyklus über nachfolgende Instanzen von τ_1 geschlossen werden kann (siehe Abbildung 5.12 c). Mit der ersten Bedingung ist die Bearbeitung von τ_2 und τ_3 vor dem Auftreten einer neuen Instanz von τ_1 bereits abgeschlossen, im zweiten Fall wird die Bearbeitung der neuen Instanz von τ_1 nicht vor der Bearbeitung von τ_2 und τ_3 gestartet.

Abbildung 5.12 zeigt ein Beispiel. a) zeigt einen Zyklus im TCG($\mathcal{T}$) einer Applikation, wie er beispielsweise im Anschluß an eine Sensor-Transaktion auftreten kann (hier τ_1). Falls im Analysemodell des Realzeitnachweises eine Konfiguration vorliegt, wie sie in b) dargestellt ist, kann ein Zyklus in $G(s)$ nie über τ_1 werden.

5.2.2. Betrachtung der aktiven Datenbankfunktionalität

In diesem Abschnitt wird die Abbildung der aktiven Datenbankfunktionalität auf das Analysemodell des Realzeitnachweises behandelt. Dazu müssen zum einen zusätzliche Rechenzeitanforderungen integriert werden. Zum anderen müssen im Realzeitnachweis Zyklen, die möglicherweise in der Applikation durch die aktiven Datenbankfunktionen entstehen können, berücksichtigt werden. Zusätzlich wird in diesem Abschnitt auf die Auswirkungen der in Abschnitt 4.3.3 dargestellten Möglichkeiten zur Optimierung eingegangen.

Abbildung auf das Analysemodell des Realzeitnachweises

Zur Abbildung der aktiven Funktionalität der Datenbank auf das Analysemodell des Realzeitnachweises müssen, neben den Taskknoten der Applikation, weitere Rechenzeitanforderungen berücksichtigt werden (*erweitertes Analysemodell*). Diese ergeben sich durch das Auslösen von Regeln der Regelbasis $\mathcal{R}$, das heißt z. B. durch Transaktionen zur Sicherung der Konsistenz oder durch Triggern zusätzlicher Tasks über interne Ereignisse der Applikation.

Die Integration dieser Rechenzeitanforderungen geschieht, wie in Abbildung 5.13 dargestellt, durch das Einfügen zusätzlicher Taskblocks $\mathrm{T}_{a,i}$ und zusätzlicher Kanten $\mathrm{E}_{\mathrm{ap},i}$ in das ursprüngliche Analysemodell. Der Taskblock $\mathrm{T}_{a,i}$ kapselt dabei alle Transaktionen τ_{ik} und τ_{ik}^c, die über die Regeln $\rho_{ik} \in \mathcal{R}$ mit dem Schreib-Datensatz W_{τ_i} der Transaktion τ_i verknüpft sind:

$$\rho_{ik} \in \{\rho_{lm} \mid x_l \in W_{\tau_i},\ \rho_{lm} \in \mathcal{R}\}. \tag{5.25}$$

Die Deadline von $\mathrm{T}_{a,i}$ ist identisch mit der Deadline d_i der auslösenden Benutzer-Transaktion τ_i. Äquivalent ergeben sich die Kanten $\mathrm{E}_{\mathrm{ap},i}$. Sie führen zu Taskknoten der Applikation.

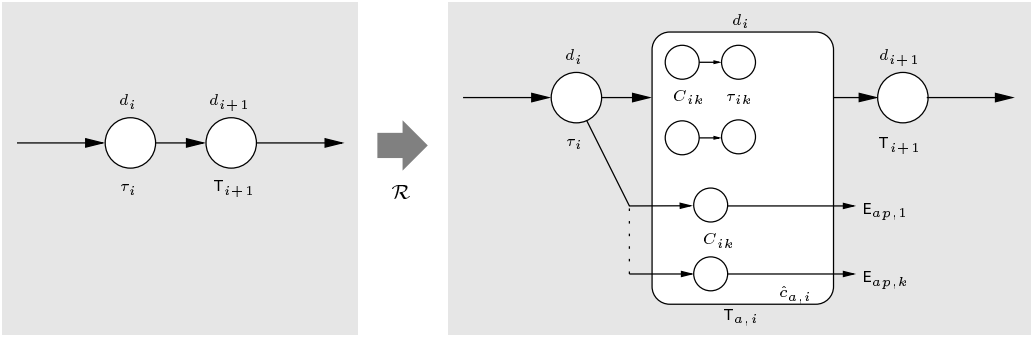


Abbildung 5.13.: Integration der zusätzlichen Rechenzeitanforderungen aus der aktiven Datenbankfunktionalität in das Analysemodell der Applikation.

Die Abarbeitungsreihenfolge der im Anschluß an τ_i sukzessive getriggerten Aktionen ist durch das Ausführungsmodell nicht festgelegt und daher beliebig. Sie kann beispielsweise von der Reihenfolge in der Systemspezifikation abhängig sein. Es können daher keine Präzedenzbeziehungen zwischen den Transaktionen innerhalb von $\mathrm{T}_{a,i}$ und für die Auslösung der Ereignisse angegeben werden, die in die Realzeitanalyse eingehen würden (siehe Abschnitt 5.2.1). Für die Berechnung des Startzeitpunktes einer Transaktion bzw. eines durch ein Ereignis $\mathrm{E}_{\mathrm{ap},i}$ ausgelösten Tasks ist lediglich bekannt, daß vorher die jeweils zugehörige Bedingung $C_{ik} = (c_{ik}, \tau_{ik}^c)$ ausgewertet werden muß.

Da nach dem Ausführungsmodell alle im Taskblock $\mathrm{T}_{a,i}$ zusammengefaßten Transaktionen nacheinander ausgeführt werden, kann kein gegenseitiger Ausschluß zwischen diesen Transaktionen auftreten, der in der Realzeitanalyse zu berücksichtigen wäre. Gegenseitiger Ausschluß von Transaktionen innerhalb von $\mathrm{T}_{a,i}$ und anderen Transaktionen der Applikation außerhalb von $\mathrm{T}_{a,i}$ können allerdings auftreten und sind in der Analyse zu berücksichtigen (siehe z. B. weiter unten: Optimierungsmöglichkeiten). Aus Sicht der innerhalb des Präzedenzsystems von τ_i nachfolgenden Taskknoten kann der Taskblock $\mathrm{T}_{a,i}$ als ein einziger Task mit der WCET

$$\hat{c}_{a,i} = \sum \hat{c}_{\tau_{ik}} + \sum \hat{c}_{C_{ik}};\ \tau_{ik}, C_{ik} \in \mathrm{T}_{a,i} \tag{5.26}$$

betrachtet werden.

Treten im Taskblock $\mathrm{T}_{a,i}$ Transaktionen auf, die wiederum neue Regeln auslösen, wird die in Abbildung 5.13 dargestellte Umformung rekursiv mehrmals hintereinander angewendet.

Für die Realzeitanalyse ist stets der pessimistischste Fall der Systemkonfiguration anzunehmen, d. h. für jede in der Datenbank enthaltene ECA-Regel, die von einer Transaktion ausgelöst werden kann, ist die zugeordnete Bedingung erfüllt. Ist allerdings bekannt, daß von mehreren Transaktionen jeweils nur ein Teil ausgeführt wird, oder daß sich die Auslösung mehrerer Regeln gegenseitig ausschließt, so kann dadurch die Realzeitanalyse genauer gefaßt werden. Allerdings ist dann für jede potentielle Kombination ein eigener Nachweis zu führen.

Zyklen. Endlosschleifen in einer Applikation, die eine aktive Datenbank zur Datenverwaltung nutzt, können zum einen durch Zyklen im Analysegraphen $A(\mathcal{D}, \mathcal{R})$, d. h. zyklische Datenabhängigkeiten, auftreten (siehe Abschnitt 4.3.2). Diese Zyklen sind nicht vermeidbar und müssen zur Laufzeit von der Datenbank abgebrochen werden. In der Realzeitanalyse werden

Zyklen bei der Erstellung des erweiterten Analysemodells ebenfalls abgebrochen. Es wird nur die Rechenzeit eines einzigen Durchlaufes berücksichtigt.

Zum anderen können Endlosschleifen in der Applikation durch zusätzliche interne Ereignisse der Applikation $E_{\mathrm{ap},i}$, die durch Regeln aus der Datenbank heraus ausgelöst werden, entstehen. Hier ergeben sich Zyklen im erweiterten Analysemodell des Realzeitnachweises. Solche Zyklen sind Fehler im Design der Anwendung, da die Bearbeitung eines Ereignisses in diesem Fall nicht terminiert. Sie müssen vom Anwendungsentwickler durch Designänderungen behoben werden.

Temporale Konsistenz kontinuierlicher Datenobjekte Wie bereits in Abschnitt 4.3.2 beschrieben, gilt nach (4.19) die folgende Bedingung für die temporale Konsistenz kontinuierlicher Umweltdaten:

$$\Delta t_i + d_i \leq T_{\max}(x_i). \tag{5.27}$$

Bei endlicher Worst-Case Execution Time $\hat{c}_i$ des Tasks T_i gilt zusätzlich die folgende Erweiterung der Ungleichung (4.20):

$$\hat{c}_i \leq d_i \leq \Delta t_i. \tag{5.28}$$

Die Aufteilung zwischen den beiden Parametern Δt_i und d_i ist variabel. Bezeichnet man das Verhältnis von Δt_i und $T_{\max}(x_i)$ mit $r := \frac{\Delta t_i}{T\max(x_i)}$, kann für ein bestimmtes r die mittlere Auslastung $\bar{\rho}_i$ der CPU durch T_i angegeben werden:

$$\bar{\rho}_i = \frac{\bar{c}_i}{r\, T_{\max}(x_i)}. \tag{5.29}$$

Auf Grund von $\bar{c}_i \leq d_i$ gilt stets $\rho_i \geq \bar{\rho}_0 = \lim_{r\to 1} \bar{\rho}_i = \frac{\hat{c}_i}{T\max(x_i)}$.

Das heißt, mit steigender Abtastperiode Δt_i und kleiner werdender Deadline d_i sinkt die mittlere Rechnerauslastung. Problematisch dabei ist der verringerte Spielraum für die Transaktionsbearbeitung durch die kleinere Deadline. Das gilt insbesondere bei mehreren unkorrelierten Tasks zur Pflege von kontinuierlichen Umweltdaten.

Dies ist auch in Abbildung 5.14 zu erkennen, die die Rechenzeit-Anforderungsfunktion $C_a(I)$ eines Tasks T_i exemplarisch für drei unterschiedliche Werte von r vergleicht. Im Mittel ist die benötigte Rechenzeit im Fall des größeren r niedriger. Am Anfang liegt sie allerdings auf Grund der knapperen Deadline höher. Bei $T_{\max}(x_i)$ ist $C_a(I)$ wegen (5.27) für alle r identisch.

Um eine optimale Lösung für eine Applikation zu erreichen, d. h. die Einhaltung aller Realzeitbedingungen bei minimaler mittlerer Auslastung der CPU, muß der Ausdruck

$$\bar{\rho}_m = \sum_{\mathrm{T}_i} \frac{\bar{c}_i}{\Delta t_i} \tag{5.30}$$

unter den Nebenbedingungen (5.27), (5.28) und (5.10) minimiert werden.

Im Augenblick existiert allerdings keine allgemeingültige mathematische Lösung für dieses nichtlineare Optimierungsproblem [XR99]. Die Angabe eines geschlossenen Algorithmus zur Optimierung ist daher nicht möglich. Ansätze zur Lösung des Problem können ebenfalls in [XR99] gefunden werden.

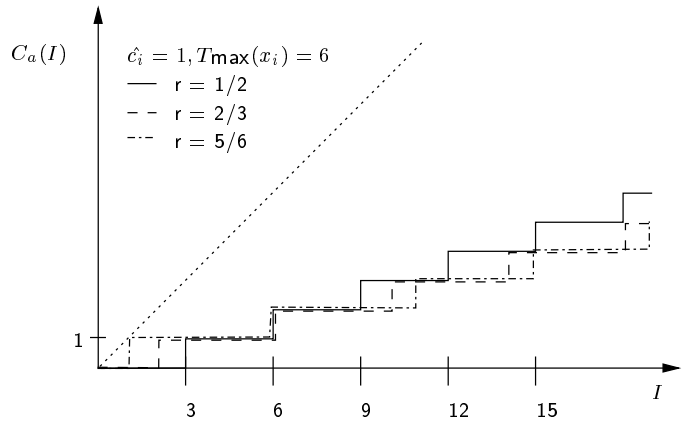


Abbildung 5.14.: Exemplarischer Vergleich der Rechenzeit-Anforderungsfunktion für drei unterschiedliche Werte von r.

Optimierungspotentiale

Die in 4.3.3 dargestellten Optimierungspotentiale werden in diesem Abschnitt exemplarisch auf ihre Wirkung in Bezug auf den Realzeitnachweis bei Systemen mit harten Zeitbedingungen untersucht. Nach der Optimierung ist im Idealfall die Rechenzeit-Anforderungsfunktion $C_a(I)$ für die optimierte Konfiguration für alle Intervalle kleiner oder gleich dem Wert der Ausgangskonfiguration. Ist dies nicht erreichbar, sollte zumindest die Auslastung der CPU sinken.

Fusion von kontinuierlichen Datenströmen Abbildung 5.15 zeigt ein Beispiel für die Fusion zweier Datenströme in einem Datenobjekt x_1. a) zeigt den Analysegraphen $A(\mathcal{D}, \mathcal{R})$ und die Zeitanforderungen $T_{\max}(x_1)$ und $T_{\max}(x_2)$ an die Daten, die sich aus der temporalen Konsistenz ergeben.

In b) ist die Abbildung auf die Implementierung, d. h. auf das erweiterte Analysemodell des Realzeitnachweises, im nicht optimierten Fall bzw. für $T_{\max}(x_1) \ll T_{\max}(x_2)$ dargestellt. Bei der Optimierung entfällt der in der Abbildung durchgestrichene Anteil des Analysemodells. Zu beachten ist der paarweise gegenseitige Ausschluß der Transaktionen, der sich auf Grund des Concurrency-Control Protokolls der Datenbank PRED-DF ergibt.

Abbildung 5.15 c) zeigt das erweiterte Analysemodell der Applikation für den in Abschnitte 4.3.3 dargestellten 2. Fall.

In Abbildung 5.16 sind die Rechenzeit-Anforderungsfunktionen für die unterschiedlichen Fälle aus Abbildung 5.15 dargestellt. Der Einfachheit halber wird von einer WCET $\hat{c}_i = 1$ für alle Tasks und Transaktionen ausgegangen. Für die Berechnung von $C_a(I)$ wird im ersten Fall für die Umweltdaten x_1 und x_2 von $\Delta t_1 = d_1 = 5$ und $\Delta t_2 = d_2 = 15$ ausgegangen. Bei der Berechnung von $C_a(I)$ ohne Optimierung muß zusätzlich der in Abbildung 5.15 b) dargestellte gegenseitige Ausschluß berücksichtigt werden. Der gegenseitige Ausschluß erhöht zusätzlich den Erfolg der Optimierung. Für das Beispiel wird Priority Inheritance angenommen. Dies bedeutet hier eine Vorversetzung der Deadline $d_2 = 15$ der längeren der beiden Transaktionen τ_3^k und τ_{31} auf $d_2' = d_1 = 5$. Im zweiten Fall wird von $\Delta t_1 = d_1 = \Delta t_2 = d_2 = 5$ ausgegangen.

Wie aus der Abbildung zu erkennen ist, liegt in beiden Fällen die Rechenzeit-Anforderungsfunktion unter der Rechenzeit in der nicht optimierten Konfiguration. Im Fall c) wäre ohne Optimierung eine rechtzeitige Bearbeitung der Transaktionen nicht möglich.

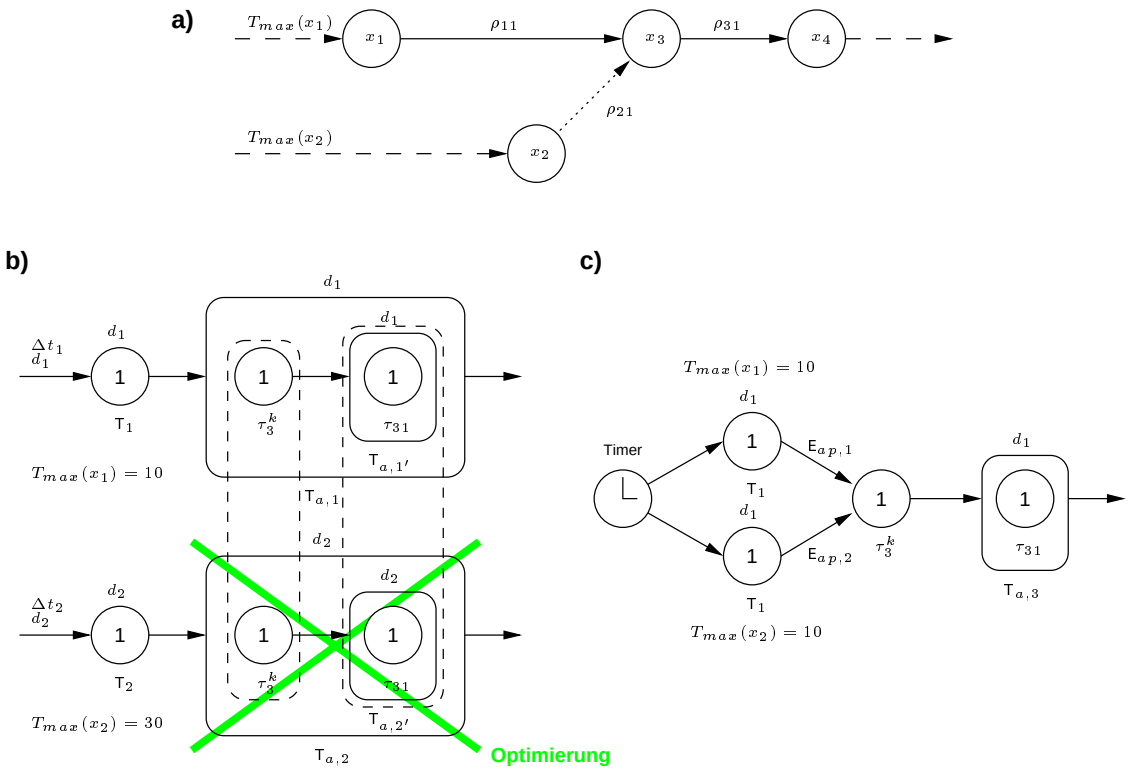


Abbildung 5.15.: Fusion zweier Datenströme in einem Datum x_1 (Beispiel). Optimierungsmöglichkeit bei der Sicherstellung temporaler Konsistenz. Dargestellt ist in a) der Analysegraph $A(\mathcal{D}, \mathcal{R})$. b) zeigt die Abbildung auf die Implementierung im Fall $T_{\min}(x_1) \ll T_{\min}(x_2)$, c) zeigt die Abbildung auf die Implementierung, falls der erzielbare Rechenzeitgewinn zu gering ist (siehe Abschnitt 4.3.3).

Aufspaltung eines kontinuierlichen Datenflusses Abbildung 5.17 zeigt ein Beispiel für die Aufspaltung eines Datenflusses in zwei Zweige mit unterschiedlichen Zeitanforderungen. a) zeigt wieder den Analysegraphen $A(\mathcal{D}, \mathcal{R})$ und die Angaben der Zeitanforderungen, die im Beispiel angenommen werden.

Bild 5.17 b) zeigt die Abbildung auf das erweiterte Analysemodell des Realzeitnachweises im nicht optimierten Fall. c) zeigt das erweiterte Analysemodell nach der Optimierung durch die Einführung eines zusätzlichen Tasks mit der längeren Deadline d_2, der die Transaktion τ_4^k und alle daraus folgenden Regeln bearbeitet.

Abbildung 5.18 zeigt die zu diesem Beispiel gehörigen Rechenzeit-Anforderungsfunktionen. Bei der Berechnung wird von $\Delta t_1 = 15$, $d_1 = 5$ und im Fall der Optimierung $\Delta t_2 = d_2 = 15 \leq \Delta t_1$ ausgegangen. Auch in diesem Beispiel verringert die Optimierung die Rechenzeit-Anforderungsfunktion der Transaktionen.

Ganz äquivalent kann mit ähnlichem Ergebnis die Optimierung bei unterschiedlichen Zeitanforderungen bezüglich eines Ausgangsdatums behandelt werden.

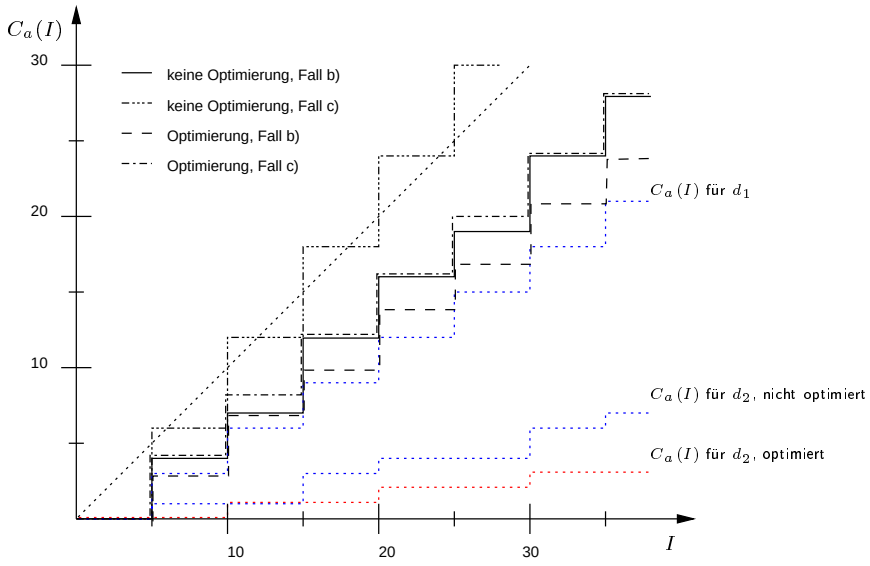


Abbildung 5.16.: Rechenzeit-Anforderungsfunktion für das Beispiel aus Abbildung 5.15 in der Ausgangskonfiguration und im optimierten Fall (die Bezeichnung der verschiedenen Konfigurationen bezieht sich auf Abbildung 5.15). Zusätzlich eingezeichnet sind verschiedene Zwischenergebnisse.

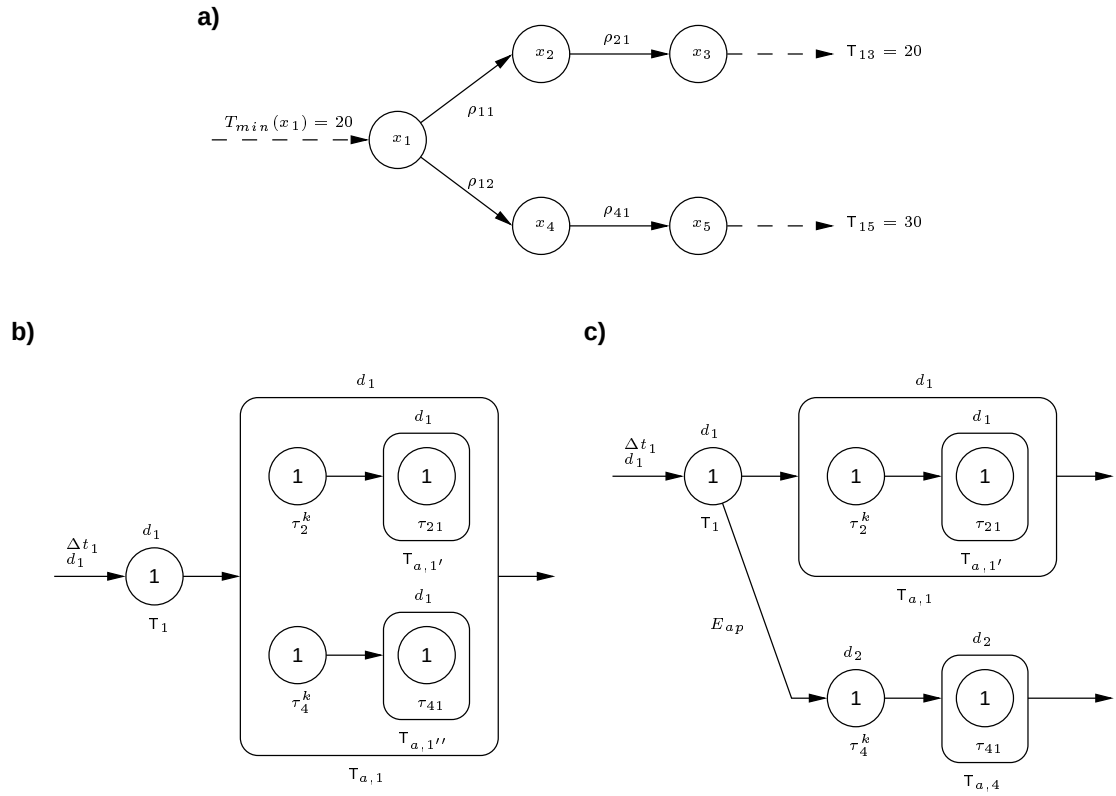


Abbildung 5.17.: Beispiel für die Aufspaltung eines Datenflusses in zwei Zweige mit unterschiedlichen Zeitanforderungen. a) zeigt den Analysegraphen $A(\mathcal{D}, \mathcal{R})$, b) gibt die Abbildung auf das erweiterte Analysemodell des Realzeitnachweises im nicht optimierten Fall. c) zeigt dieselbe Abbildung bei Optimierung durch die Einführung eines zusätzlichen Tasks.

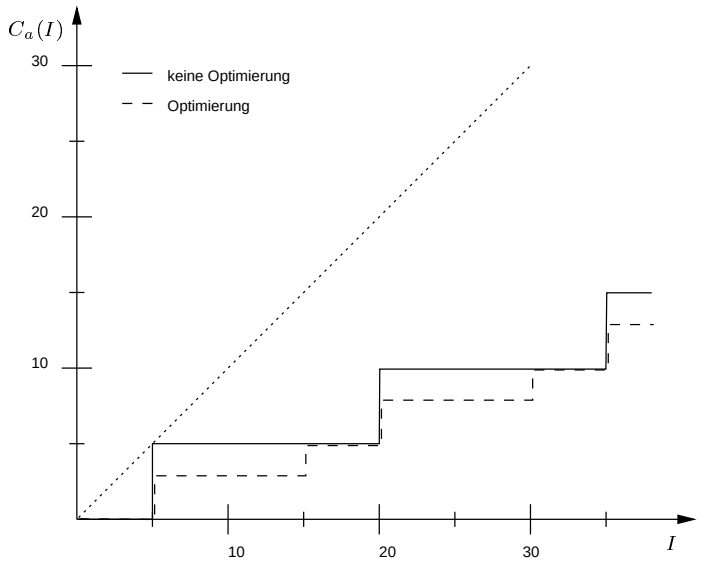


Abbildung 5.18.: Rechenzeit-Anforderungsfunktion für das Beispiel aus Abbildung 5.17 in der Ausgangskonfiguration und im optimierten Fall.

6. Zusammenfassung und Ausblick

In dieser Arbeit wurde ein neues Datenbankmodell für hauptspeicherbasierte, aktive Realzeitdatenbanken (ARTDBs) entwickelt, das für den Einsatz in datendominierten Realzeitsystemen mit harten Zeitanforderungen optimiert ist. So können die Vorteile des datenbankbasierten Applikationsdesigns auch für zeitkritische Applikationen genutzt werden. Das Verhalten der Datenbank ist vorhersagbar. Damit ist ein Echtzeitnachweis, d. h. die Verifikation von Zeitanforderungen im vorhinein, für die gesamte Anwendung, also für die Applikation in Verbindung mit der darin eingebetteten Datenbank, möglich. Hierzu wurde ein bereits bekanntes Verfahren zum Echtzeitnachweis für ereignisgesteuerte Realzeitsysteme erweitert und das Datenbankmodell dieser Arbeit darin integriert. Das neue Modell wurde prototypisch implementiert, getestet und auf seine Vorteile bezüglich des Einsatzes in harten Echtzeitsystemen hin analysiert.

Das Datenbankmodell hat im wesentlichen zwei Kernaspekte:

- *Entwicklung des neuen Concurrency-Control Protokolls (PRED-DF).* Mit PRED-DF wird durch die Nutzung zusätzlicher Informationen über den Datenfluß in der Applikation eine Minimierung der Blockierzeiten für einzelne Datensätze in der Datenbank erreicht, bei gleichzeitiger Vorhersagbarkeit des Datenbankverhaltens. Dadurch verkürzt sich die Verweildauer einer Transaktion im System. Außerdem ist ein Echtzeitnachweis mit hoher Genauigkeit möglich, da die zusätzlichen Rechenzeiten, die auf Grund potentieller Blockierungen berücksichtigt werden müssen, sehr gering sind.
- *Entwicklung eines Wissens- und Ausführungsmodells für ARTDBs in harten Realzeitsystemen.* Mittels der aktiven Funktionalität der Datenbank können Datenabhängigkeiten abgeglichen und die einbettende Applikation über spezifische Datenbankzustände unterrichtet werden. Die Regeln in der Datenbank werden dabei so ausgeführt (bzgl. Frequenz, Priorität etc.), daß zum einen die temporale Konsistenz der Datenobjekte und die rechtzeitige Abarbeitung von Regeln sichergestellt sind und zum anderen eine Einbindung in das Verfahren zum Realzeitnachweis erfolgen kann. Zusätzlich werden Optimierungsmöglichkeiten dargestellt, die es erlauben, die Effizienz bei der Abarbeitung der Konsistenzinformation zu verbessern.

Der bevorzugte Anwendungsbereich von Datenbanken in Echtzeitsystemen sind Applikationen, die einen umfangreichen Datensatz zu verwalten haben, auf den konkurrierend von unterschiedlichen Systemteilen aus zugegriffen wird. Beispiele hierfür sind, gemeinsam mit einigen charakteristischen Größen, in Tabelle 6.1 zusammengestellt. Angegeben sind die Größenordnungen für das Alter von kontinuierlichen Umweltdaten, die Abtasthäufigkeit und die dazugehörigen Deadlines. Zusätzlich gibt die „Response Time“ den ungefähren Zeitraum an, der bis zum Lesen und/oder Schreiben eines Datums in der Datenbank durch die Anwendung verstreichen darf.

Die Zeiten variieren stark mit dem Anwendungsgebiet. Sie bewegen sich aber alle im Bereich dessen, was beispielsweise mit der in dieser Arbeit durchgeführten Implementierung erreichbar ist.

Anwendungsgebiet	Charakteristika	
Luft- und Raumfahrt (Onboard-Computer)	Response Time	0.05 - 1 ms
	$T_{\max}(x_i)$	250 ms
	Δt_i	200 ms
	d_i	50 ms
Luftraumkontrolle	Response Time	0.05 - 5 ms
	$T_{\max}(x_i)$	4.5 s
	Δt_i	3 s
	d_i	1.5 s
Regelung/Steuerung (z. B. CIM-Systeme)	Response Time	1 - 10 ms
	$T_{\max}(x_i)$	700 ms
	Δt_i	200 ms
	d_i	500 ms
Steuerungs-/Kontrollsysteme (z. B. Space-Shuttle Kontrollzentrum)	Response Time	0.5 - 5 ms
	$T_{\max}(x_i)$	150 ms
	Δt_i	50 ms
	d_i	100 ms

Tabelle 6.1.: Beispiele für datendominierte Realzeitsysteme und deren charakteristische Größen [Loc00].

Allgemein ist der Einsatz von Datenbanken in Realzeitsystemen, insbesondere bei Systemen mit harten Zeitanforderungen, in der Industrie im Augenblick noch relativ wenig verbreitet. Dies liegt zum einen daran, daß die Größe und die Komplexität der betrachteten Systeme erst in letzter Zeit sehr stark zugenommen hat. Zum anderen sind „Realzeitdatenbanken“ und „aktive Realzeitdatenbanken“ insgesamt ein noch relativ junges Forschungsgebiet. Zusätzlich ist die Zahl der kommerziell verfügbaren Implementierungen vor allem für harte Realzeitanforderungen und bei aktiven Realzeitdatenbanken im Augenblick noch sehr gering.

Des weiteren bestehen eine Zahl bisher ungelöster Fragestellungen, die problematisch für den industriellen Einsatz solcher Datenbanken sind, und die in zukünftigen Forschungsarbeiten gelöst werden müssen. Die folgende Aufzählung nennt hierfür einige Beispiele:

- *Logging und Recovery.* Auch für harte Realzeitsysteme müssen Verfahren gefunden werden, die die Korrektheit und Dauerhaftigkeit der Informationen günstig und zuverlässig auch im Fehlerfall sicherstellen, die Archivierung von Vorgängen in der Datenbank über längere Zeiträume ermöglichen (Historian) und dabei gleichzeitig die Realzeitfähigkeit der Datenbank nicht beeinträchtigen.

- *Gemischte Systeme.* In der Regel gibt es keine umfangreichen Systeme, in denen ausschließlich nur harte oder nur weiche Zeitanforderungen vorliegen. Der Normalfall ist, daß stets beide Systemanteile in einem System nebeneinander existieren und miteinander wechselwirken. Das heißt, auch auf der Datenbank arbeiten Transaktionen mit harten und weichen Realzeitbedingungen parallel. Trotz der Erzielung einer hohen Systemperformance muß dabei weiterhin sichergestellt sein, daß alle Transaktionen mit harten Zeitbedingungen ihre Zeitschranken einhalten, also nicht durch Transaktionen mit weichen Zeitanfoderungen blockiert werden können. Hierzu müssen entsprechende Concurrency-Control Protokolle entwickelt werden.

- *Verteilte Systeme.* Viele Systeme werden inzwischen verteilt implementiert. Daher muß auch die Datenbank in der Lage sein, Verteilung zu handhaben. Hier müssen entsprechende

Methoden entwickelt werden, um beispielsweise die systemweite logische und temporale Konsistenz des Datenbestandes sicherzustellen.

- *Toolunterstützung.* Sowohl die Spezifikation des Datenbankinhaltes als auch die Spezifikation von Regeln in einer aktiven Realzeitdatenbank müssen durch entsprechende Werkzeuge unterstützt werden, um eine einfache Systementwicklung und -wartung zu ermöglichen. Das gleiche gilt für die entsprechenden Verfahren zum Realzeitnachweis.

Insgesamt kann die intensivere Nutzung von Datenbanken auch im Bereich der Realzeitsysteme mit harten Zeitanforderungen sicherlich dazu beitragen, daß diese Systeme leistungsfähiger werden und dabei einfacher, schneller und sicherer entwickelt werden können. Mit dieser Arbeit wurde versucht, einen Beitrag dazu zu leisten.

A. Anhang: Anwendungsbeispiele

Dieses Kapitel stellt zwei unterschiedliche Anwendungsbeispiele für das in dieser Arbeit entwickelte Modell einer aktiven Realzeitdatenbank vor. In Abschnitt A.1 wird anhand einer Fallstudie exemplarisch der Einsatz der grundlegenden Prinzipien des Datenbankmodells (aktive Funktionalität, Concurrency-Control Protokoll PRED-DF) und der hierfür entwickelten Methode zum Realzeitnachweis dargestellt. Die Vorteile von PRED-DF gegenüber anderen Concurrency-Control Protokollen (PRED, 2PL-HP) werden ebenfalls diskutiert. Bei der zu Grunde liegenden Aufgabenstellung handelt es sich um die Steuerung und Regelung einer 2-achsigen, automatischen Fräsmaschine. Abschnitt A.2 gibt ein Beispiel für einen potentiellen industriellen Einsatz des Datenbankmodells anhand eines allgemeinen Architekturmodells für datendominierte Realzeitsysteme, dessen zentrales Element eine aktive Realzeitdatenbank bildet.

A.1. Steuerung und Regelung einer 2-achsigen Fräsmaschine (Fallstudie)

A.1.1. Aufgabenstellung

Die Aufgabenstellung dieser Fallstudie besteht darin, in Teilen die Steuerung einer 2-achsigen automatischen Fräsmaschine zu realisieren. Diese besteht im Zentrum aus einem in x- und y-Richtung verfahrbaren Tisch (xy-Tisch), der das zu bearbeitende Werkstück trägt. Für die Materialbearbeitung steht eine in z-Richtung liegende, rotierende Spindel mit Fräser zur Verfügung. Die Position des Fräsers kann nicht verändert werden.

Von der zentralen Steuerungseinheit müssen unter anderem die folgenden Aufgaben bearbeitet werden:

1. *Steuerung und Regelung des xy-Tisches.* Die relative Positionsaufnahme für eine Achse erfolgt über einen Drehwinkelgeber, der auf der jeweiligen Antriebsachse befestigt ist und einen Impuls pro 10 μm Linearverschiebung des xy-Tisches in eine Richtung gibt. Die Sollwerte für die relative x- und y-Position x_{ref} und y_{ref} werden von der Steuereinheit aus dem aktuell gültigen Nullpunkt des Koordinatensystems $\vec{x}_0$ und den in einem Eingabefile vorliegenden CAD-Daten berechnet. Um den Konturfehler $\epsilon = \sqrt{(x - x_{ref})^2 + (y - y_{ref})^2}$ beim Fräsen zu minimieren, erfolgt die Positionsregelung der beiden Achsen mittels eines Cross-Coupled Controllers (CC-Controller) [YTCS98]. Abbildung A.1 zeigt die schematische Darstellung eines solchen Reglers.

   Aus Sicherheitsgründen sind an beiden Achsen des Frästisches Endschalter fest installiert. Werden diese vom xy-Tisch angefahren, ist der Vorschub in diese Richtung für diese Achse sofort anzuhalten, d. h. nach dem Ansprechen der Endschalter darf sich der Tisch, um eine mechanische Beschädigung der Anlage auszuschließen, noch höchstens 0.5 mm in dieselbe

Richtung weiterbewegen. Danach soll eine Bewegung in die entgegengesetzte Richtung weiterhin möglich sein.

2. *Visualisierung.* Zum einen soll der Zustand der Fräsmaschine visualisiert werden (z. B. der Zustand der Sicherheitseinrichtungen). Zum anderen soll eine Grafik über den Fortgang der Fräsung dargestellt werden (z. B. Ist-/Sollwert Vergleich). Der Benutzer kann für diese Darstellung entweder direkt einen Bereich wählen (`START/STOP`), oder er gibt das Zentrum der Darstellung und deren Maßstab an (`CENTER/ZOOM`). Beide Wertepaare werden zusätzlich auch noch an anderen Stellen der Applikation benötigt.

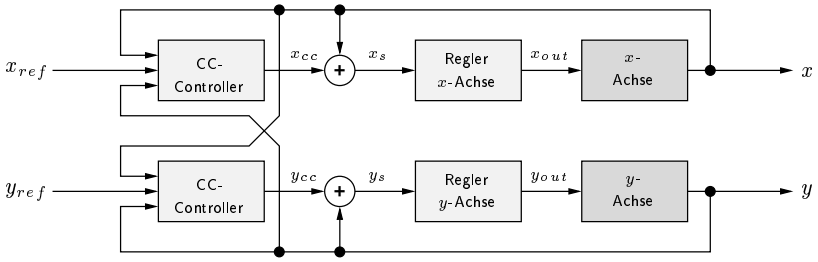


Abbildung A.1.: Cross-Coupled Controller zur Regelung der Bewegung von x- und y-Achse des Frästisches.

A.1.2. Lösung

Abbildung A.2 zeigt das erweiterte Analysemodell des Realzeitnachweises für eine mögliche Lösung der Fallstudie (aus Gründen der Übersichtlichkeit sind nur die Tasks für die Behandlung der Endschalter an der x-Achse eingezeichnet; die Implementierung für die y-Achse erfolgt völlig analog). Die Informationen über die aktiven Inhalte der Datenbank sind in diese Darstellung bereits integriert.

In der nachfolgenden Übersicht werden die Aufgaben der einzelnen Tasks und Transaktionen stichwortartig erläutert:

- Die Signale der Drehwinkelgeber und der Endschalter werden mittels Interrupts und den dazugehörigen Interrupt-Serviceroutinen (ISR) bearbeitet. Diese geben dann den jeweils ersten Task des „verantwortlichen“ Präzedenzsystems frei. Es wird davon ausgegangen, daß die für die Implementierung verwendete Hardware mit flankengetriggerten Interrupts arbeitet.

- Task T_1 bearbeitet den Interrupt des Drehwinkelgebers an der x-Achse und setzt den jeweils aktuellen Wert für die relative x-Koordinate über die Transaktion τ_1 in der Datenbank. Die weitere Verarbeitung des x-Wertes wird dann über die entsprechenden Regeln der Datenbank gesteuert (siehe Tabelle A.2). Zum einen ist so sichergestellt, daß der aktuelle x-Wert stets allen Systemteilen zur Verfügung steht. Gleichzeitig werden Neuberechnungen nur nach dem Setzen eines neuen x-Wertes in der Datenbank angestoßen.

  τ_2 berechnet aus den aktuellen x- und y-Werten und aus x_{ref} das Datum x_{cc} (CC-Controller). τ_3 berechnet die Summe x_s. τ_4 enthält den eigentlichen Regelalgorithmus für die x-Achse und schreibt x_{out} (ρ_1, ρ_2, ρ_3). Am Ende der Kette wird von τ_4 über das interne Ereignis $E_{ap,1}$ der Task T_2 gestartet (ρ_4), der mit τ_5 den aktuellen Stellwert x_{out} aus der Datenbank entnimmt und die Aktorik, d. h. den Motor, entsprechend ansteuert (T_2').

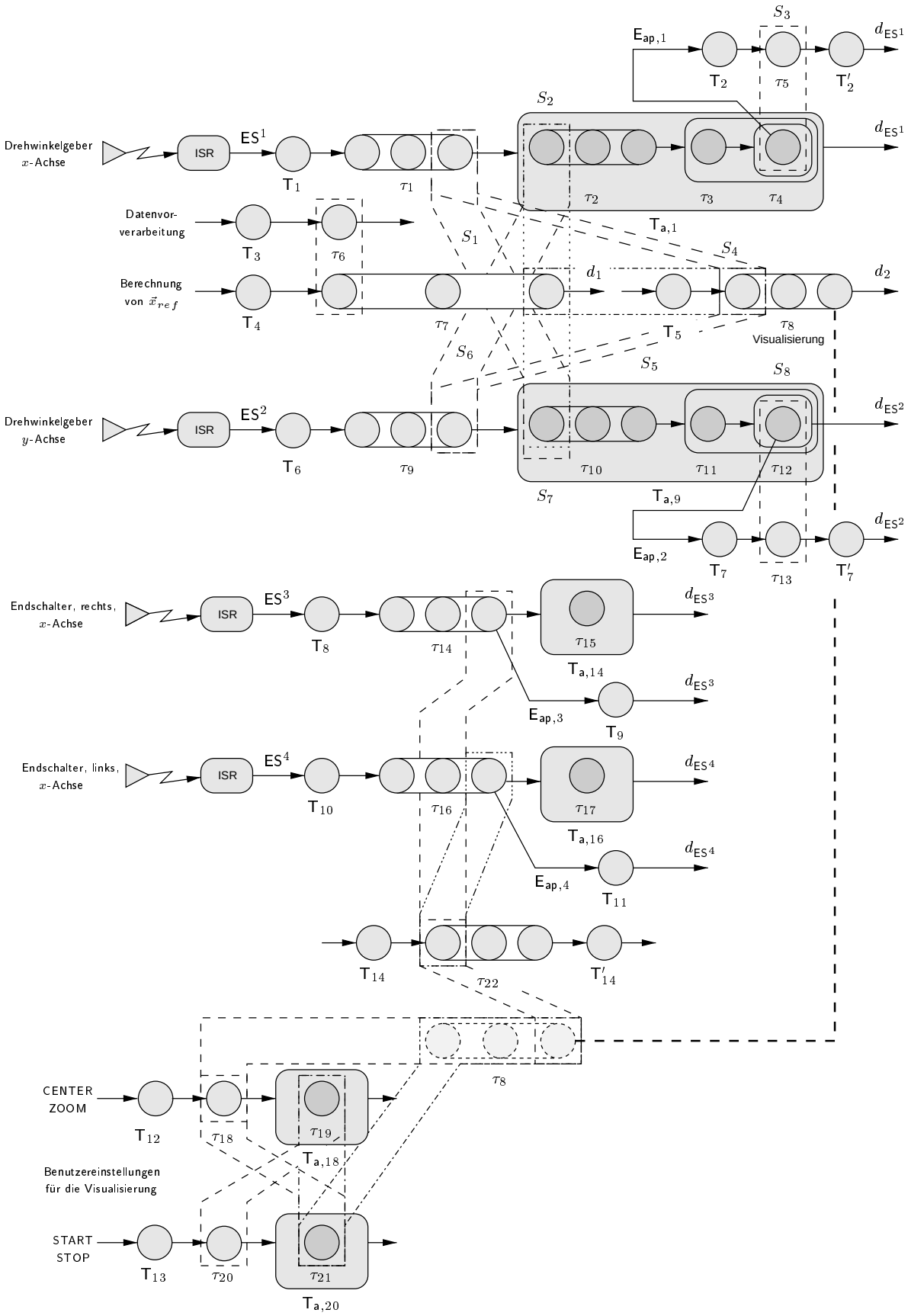


Abbildung A.2.: Realisierung der Fallstudie. Erweitertes Analysemodell des Realzeitnachweises (in Ausschnitten). Aus Gründen der Übersichtlichkeit sind bei normalisierten Transaktionen die drei Bearbeitungsphasen nur dann getrennt gezeichnet, wenn sie Bereiche mit unterschiedlichem gegenseitigem Ausschluß enthalten.

- Genau analog regeln die Tasks T_6 und T_7, sowie die Transaktionen $\tau_9, \tau_{10}, \tau_{11}, \tau_{12}$ und τ_{13} die y-Achse (ρ_4 bis ρ_8).

- T_3 entnimmt aus dem CAD-File die jeweils nächsten Frässtrecken, bereitet sie auf und legt sie vorverarbeitet über τ_6 als Segment `Seg` (z. B. Kreisbogen, Gerade) in der Datenbank ab. T_4 und τ_7 berechnen aus dem augenblicklich aktuellen Segment `Seg` und dem Ursprung des Koordinatensystems $\vec{x}_0$ die Werte für x_{ref} und y_{ref} (Zerlegung in einzelne Geradenstücke).

- Zur Überwachung der Endschalter an der x-Achse dienen die Tasks T_8 und T_{10}. Diese verarbeiten den dazugehörigen Interrupt und setzen den neuen Wert in der Datenbank (τ_{14} und τ_{16}). Daraufhin wird über entsprechende Regeln ρ_9 bzw. ρ_{10} mit den Transaktionen τ_{15} und τ_{17} geprüft, ob eine gefährliche Situation vorliegt. Gegebenenfalls werden über die Ereignisse $E_{ap,3}$ und $E_{ap,4}$ die Tasks T_9 und T_{11} angestoßen, die für das Abschalten des Achsantriebes sorgen. Parallel dazu stehen die Daten zur Visualisierung des Maschinenstatus zur Verfügung. Die Überwachung der y-Achse erfolgt analog, ist aber aus Gründen der Übersichtlichkeit nicht dargestellt.

- Über Task T_{12} und Transaktion τ_{18} hat der Benutzer die Möglichkeit, `CENTER` und `ZOOM` für die graphische Visualisierung zu wählen. Mittels T_{13} und τ_{20} kann er `START` und `STOP` setzen. Der automatische Abgleich der beiden Darstellungen geschieht über die Transaktionen τ_{19} und τ_{21} (ρ_{11}, ρ_{12}). Bei einer Parameteränderung werden stets beide Werte in der Datenbank gesetzt. Pro Parameterpaar ist also eine Regel zur Überwachung ausreichend.

- Task T_5 liest über Transaktion τ_8 die für die Visualisierung der Fräsung notwendigen Daten aus der Datenbank aus, bereitet sie auf und legt sie im Datensatz x_v ab. Dort wird sie von T_{14} und τ_{22} entnommen und gemeinsam mit dem restlichen Systemzustand zur Darstellung gebracht (T'_{14}).

Aus dem Transaktions-Konflikt-Graphen TCG($\mathcal{T}$) in Abbildung A.3 können die Friend-Sets der einzelnen Transaktionen bestimmt werden. Aus der Aufteilung der Transaktionen in Friend-Sets ergeben sich wiederum die Bereiche gegenseitigen Ausschlusses der Transaktionen untereinander (S_i). Diese sind im erweiterten Analysegraphen in Abbildung A.2 eingezeichnet.

Es wird davon ausgegangen, daß alle Transaktionen normalisierbar sind. Innerhalb der beiden Zyklen Z_1 und Z_2 ist gemäß dem erweiterten Analysemodell in Abbildung A.2 sichergestellt, daß τ_1 stets vor τ_2 und τ_3 bzw. τ_9 immer vor τ_{10} und τ_{11} ausgeführt wird. Ein Zyklus im Serialisierbarkeitsgraphen kann zur Laufzeit damit nicht entstehen (siehe Abschnitt 5.2.1, Integration von Informationen des Analysemodells und des Transaktions-Konflikt-Graphen). Zyklen im Transaktions-Konflikt-Graphen, die über $\tau_1, \tau_2, \tau_7, \tau_8, \tau_9, \tau_{10}$ geschlossen werden, können entsprechend Abschnitt 4.2.6 für die Bestimmung der Friend-Sets unberücksichtigt bleiben, da entweder nur die Lese- oder nur die Schreibmenge einer Transaktion in diesen Zyklen enthalten ist. Auf Grund der Zyklen im Transaktions-Konflikt-Graphen können in $\mathcal{T}_{kc}$ keine Friend-Sets definiert werden. Tabelle A.1 gibt abschließend eine Übersicht über die Lese- und Schreibdatensätze und die Friend-Sets aller Transaktionen der Applikation.

Die Regelbasis $\mathcal{R}$ ist entsprechend Tabelle A.2 definiert. Da bei der Wahl des Darstellungsbereiches stets `START` und `STOP` oder `CENTER` und `ZOOM` gesetzt werden, reichen zur Angleichung der Parameter lediglich zwei Regeln (ρ_{11} und ρ_{12}) aus. Die Variable `MotorRichtungX` enthält die augenblickliche Bewegungsrichtung des Vorschubs in x-Richtung; der entsprechende Wert wird von der Motorsteuerung zur Verfügung gestellt. Die flankengetriggerten Interrupts stellen sicher, daß nach dem Anfahren eines Endschalters der Frästisch noch in die entgegengesetzte Richtung bewegt werden kann.

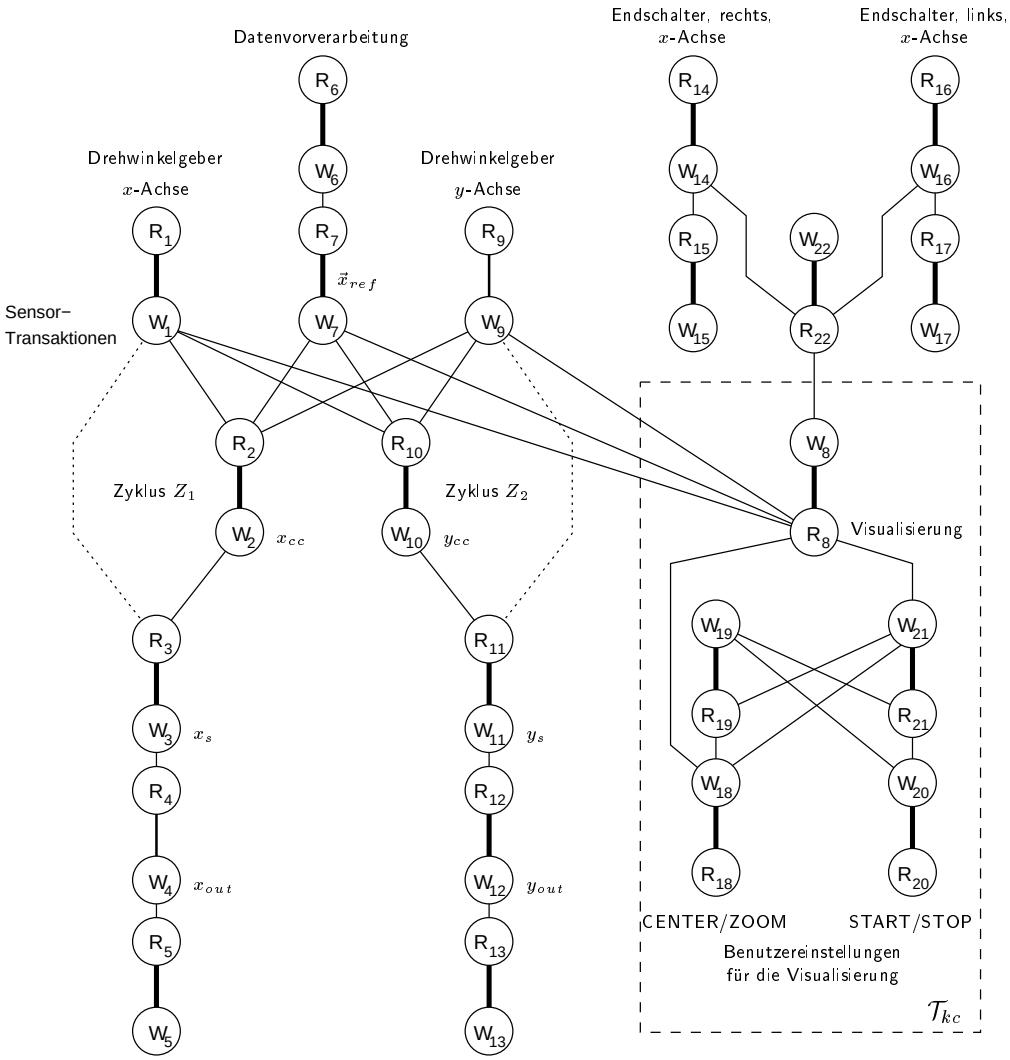


Abbildung A.3.: Realisierung der Fallstudie. Transaktions-Konflikt-Graph TCG($\mathcal{T}$).

TA	Lesemenge	Schreibmenge	$\mathcal{T}_f(\tau_i)$
τ_1	–	x	$\mathcal{T}_f(\tau_1) = \{\tau_2, \tau_3, \tau_8, \tau_{10}\}$
τ_2	x, y, x_{ref}	x_{cc}	$\mathcal{T}_f(\tau_2) = \{\tau_1, \tau_3, \tau_7, \tau_9\}$
τ_3	x, x_{cc}	x_s	$\mathcal{T}_f(\tau_3) = \{\tau_1, \tau_2, \tau_4\}$
τ_4	x_s	x_{out}	$\mathcal{T}_f(\tau_4) = \{\tau_3, \tau_5\}$
τ_5	x_{out}	–	$\mathcal{T}_f(\tau_5) = \{\tau_4\}$
τ_6	–	Seg	$\mathcal{T}_f(\tau_6) = \{\tau_7\}$
τ_7	Seg, $\vec{x}_0$	x_{ref}, y_{ref}	$\mathcal{T}_f(\tau_7) = \{\tau_2, \tau_6, \tau_8, \tau_{10}\}$
τ_8	x, y, x_{ref}, y_{ref}, CENTER, ZOOM	x_v	$\mathcal{T}_f(\tau_8) = \{\tau_1, \tau_7, \tau_9, \tau_{22}\}$
τ_9	–	y	$\mathcal{T}_f(\tau_9) = \{\tau_2, \tau_8, \tau_{10}, \tau_{11}\}$
τ_{10}	y, x, y_{ref}	y_{cc}	$\mathcal{T}_f(\tau_{10}) = \{\tau_1, \tau_7, \tau_9, \tau_{11}\}$
τ_{11}	y, y_{cc}	y_s	$\mathcal{T}_f(\tau_{11}) = \{\tau_9, \tau_{10}, \tau_{12}\}$
τ_{12}	y_s	y_{out}	$\mathcal{T}_f(\tau_{12}) = \{\tau_{11}, \tau_{13}\}$
τ_{13}	y_{out}	–	$\mathcal{T}_f(\tau_{13}) = \{\tau_{12}\}$
τ_{14}	–	$e_{x,r}$	$\mathcal{T}_f(\tau_{14}) = \{\tau_{22}, \tau_{15}\}$
τ_{15}	$e_{x,r}$	–	$\mathcal{T}_f(\tau_{15}) = \{\tau_{14}\}$
τ_{16}	-	$e_{x,l}$	$\mathcal{T}_f(\tau_{16}) = \{\tau_{22}, \tau_{17}\}$
τ_{17}	$e_{x,l}$	–	$\mathcal{T}_f(\tau_{17}) = \{\tau_{16}\}$
τ_{18}	–	CENTER, ZOOM	$\mathcal{T}_f(\tau_{18}) = \{\}$
τ_{19}	CENTER, ZOOM	START, STOP	$\mathcal{T}_f(\tau_{19}) = \{\}$
τ_{20}	–	START, STOP	$\mathcal{T}_f(\tau_{20}) = \{\}$
τ_{21}	START, STOP	CENTER, ZOOM	$\mathcal{T}_f(\tau_{21}) = \{\}$
τ_{22}	$x_v, e_{x,r}, e_{x,l}$	–	$\mathcal{T}_f(\tau_{21}) = \{\tau_8, \tau_{14}, \tau_{16}\}$

Tabelle A.1.: Überblick über alle Transaktionen der Fallstudie, ihre Lese- und Schreibmengen und die jeweiligen Friend-Sets $\mathcal{T}_f(\tau_i)$.

Regel	Bedingung
$\rho_1 := (\ w(x),\ t,\ \tau_2\ )$	–
$\rho_2 := (\ w(x_{cc}),\ t,\ \tau_3\ )$	–
$\rho_3 := (\ w(x_s),\ t,\ \tau_4\ )$	–
$\rho_4 := (\ w(x_{out}),\ t,\ \mathrm{E}_{ap,1}\ )$	–
$\rho_5 := (\ w(y),\ t,\ \tau_{10}\ )$	–
$\rho_6 := (\ w(y_{cc}),\ t,\ \tau_{11}\ )$	–
$\rho_7 := (\ w(y_s),\ t,\ \tau_{12}\ )$	–
$\rho_8 := (\ w(y_{out}),\ t,\ \mathrm{E}_{ap,2}\ )$	–
$\rho_9 := (\ w(e_{x,r}),\ c_1,\ \mathrm{E}_{ap,3}\ )$	$C_9 := (\ e_{x,r} \equiv 1 \wedge$ MotorRichtungX $\equiv$ rechts, $\tau_{15}\ )$
$\rho_{10} := (\ w(e_{x,l}),\ c_2,\ \mathrm{E}_{ap,4}\ )$	$C_{10} := (\ e_{x,l} \equiv 1 \wedge$ MotorRichtungX $\equiv$ links, $\tau_{17}\ )$
$\rho_{11} := (\ w(\mathtt{CENTER}),\ t,\ \tau_{19}\ )$	–
$\rho_{12} := (\ w(\mathtt{START}),\ t,\ \tau_{21}\ )$	–

Tabelle A.2.: Regelbasis $\mathcal{R}$ für die Implementierung der Fallstudie.

Zum Schluß gibt der Analysegraph $A(\mathcal{D}, \mathcal{R})$ in Abbildung A.4 einen Überblick über die Abhängigkeiten der Datensätze in der Applikation untereinander (durchgezogene Linien). Zusätzlich eingezeichnet sind andere Abhängigkeiten der Daten untereinander, die nicht von der Datenbank angepaßt werden (gestrichelte Linien).

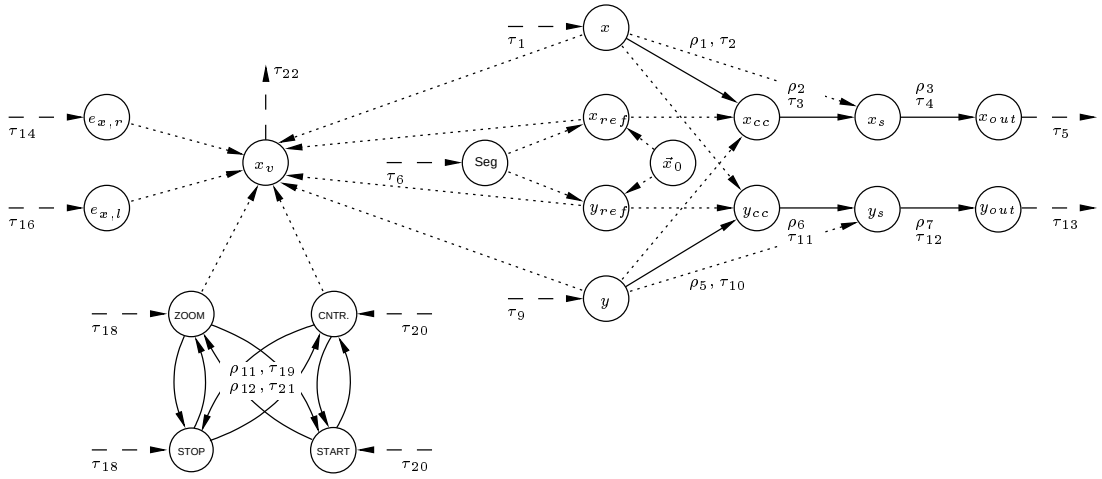


Abbildung A.4.: Realisierung der Fallstudie. Analysegraph $A(\mathcal{D}, \mathcal{R})$. Durchgezogene Linien symbolisieren Datenabhängigkeiten, die durch Regeln in der Datenbank automatisch sichergestellt werden.

A.1.3. Realzeitanalyse

In dieser Fallstudie stellen die Regelung der Achsen und die Überwachung der Endschalter Aufgaben mit harten Deadlines dar. Für alle anderen Aufgaben sind die Zeitvorgaben weniger strikt. Werden sie nicht eingehalten, so hat dies zwar eine Verringerung der Servicequalität, aber keine schwerwiegenden Systemfehler zur Folge. Aus diesem Grund konzentriert sich dieser Abschnitt bei der Untersuchung des Realzeitverhaltens auf die Analyse des Cross-Coupled Controllers. Anhand dessen können auch die Vorteile des Concurrency-Control Protokolls PRED-DF noch einmal dargestellt werden. Ganz analog verläuft auch der Realzeitnachweis für die Überwachung der Endschalter. Beim gesamten Nachweis der Realzeitfähigkeit wird davon ausgegangen, daß die Rechenzeit nach EDF-Scheduling zugeteilt wird. Bei der Blockierung eines hochprioren Task durch einen niederprioren Task werden die Deadlines entsprechend dem Priority-Inheritance Protokoll angehoben (siehe Abschnitt 5.2.1).

Zeitanforderungen

Die Ereignisströme für die Drehwinkelgeber ergeben sich aus den technischen Daten der Fräsmaschine:

maximaler Vorschub	200 mm/min
Drehwinkelgeber	100 Impulse/mm

Das heißt, im ungünstigsten Fall ergibt sich 1 Impuls in 3 ms. Dies entspricht den folgenden Ereignisströmen (falls im folgenden nicht anders vermerkt, sind alle Zeiten in der Zeiteinheit ms angegeben):

$$\mathrm{ES}^1 = \mathrm{ES}^2 = \{(0, 3)\}.$$

Für jedes Signal muß die Berechnung des Regelalgorithmus spätestens bis zum nächsten Impuls des Drehwinkelgebers abgeschlossen sein (Deadline = Periodendauer), daraus folgt:

$$d_{\mathrm{ES}^1} = d_{\mathrm{ES}^2} = 3.$$

Auf Grund des langen zeitlichen Abstandes zwischen zwei Ereignissen, die von den Endschaltern der x-Achse ausgelöst werden können, kann für die beiden dazugehörigen Ereignissströme von

$$\mathrm{ES}^3 = \mathrm{ES}^4 = \{(0, \infty)\}$$

ausgegangen werden. Für die Deadlines bei der Behandlung der Endschalter muß, bei $\Delta s_{\max} = 0.5$ mm und $v = 200$ mm/min,

$$d_{\mathrm{ES}^3} = d_{\mathrm{ES}^4} = 150.$$

gefordert werden. Wenn davon ausgegangen wird, daß jedes Segment in Strecken von mindestens 0.02 mm zerlegt wird, liegt die Deadline d_1 für die Bereitstellung der Referenzkoordinaten bei 6 ms:

$$d_1 = 6.$$

Für die Visualisierung ist eine Deadline von

$$d_2 = 200$$

ausreichend.

Rechenzeiten

Für den zur Implementierung verwendeten Prozessor werden die folgenden Leistungsdaten zugrunde gelegt:

- RISC-CPU (z. B. ähnlich MIPS R4600),
- Taktfrequenz 50 MHz,
- 1.5 Cycles/Instruction.

Für die einzelnen Tasks und Transaktionen ergeben sich mit diesen Vorgaben die in Tabelle A.3 zusammengefaßten Annahmen für die Rechenzeiten. Die Summen in der Spalte „Befehle“ setzen sich dabei aus dem Rechenaufwnad für die Bearbeitung des Task- bzw. Transaktionsgerüstes (1. Summand) und aus der Befehlszahl, die für die Bearbeitung der eigentlichen Aufgabe der Transaktion oder des Tasks benötigt wird, zusammen (2. Summand).

Realzeitnachweis

Wesentlich für die Betrachtung des Realzeitverhaltens der Achsenregelung sind zum einen die Rechenzeiten der betroffenen Task und Transaktionen selbst und zum anderen die zusätzlich mit einzubeziehenden Zeiten, die sich aus der Blockierung durch andere Tasks und Transaktionen ergeben, die eine größere Deadline als die Achsregelung besitzen und dieselben Ressourcen nutzen.

Da die Regelung der x- und der y-Achse mit derselben Deadline bearbeitet werden, ist zwischen diesen beiden Tasksystemen kein gegenseitiger Ausschluß zu berücksichtigen. Aus dem selben

T_i/τ_i, Phase	Befehle	$\hat{c}_i$ [μs]	Deadline
T_1	4000 + 500	135	d_{ES^1}
T_2	2500	75	d_{ES^1}
T_2'	1500 + 500	60	d_{ES^1}
T_6	4000 + 500	135	d_{ES^2}
T_7	2500	75	d_{ES^2}
T_7'	1500 + 500	60	d_{ES^2}
τ_1, rc	670	20	d_{ES^1}
τ_1, w	730 + 100	30	d_{ES^1}
τ_2, r	890 + 300	35	d_{ES^1}
τ_2, cw	460 + 100	25	d_{ES^1}
τ_3	800 + 400	45	d_{ES^1}
τ_4, rc	670 + 1500	65	d_{ES^1}
τ_4, w	730 + 100	30	d_{ES^1}
τ_5, r	890 + 100	30	d_{ES^1}
τ_5, cw	460	15	d_{ES^1}
τ_7, r	900 + 1000	60	$d_1 > d_{ES^{1/2}}$
τ_7, c	5000	150	$d_1 > d_{ES^{1/2}}$
τ_7, w	730 + 500	55	$d_1 > d_{ES^{1/2}}$
τ_8, r	900	30	$d_2 > d_{ES^{1/2}}$
τ_8, cw	460 + 30000	915	$d_2 > d_{ES^{1/2}}$
τ_9, rc	670	20	d_{ES^2}
τ_9, w	730 + 100	30	d_{ES^2}
τ_{10}, r	890 + 300	35	d_{ES^2}
τ_{10}, cw	460 + 100	25	d_{ES^2}
τ_{11}	800 + 400	45	d_{ES^2}
τ_{12}, rc	670 + 1500	65	d_{ES^2}
τ_{12}, w	730 + 100	30	d_{ES^2}
τ_{13}, r	890 + 100	30	d_{ES^2}
τ_{13}, cw	460	15	d_{ES^2}

Tabelle A.3.: Zusammenfassung der Annahmen über die maximalen Rechenzeiten $\hat{c}_i$ für die Tasks und Transaktionen der Fallstudie. Die Ausführungszeiten sind auf volle 5 μs gerundet und enthalten auch die Rechenzeiten des Betriebssystems (z. B. für Nachrichtenempfang und -versand).

Grund ist es ausreichend anzunehmen, daß durch jede Instanz von τ_7 und τ_8 höchstens einer der beiden Regelkreise blockiert werden kann. Gegenseitige Blockierung von Tasks innerhalb eines Tasksystems, das eine Achse regelt, ist nicht möglich, da alle Tasks mit derselben Deadline bearbeitet werden und $z_0^1 = z_0^2 = d_{\mathrm{ES}^1} = d_{\mathrm{ES}^2}$.

Für den Realzeitnachweis sind daher nur S_2, S_4, S_5 und S_7 zu berücksichtigen, d. h. hier muß damit gerechnet werden, daß die Berechnung der Stellwerte für die Achsen durch τ_7 und/oder durch τ_8 verzögert wird. Im Realzeitnachweis sind daher die Deadlines von τ_7 und τ_8 entsprechend Abschnitt 5.2.1 zu verkürzen.

In Abbildung A.5 sind die Rechenzeit-Anforderungsfunktionen $C_{a,\mathrm{ges}}(I)$ für die Achsregelung sowohl bei der Verwendung von PRED-DF als auch beim Einsatz von PRED als Concurrency-Control Protokoll eingezeichnet. Zusätzlich dargestellt sind einzelne Teilsummen der gesamten Rechenzeit-Anforderungsfunktion.

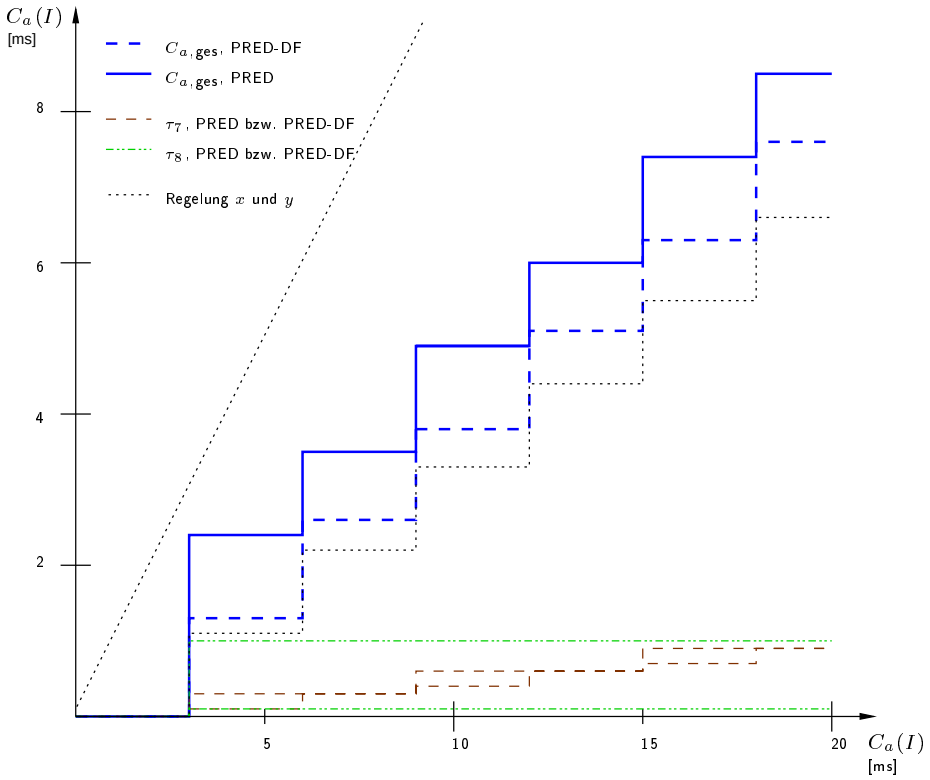


Abbildung A.5.: Rechenzeitanforderungsfunktion $C_{a,\mathrm{ges}}(I)$ für die Regelung der x- und y-Achse. Dargestellt ist $C_{a,\mathrm{ges}}(I)$ sowohl für die Verwendung von PRED-DF als auch für die Verwendung von PRED als Concurrency-Control Protokoll.

Die Rechenzeit-Anforderungsfunktion liegt im Fall von PRED-DF deutlich unter der von PRED, da hier für den gegenseitigen Ausschluß lediglich die im Verhältnis sehr kurzen Lese-Phasen einer Transaktion berücksichtigt werden müssen. PRED-DF fordert für einen erfolgreichen Realzeitnachweis somit deutlich geringere CPU-Ressourcen.

Würde zum Concurrency-Control ein Protokoll verwendet werden, das Transaktionskonflikte durch den Abbruch der Transaktion mit niedrigerer Priorität löst (z. B. 2PL-HP), wäre bei dem Beispiel dieser Fallstudie nicht sichergestellt, daß beispielsweise Transaktion τ_8 überhaupt terminiert, da sie bei jedem Neustart von einer Transaktion mit einer Deadline von 3 ms wieder abgebrochen werden kann [MF00]. Der Nachweis der Realzeitfähigkeit schlägt damit fehl.

A.2. Industrielle Anwendungsmöglichkeit des Datenbankmodells

Dieser Abschnitt stellt ein allgemeines Architekturmodell für die komponentenbasierte Implementierung von datendominierten Realzeitanwendungen vor [MBFW99][BMW00]. Ein zentrales Element dieser Architektur bildet eine aktive Realzeitdatenbank. Diese dient unter anderem dazu, eine universelle Möglichkeit zum Datentransfer zwischen den verschiedenen Komponenten einer Applikation zu realisieren. Das geschilderte Architekturmodell wird in Zukunft bei der Firma Rohde & Schwarz, München, die Basis der Softwareentwicklung für komplexe Meßgeräte bilden. Im folgenden wird knapp die Bedeutung der aktiven Realzeitdatenbank für die Kommunikation der Komponenten dargestellt.

Alle Arbeiten in diesem Umfeld wurden in enger Zusammenarbeit mit dem „Center of Competence für Softwarearchitekturen“ der Firma Rohde & Schwarz im Rahmen der durch die Bayerische Forschungsstiftung geförderten Forschungsverbünde FORSOFT I, Teilprojekt C1, und FORSOFT II, Teilprojekt HRS, durchgeführt.

A.2.1. Komponentensoftware

Das Konzept der komponentenorientierten Softwareentwicklung gewinnt immer mehr an Bedeutung und kann als logische Fortsetzung bzw. Erweiterung des objektorientierten Programmierens verstanden werden [Szy97]. Letztendliches Ziel ist es, Anwendungsprogramme ohne zusätzliche Codeentwicklung aus Komponenten, d. h. aus vorgefertigten, in sich abgeschlossenen binären Softwarebausteinen[1] mit definierter Aufgabe zusammenzusetzen („LEGO™-Prinzip“). Diese werden für die jeweilige Anwendung nur noch entsprechend konfiguriert. Das Zusammenspiel aller Komponenten stellt dann die gewünschte Funktionalität der Applikation zur Verfügung.

Die Vorteile des komponentenbasierten Softwareentwurfes sind:

- *Wiederverwendbarkeit.* Kürzere Entwicklungszeiten, geringere Kosten, geringerer Testaufwand; d. h. 'Time-To-Market' verkürzt sich.
- *Fehlervermeidung durch weniger Neucodieren.* Das heißt höhere Softwarequalität, Zuverlässigkeit, Sicherheit und Performance durch ausgereifte Software.
- *Vereinfachung der 'Make-Or-Buy'-Entscheidung.*

Ein konsequent komponentenbasierter Entwicklungsprozeß kann daher wesentlich dazu beitragen, die Systementwicklungszeiten zu verkürzen und qualitativ bessere Software zu geringeren Kosten bereitzustellen.

Augenblicklich existiert eine Vielzahl unterschiedlicher Komponententechnologien. Zum Beispiel: COM[Mic97], CORBA[Obj96] und JavaBeans/JEB [Sun97]. Der problematische Punkt bei der Entwicklung von Komponentensoftware bleibt aber stets die Kommunikation der einzelnen Komponenten untereinander. Diese findet zwar über einen normierten, allgemeingültigen Schnittstellenmechanismus statt, da nur so beliebige, unterschiedliche Komponenten (bzgl. Herkunft, Programmiersprache, Version etc.) miteinander kommunizieren und kooperieren können. Bei allen gegenwärtigen Lösungen ist es jedoch nicht ohne weiteres möglich, daß zwei unterschiedliche Komponenten, die ohne Berücksichtigung der jeweils anderen Komponente erstellt wurden, direkt Daten untereinander austauschen können. Für einen Methodenaufruf mit Datenaustausch müssen die genauen Übergabeparameter und Rückgabewerte schon beim Erstellen der

[1] Das heißt, die Bausteine liegen bereits vollständig fertig compiliert vor.

beteiligten Komponenten bekannt sein (z. B. hinsichtlich Typ, Anzahl, Reihenfolge, Strukturierung der Parameter). Der ebenfalls mögliche Datenaustausch über sogenannte Interface-Objekte weist dieselben Probleme auf, da die exakte Schnittstellendefinition bereits zum Zeitpunkt der Programmerstellung vorliegen muß. Um das Zusammenspiel der Komponenten zu ermöglichen, ist daher also auch bei der Softwareentwicklung innerhalb einer eng begrenzten Anwendungsdomäne in der Regel stets noch zusätzlicher Programmcode notwendig (meist als „Glue-Code" bezeichnet). Dort müssen vom Programmierer der Programmablauf und insbesondere der Datenaustausch explizit festgelegt werden.

Ein ideales Bausteinsystem, das die Systementwicklung nur durch das Zusammenstellen und Konfigurieren von Komponenten zuläßt, ist so noch nicht vollständig realisierbar. Die großen Vorteile, die der komponentenbasierte Entwurf, insbesondere in großen Projekten mit wiederkehrendem, ähnlichem Funktionsumfang bietet kann (hohe Softwarequalität bei geringen Entwicklungszeiten, hohe Reuse-Quote etc.), kommen daher augenblicklich der Softwareentwicklung nur eingeschränkt zugute.

A.2.2. Architekturmodell

Ein wichtiges Problem, das der Realisierung eines solchen idealen Baukastensystems im Wege steht, ist die *verknüpfte* Übertragung von normierbarem Kontroll- und nicht normierbarem Datenfluß über eine gemeinsame Schnittstelle.

Hier wird daher ein Architekturmodell vorgestellt, das den *getrennten* Austausch von Kontrollinformationen und Datenfluß realisiert:

- Der Austausch des *Kontrollflusses* erfolgt über direkten Methodenaufruf. Er wird auf eine relativ geringe Anzahl absolut notwendiger Kommandos begrenzt (z. B. start component, stop component, get data, place data). Durch die Normierung ist es möglich, daß Komponenten unmittelbar Kommandos austauschen können. Zur Übertragung werden die Möglichkeiten der unterlegten Komponentenarchitektur genutzt. Notwendig ist hierzu, daß die Komponenteninfrastruktur den transparenten Zugriff auf unterschiedliche Komponenten vorsieht und die Definition spezifischer, nach Aufgabenstellung und Rolle getrennter Schnittstellen zuläßt, z. B. COM/DCOM.

- Die Übertragung des *Datenflusses* wird über eine aktive Realzeitdatenbank mit konfigurierbarem Interfacelayer abgewickelt (als Zustandsdatenbasis bezeichnet). Sie enthält alle Informationen d. h. Daten über den augenblicklichen Zustand einer Anwendung, die von unterschiedlichen Komponenten genutzt werden. Die Datenbank verallgemeinert die Kommunikationsmechanismen, die von den Komponententechnologien zur Verfügung gestellt werden. Sie stellt dazu jeder Komponente die Sicht auf die Daten in der Zustandsdatenbasis zur Verfügung, die sie fordert und übernimmt dabei gleichzeitg den automatischen Abgleich dieser unterschiedlichen Datensichten. Zusätzlich verfügt die Zustandsdatenbasis über die Möglichkeit, Komponenten über bestimmte Datenbankzustände zu benachrichtigen. Die Zustandsdatenbasis selbst ist wiederum als eine Komponente realisiert.

Eine Prinzipskizze zum Aufbau der Architektur zeigt Bild A.6. Dargestellt ist die Infrastruktur der Komponententechnologie, der direkte Austausch von Kontrollinformationen zwischen den Komponenten und der davon getrennte Austausch von Daten über die Zustandsdatenbasis.

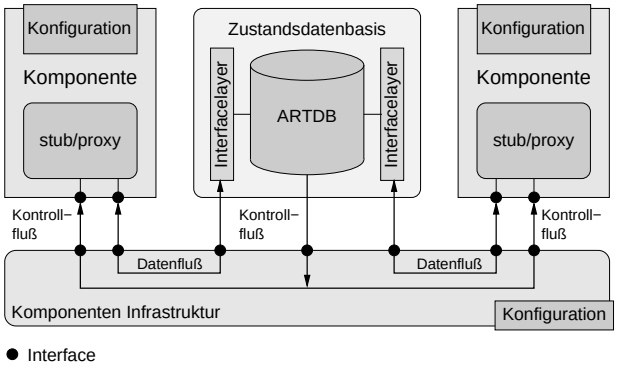


Abbildung A.6.: Architekturmodell: Abgebildet ist die getrennte Übertragung von Kontroll- und Datenfluß. Die Übertragung des Datenflusses erfolgt über die Zustandsdatenbasis.

A.2.3. Datenfluß

Abbildung A.7 a) zeigt den allgemeinen Ablauf eines Datentransfers zwischen zwei Komponenten. Das zu übertragende Datum `D` wird durch die sendende Komponente in die Datenbank eingetragen. Welcher Datensatz dazu angesprochen wird, legt der Konfigurationsdatensatz dieser Komponente fest. Die empfangende Komponente greift zu einem späteren Zeitpunkt auf dieses Datum zu und entnimmt den Wert der Datenbank. Die Konfiguration der Komponente bestimmt, aus welchem Datensatz gelesen wird. Existieren dabei zwei unterschiedliche Sichten `D` und $\mathtt{D}'$ auf die Information, die in `D` gespeichert ist, so erfolgt die Angleichung $\mathtt{D} \rightarrow \mathtt{D}'$ automatisch durch die aktive Datenbank.

Mittels der aktiven Funktionalität der Datenbank kann über die Zustandsdatenbasis auch asynchroner Datentransfer stattfinden oder eine Komponente über einen spezifischen Datenbankzustand informiert werden. In Abbildung A.7 b) ist der Abblauf des asynchronen Datenaustausches abgebildet: Komponente 1 setzt ein Datum in der Datenbank. Die Zustandsdatenbasis erkennt das Setzen des geänderten Parameterwertes und löst entsprechend ihrer Konfiguration das entsprechende Ereignis aus, d. h. Komponente 2 wird von der Veränderung des Datenbestandes informiert. Durch die Entnahme des geänderten Datenwertes durch Komponente 2 aus der Datenbank wird die Datenübertragung beendet.

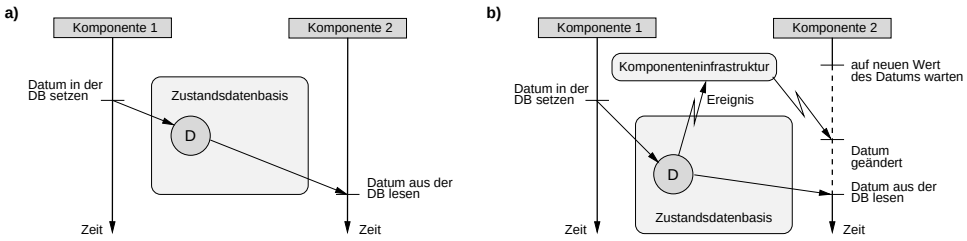


Abbildung A.7.: a) Prinzipieller Ablauf des Datentransfers zwischen zwei Komponenten über die Zustandsdatenbasis. b) Asynchroner Datentransfer mit blockierendem Empfangen über die aktive Realzeitdatenbank.

Die Benachrichtigung einer Komponente über einen spezifischen Datenbankzustand funktioniert prinzipiell ähnlich, verwendet aber zusätzlich noch die Möglichkeit, Bedingungen in der Datenbank zu definieren.

Bei der Anwendungserstellung entfällt bei diesem Architekturmodell im Idealfall die Entwicklung

von neuem Programmcode vollständig. Notwendig ist nur noch die Konfiguration der Komponenten und die Konfiguration der Zustandsdatenbasis. Komponenten können ohne zusätzlichen Programmieraufwand hinzugefügt und wieder entfernt werden. Einfache Erweiterbarkeit ist so sichergestellt und eine Anwendung kann leicht an neue Problemstellungen angepaßt werden. Die Realisierung eines Baukastensystems ist somit möglich.

B. Anhang: Theorie der Transaktionsverarbeitung

B.1. Serialisierbarkeit

Im Kontext dieser Arbeit wird unter Serialisierbarkeit stets *Konflikt-Serialisierbarkeit* verstanden (siehe hierzu z. B. [VGH93]). Ein Schedule ist konflikt-serialisierbar, falls er konfliktäquivalent zu einem Schedule mit serieller Ausführung der beteiligten Transaktionen ist. Das heißt, er kann durch eine Folge von Vertauschungen von sich nicht im Konflikt befindlichen Datenbankoperationen in einen seriellen Schedule transformiert werden. Zwei Operationen aus unterschiedlichen Transaktion befinden sich im Konflikt, falls sie auf dasselbe Datenobjekt zugreifen und mindestens eine der beiden Operationen eine Schreib-Operation ist.

Der Nachweis von Konflikt-Serialisierbarkeit ist mittels des gerichteten Konflikt-Graphen $G(s)$ möglich. $G(s)$ ist dabei folgendermaßen definiert:

Definition 15 (Konflikt-Graph) *Der Konflikt-Grpah $G(s) := (V, E)$ eines Schedules s ist durch die folgenden Mengen festgelegt:*

- *V ist die Menge der Knoten und enthält alle in s bereits freigegebenen Transaktionen,*
- *E ist die Menge der Kanten. Zwischen zwei Transaktionen τ_1, τ_2 in s gibt es eine Kante $\tau_1 \rightarrow \tau_2$, fallls τ_1 eine Operation, die sich mit einer Operation in τ_2 in Konflikt befindet, in s vor τ_2 ausführt.*

Es gilt:

Satz 1 (Konflikt-Serialisierbarkeit) *Ein Schedule s ist genau dann konflikt-serialisierbar, falls $G(s)$ nicht zyklisch. Das heißt, es gilt:*

$$s \text{ konfliktserialiserbar} \Longleftrightarrow G(s) \text{ azyklisch} .$$

B.2. Deadlocks

Ein System von Transaktionen befindet sich im Deadlock, falls jede Transaktion τ_i im System darauf wartet, daß eine andere Transaktion τ_j eine Ressource freigibt, die von τ_i zur Weiterarbeit benötigt wird (siehe hierzu zum Beispiel [CES71][VGH93]).

Deadlocks können mittels des gerichteten Wait-Graphen W beschrieben werden. Dieser ist wie folgt definiert:

Definition 16 (Wait-Graph) *Der Wait-Graph $W := W(V, E)$ ist durch die folgenden Mengen definiert:*

- *V ist die Menge der Knoten und enthält alle im System aktiven Transaktionen.*
- *E ist die Menge der Kanten. Eine Kante $\tau_1 \rightarrow \tau_2$ bedeutet, daß τ_1 darauf wartet, daß τ_2 eine von τ_1 benötigte Ressource freigibt.*

Ein Zyklus in W bedeutet, daß im Transaktionssystem ein Deadlock vorliegt.

C. Anhang: Graphentheorie

C.1. Bestimmung von Zusammenhangkomponenten

Im folgenden wird der Pseudocode eines Algorithmus zur Bestimmung der Zusammenhangkomponenten eines Graphen angegeben (siehe hierzu zum Beispiel [Tur96]). Er basiert auf einem Algorithmus zur Tiefensuche. Dabei werden – ausgehend von einer Startecke – stets alle Ecken des Graphen durchlaufen, die von dieser Ecke aus erreichbar sind, also eine Zusammenhangkomponente. Dabei wird für jede Zusammenhangkomponente ein Nummer vergeben, mit der die entsprechenden Ecken gekennzeichnet werden.

```
var
    ZKNummer: array[1 ... max] of Integer;
    EZähler: Integer;

procedure zusammenhangkomponente(G: Graph);
var
    i: Integer;
begin
    Initialisiere ZKNummer und EZähler mit 0;
    for jede Ecke i do
        if ZKNummer[i] = 0 then begin
            EZähler := EZähler + 1;
            zusammenhang(i);
        end
end

procedure zusammenhang(i: Integer);
var
    j: Integer;
begin
    ZKNummer[i] := EZähler;
    for jeden Nachbar j von i do
        if ZKNummer [j] = 0 then
            zusammenhang(j);
end
```

C.2. Bestimmung von Zyklen

Ein Algorithmus zur Kreissuche kann aus einem rekursiven Algorithmus zur Tiefensuche abgeleitet werden (siehe hierzu auch [Tur96]). Der folgende Algorithmus prüft für eine Ausgangsecke i eines zusammenhängenden Graphen, ob der Graph einen Zyklus enthält, d. h. ob er eine Rückwärtskante enthält, die im aktuellen „Ast“ der Rekursion liegt (verlassen = false). Die während der Tiefensuche besuchten Ecken werden aufsteigend numeriert (EZähler). Wird ein Zyklus gefunden, können anhand der Numerierung TSNummer die am Zyklus beteiligten Ecken rückwärts, ausgehend von der letzten betrachteten Ecke, ermittelt werden.

```
var
        verlassen: array[1 ... max] of Boolean;
        TSNummer: array[1 ... max] of Integer;
        EZähler: Integer;

procedure kreisfrei(G: Graph);
var
        i: Interger;
begin
        Initialisiere besucht mit false;
        Initialisiere TSNummer und EZähler mit 0;
        for jede Ecke i do
                if Besucht[i] = false then
                        kreissuchen(i);
        exit('kreisfrei');
end

procedure kreissuche(i: Integer);
var
        j: Integer;
begin
        EZähler := EZähler + 1;
        TSNummer[i] = EZähler;
        for jeden Nachbarn j von i do
                if TSNummer[j] = 0 then
                        kreissuche(j)
                else
                        if verlassen[j] = false then
                        begin
                                Ermittle Zyklusmitglieder aus TSNummer;
                                exit('Nicht kreisfrei');
                        end
        verlassen[i] = true;
end
```

Literaturverzeichnis

[ABRW93] Audsley, N. C. ; Burns, A. ; Richardson, M. F. ; Wellings, A. J.: Data Consistency in Hard Real-Time Systems / Univ. of York, Dep. of Computer Science. 1993 (YCS 203). – Forschungsbericht

[ACL91] Agrawal, D. ; Cochrane, R. J. ; Lindsay, B.: On Maintaining Priorities in a Production Rule System. In: *Proceedings of the 17th Conference on Very Large Databases* (1991), S. 479–487

[AGM89] Abbott, R. K. ; Garcia-Molina, H.: Scheduling Real-Time Transactions with Disk Resident Data. In: *Proceedings of the 15th International Conference on Very Large Data Bases* (1989), S. 385–396

[AGM92] Abbott, R. K. ; Garcia-Molina, H.: Scheduling Real-Time Transactions: A Performance Evaluation. In: *ACM Transactions on Database Systems* 17 (1992), Nr. 3, S. 513–560

[AHE+96] Andler, S. ; Hansson, J. ; Eriksson, J. ; Mellin, J. ; Berendtsson, M. ; Eftring, B.: DeeDS Towards a Distributed and Active Real-Time Database System. In: *ACM SIGMOD Record* 25 (1996), Nr. 1, S. 38–40

[AKGM96] Adelberg, B. ; Kao, B. ; Garcia-Molina, H.: Overview of the STanford Real-Time Information Processor (STRIP). In: *ACM SIGMOD Record* 25 (1996), Nr. 1, S. 34–37

[AWH92] Aiken, A. ; Widom, J. ; Hellerstein, J.: Behaviour of Database Production Rules: Termination, Confluence and Observable Determinism. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1992), S. 59–68

[BBDW97] Byun, J. ; Burns, A. ; Davis, R. ; Wellings, A.: A Worst-Case Behaviour Analysis for Hard Real-Time Transactions. In: Bestavros, A. (Hrsg.): *Real-Time Database Systems, Issues and Applications.* Kluwer, 1997, S. 235–249

[BBKZ93] Branding, H. ; Buchmann, A. ; Kudrass, T. ; Zimmermann, J.: Rules in an Open System: The REACH Rule System. In: Paton, N. W. (Hrsg.) ; Williams, M. H. (Hrsg.): *Rules in Database Systems.* Springer, 1993, S. 111–126

[BEHR82] Bayer, R. ; Ehlhardt, K. ; Heigert, J. ; Reiser, A.: Dynamic Timestamp Allocation for Transactions in Database Systems. In: *Proceedings of 2nd International Symposium on Distributed Databases* (1982), S. 9–20

[BFKM85] Brownstone, L. ; Farrell, R. ; Kant, E. ; Martin, N.: *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison Wesley, 1985

[BFW97] Bestavros, A. (Hrsg.) ; Fay-Wolfe, V. (Hrsg.): *Real-Time Database and Information Systems: Research Advances.* Kluwer Academic, 1997

[BHR80] Bayer, R. ; Heller, H. ; Reiser, A.: Parallelism and Recovery in Database Systems. In: *ACM Transactions on Database Systems* 5 (1980), Nr. 2, S. 139–156

[Blö00] Blösch, J. *Entwicklung einer generischen Test-Bench basierend auf einer aktiven Realzeitdatenbank.* Diplomarbeit, Technische Universität München. 2000

[Blo99] Bloos, M. F. *Echtzeitanalyse der Kommunikation in Kfz-Bordnetzen auf Basis des CAN-Protokolls.* Diss., Technische Universität München. 1999

[BLS97] Bestavros, A. (Hrsg.) ; Lin, K.-J. (Hrsg.) ; Son, S. (Hrsg.): *Real-Time Database Systems: Issues and Applications.* Kluwer Academic, 1997

[BMW00] Birkhold, M. ; Münnich, A. ; Woitschach, P.: Eine datenbankbasierte Architektur für Komponentensoftware in eingebetteten Realzeitsystemen. In: *Informationstechnik und Technische Informatik, it+ti* 42 (2000), Nr. 3, S. 40–48

[BSR80] Bernstein, P. A. ; Shipman, D. W. ; Rothnie, J. B.: Concurrency Control in a System for Distributed Databases (SDD-1). In: *ACM Transactions on Database Systems* 5 (1980), Nr. 1, S. 18–51

[CBB+89] Chakravarthy, S. ; Blaustein, B. ; Buchmann, A. P. ; Carey, M. ; Dayal, U. ; Goldhirsch, D. ; Hsu, M. ; Jauhari, R. ; Livny, M. ; McCarthy, D. ; McKee, R. ; Rosenthal, A.: HiPAC: A Resaerch Project in Active Time-Constrained Database Management / Xerox Advanced Information Technology. 1989 (XAIT-89-02, Reference Number 187). – Forschungsbericht

[CBW94] Chapman, R. ; Burns, A. ; Wellings, A.: Integrated Program Proof and Worst-Case Timing of SPARC Ada. In: *Proceedings of the ACM SIGPLAN Language, Compiler, and Tool Support for Real-Time Systems (LCTS) Workshop* (1994)

[CES71] Coffman, E. G. ; Elphick, M. J. ; Soshani, A.: System Deadlocks. In: *Computing Surveys* 3 (1971), Nr. 2, S. 67–78

[Day99] Day, N. *Avoiding the Need for Database Concurrency Mechanisms.* White Paper, Polyhedra PLC, http://www.polyhedra.com/concur.html. 1999

[DBM88] Dayal, U. ; Buchmann, A. ; McCarthy, D.: Rules are Objects too: A Knowledge Model for Active, Object-Oriented Database Systems. In: *Advances in Object-Oriented Database Systems* (1988), S. 129–143

[DG96] Dittrich, K. R. ; Gatziu, S.: *Aktive Datenbanksysteme, Konzepte und Mechanismen.* Thomson Publishing, 1996

[DGG96] Dittrich, K. R. ; Gatziu, S. ; Geppert, A.: The Active Database Management System Manifesto: A Rulebase of ADMS Features. In: *SIGMOD Record* 25 (1996), Nr. 3, S. 40–49

[DMK+96] Datta, A. ; Mukherjee, S. ; Konana, P. ; Viguier, I. R. ; Bajaj, A.: Multiclass Transaction Scheduling and Overload Management in Firm Real-Time Database Systems. In: *Information Systems* 21 (1996), Nr. 1, S. 29–54

[EGLT76] Eswaran, K. P. ; Gray, J. N. ; Lorie, R. A. ; Traiger, I. L.: The Notions of Consistency and Predicate Locks in a Database System. In: *Communications of the ACM* 19 (1976), Nr. 11, S. 624–633

[Eic87] Eich, M. H.: MARS: The Design of a Main Memory Database Machine. In: *Proceedings of the International Workshop an Database Machines* (1987), S. 325–338

[EL95] Eich, M. H. ; Li, X.: Logging in Main Memory Databases. In: *Australian Computer Science Communications* 17 (1995), Nr. 2, S. 93–102

[Eri97] Eriksson, J.: Real-Time and Active Database Systems: A Survey. In: *Lecture Notes in Computer Science 1553* (1997), S. 1–23

[EWY99] Ericsson, C. ; Wall, A. ; Yi, W.: Timed Automata as Task Models for Event-Driven Systems. In: *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications RTCSA* (1999)

[Fär92] Färber, G.: *Prozeßrechentechnik*. Springer Verlag, 1992

[GD92] Gatziu, S. ; Dittrich, K. R.: SAMOS: An Active Object-Oriented Database System. In: *IEEE Data Engineering, Special Issue an Active Databases* 15 (1992), S. 27–30

[GGD94] Geppert, A. ; Gatziu, S. ; Dittrich, K. R.: Performance Evaluation of an Active Database Management System: 007 Meets the BEAST / Institut für Informatik, Universität Zürich. 1994 (94.18). – Forschungsbericht

[GHS95] Gerber, R. ; Hong, S. ; Saksena, M.: Guaranteeing Real-Time Requirements with Ressource-Based Calibration of Periodic Processes. In: *IEEE Transactions on Software Engineering* 21 (1995), Nr. 7

[GJ91] Gehani, N. H. ; Jagadish, H. V.: Ode as an Active Database: Constraints and Triggers. In: *Proceedings of the 17th International Conference on Very Large Data Bases* (1991), S. 327–336

[GJS92] Gehani, N. H. ; Jagadish, H. V. ; Shmueli, O: Composite Event Specification in Active Databases: Model and Implementation. In: *Proceedings of the 18th International Conference on Very Large Data Bases* (1992), S. 327–338

[GMS92] Garcia-Molina, H. ; Salem, K.: Main-Memory Database Systems: An Overview. In: *IEEE Transactions on Knowledge and Data Engeneering* 4 (1992), Nr. 6, S. 509–516

[GR93] Gray, J. ; Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, 1993

[Gre93a] Gresser, K.: An Event Model for Deadline Verification of Hard Real-Time Systems. In: *IEEE Proceedings of 5th Euromicro Workshop on Real-Time Systems*, 1993, S. 118–123

[Gre93b] Gresser, K. *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme*. Diss., VDI Verlag, Düsseldorf. 1993

[Hag87] Hagmann, R. B.: A Crash Recovery Scheme for Memory-Resident Database Systems. In: *IEEE Transactions on Computers C* 35 (1987), Nr. 9, S. 839–843

[HB99] Hansson, J. ; Berendtsson, M.: Active Real-Time Database Systems. In: Paton, N. W. (Hrsg.): *Active Rules in Database Systems*. Springer, 1999, S. 405–423

[HW91] Hanson, E. N. ; Widom, J.: Rule Processing in Active Database Systems /

Department of Computer Science and Engineering, Wright State University. 1991 (WSU-CS-01-07). – Forschungsbericht

[KB91] Kumar, V. ; Burger, A.: Performance Measurement of Some Main Memory Database Recovery Algorithms. In: *IEEE Transactions on Knowledge and Data Engineering* (1991), S. 436–443

[Kim99] Kim, Y.-K.: *Predictability and Consistency in Real-Time Transaction Processing*, University of Virginia, Diss., 1999

[KLP+98] Kwak, H.-H. ; Lee, I. ; Philippou, A. ; Choi, J.-Y. ; Sokolsky, O.: Symbolic Schedulability Analysis of Real-Time Systems. In: *Proceedings of the 19th IEEE Real-Time Systems Symposium RTSS* (1998), December

[LC87] Lehmann, T. J. ; Carey, M. J.: A Recovery Algorithm for a High Performance Memory-Resident Database System. In: *Proceedings of the ACM SIMOD Conference* (1987), S. 104–117

[LHT00] Lindgren, M. ; Hansson, H. ; Thane, H.: Using Measurements to Derive the Worst-Case Execution Time. In: *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications RTCSA* (2000), S. 15–22

[Lin89] Lin, K. J.: Consistency Issues in Real-Time Database Systems. In: *Proceedings of Hawaii International Conference on System Sciences* (1989), S. 654–661

[LL73] Liu, C. L. ; Layland, J. W.: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. In: *Journal of the ACM* 20 (1973), Nr. 1, S. 46–61

[LL93] Le Gruenwald ; Liu, S.: A Performance Study of Concurrency Control in a Real-Time Main Memory Database System. In: *SIGMOD Record* 22 (1993), Nr. 4, S. 38–44

[Loc86] Locke, C. D.: *Best-Effort Decision Making for Real-Time Scheduling*, Department of Computer Science, Carnegie Mellon University, Diss., 1986

[Loc92] Locke, C. D.: Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Scheduling. In: *Journal of Real-Time Systems* 4 (1992), Nr. 1, S. 37–53

[Loc00] Locke, D.: Applications and System Characteristics. In: Lam, K.-Y. (Hrsg.) ; Kuo, T.-W. (Hrsg.): *Real-Time Database Systems.* Kluwer Academic, 2000, S. 17–26

[LR99] Lindström, J. ; Raatikainen, K.: Dynamic Adjustment of Serialization Order using Timestamp Intervals in Real-Time Databases. In: *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications RTCSA* (1999), S. 13–20

[LS92] Levy, E. ; Silberschatz, A.: Incremental Recovery in Main Memory Database Systems / University of Texas at Austin. 1992 (CS-TR-92-01). – Forschungsbericht

[LS96] Lee, J. ; Son, S. H.: Performance of Concurrency Control Algorithms for Real-Time Database Systems. In: *Performance of Concurrency Control Mechanisms in Centralized Database Systems* (1996), S. 429–460

[LW82] Leung, J. Y.-T. ; Whitehead, J.: On the Complexity of Fixed-Priority Scheduling

of Periodic Real-Time Tasks. In: *Performance Evaluation* 2 (1982), Nr. 4, S. 237–250

[MBFW99] Münnich, A. ; Birkhold, M. ; Färber, G. ; Woitschach, P.: Towards an Active Real-Time Database-Based Architecture for Real-Time Systems using Configurable, Standardized Components. In: *IEEE Proceedings of the International Database Engineering & Applications Symposium IDEAS*, 1999, S. 351–359

[Mel98] Mellin, J. *Predictable Event Monitoring.* Linköping Studies in Science and Technology, Thesis No 737. 1998

[MF00] Münnich, A. ; Färber, G.: Calculating Worst-Case Execution Times of Transactions in Databases for Event-Driven, Hard Real-Time Embedded Systems. In: *IEEE Proceedings of the International Database Engineering & Applications Symposium IDEAS*, 2000, S. 149–157

[Mic97] Microsoft Corporation: *The Component Object Model Specification.* 1997. – http://www.microsoft.com

[Mün00a] Münnich, A.: PRED-DF – A Data Flow Based Semantic Concurrency Control Protocol for Real-Time Main-Memory Database Systems. In: *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications RTCSA*, 2000, S. 468–472

[Mün00b] Münnich, A.: Testbed zur Evaluierung von Concurrency-Control Protokollen für Realzeitdatenbanken, Source-Code / Technische Universität München. 2000. – Forschungsbericht

[Mok83] Mok, A. K.-L.: *Fundamental Design Problems of Distributed Systems for Hard-Real-Time Environment*, Massachusetts Institute of Technology, Diss., 1983

[Mor83] Morgenstern, M.: Active Databases as a Paradigm for Enhanced Computing Environments. In: *Proceedings of the 9th International Conference on Very Large Data Bases* (1983), S. 34–42

[Mos81] Moss, J. E. B.: Nested Transactions: An Approach to Reliable Distributed Computing / MIT Reports. 1981 (LCS TR 260). – Forschungsbericht

[Obj96] Object Management Group, OMG: *The Common Object Request Broker: Architecture and Specification.* 1996. – http://ww.omg.org

[Pat98] Paton, N. W. (Hrsg.): *Active Rules in Database Systems.* Springer, Monographs in Computer Science, 1998

[Pet00] Petters, S.: Bounding the Execution Time of Real-Time Tasks on Modern Processors. In: *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications RTCSA*, 2000, S. 498–502

[PFW97] Prichard, J. J. ; Fortier, P. J. ; Wolfe, V. F.: RTSQL: Extending the SQL2 Standards to Support Real-Time Databases. In: Bestavros, A. (Hrsg.) ; Fay-Wolfe, V. (Hrsg.): *Real-Time Database and Information Systems: Research Advances.* Kluwer Academic, 1997, S. 289–310

[PL92] Pu, C. ; Leff, A.: Autonomous Transaction Execution with Epsilon Serializability. In: *Proceedings of 2nd International Workshop on Research Issues on Data Engineering: Transactions and Query Processing* (1992), S. 2–11

[PMK+00] Petters, S. ; Muth, A. ; Kolloch, T. ; Hopfner, T. ; Fischer, F. ; Färber, G.: The REAR Framework for Emulation and Analysis of Embedded Hard Real–Time Systems. In: *Design Automation for Embedded Systems* 5 (2000), Nr. 3, S. 237–250

[PSS93] Purimetla, B. ; Sivasankaran, R. M. ; Stankovic, J. A.: A Study of Distributed Real-Time Active Database Applications. In: *Proceedings of the IEEE Workshop on Parallel and Distributed Real-Times Systems* (1993)

[Ram93] Ramamritham, K.: Real-Time Databases. In: *Journal of Distributed and Parallel Databases* 1 (1993), Nr. 2, S. 199–226

[RSI78] Rosenkrantz, D. J. ; Stearns, R. E. ; II, P. M. L.: System Level Concurrency Control for Distributed Database Systems. In: *Transactions on Database Systems* 3 (1978), Nr. 2, S. 178–198

[SB94] Spuri, M. ; Butazzo, G.: Efficient Aperiodic Service under Earliest Deadline Scheduling. In: *Proceedings of the IEEE Real-Time Systems Symposium* (1994), S. 2–11

[Sch99] Schwaiger, D. *Prototypische Realisierung eines Systems zum datenorientierten Entwurf einer Gerätezustandsdatenbasis.* Diplomarbeit, Technische Universität München. 1999

[SL95] Song, X. ; Liu, J. W.-S.: Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency-Control. In: *IEEE Transactions on Knowledge and Data Engineering 7* (1995), S. 786–796

[SPSR93] Sivasankaran, R. M. ; Purimetla, B. ; Stankovic, J. A. ; Ramamritham, K.: Network Services Database - A Distributed Active Real-Time Database (DARTDB) Application. In: *IEEE Workshop on Real-Time Applications* (1993)

[Spu95] Spuri, M.: Analysis of Deadline Scheduled Real-Time Systems / INRIA, Le Chesnay, France. 1995 (RR-2772). – Forschungsbericht

[SRH90] Stonebraker, M. ; Rowe, L. ; Hirohama, M.: The Implementation of POSTGRES. In: *IEEE Transactions on Knowledge and Data Engineering* 2 (1990), Nr. 7, S. 125–142

[SRL90] Sha, L. ; Rajkumar, R. ; Lehocky, J. P.: Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In: *IEEE Transactions on Computers* 39 (1990), Nr. 9, S. 1175–1185

[SRLR89] Sha, L. ; Rajkumar, R. ; Lehocky, J. P. ; Ramamritham, K.: Mode Change Protocols for Priority-Driven, Preemptive Scheduling. In: *Real-Time Systems Journal* (1989), S. 243–265

[SRS00] Sivasankaran, R. M. ; Ramamritham, K. ; Stankovic, J. A.: System Failure and Recovery. In: Lam, K.-Y. (Hrsg.) ; Kuo, T.-W. (Hrsg.): *Real-Time Database Systems.* Kluwer Academic, 2000, S. 109–124

[SSL89] Sprunt, B. ; Sha, L. ; Lehoczky, J.: Aperiodic Task Scheduling for Hard Real-Time Systems. In: *Journal of Real-Time Systems* (1989), Nr. 1, S. 27–60

[SSRB98] Stankovic, J. A. ; Spuri, M. ; Ramamritham, K. ; Buttazzo, G. C.: *Deadline Scheduling for Real-Time Systems, EDF and Related Algorithms.* Kluwer Academic, 1998

[Sta88] Stankovic, J. A.: Misconceptions About Real-Time Computing. In: *IEEE Computer* 21 (1988), Nr. 10, S. 10–19

[Sun97] Sun Microsystems. *Specification of JavaBeans API.* Sun Microsystems Inc., 2550 Garcia Avenue, Mountain 1997

[Szy97] Szyperski, C.: *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, 1997

[TBW94] Tindell, K. ; Burns, A. ; Wellings, A. J.: An Extensible Approach for Analysing Fixed Priority Hard Real-Time Tasks. In: *Real-Time Systems* 6 (1994), Nr. 2, S. 133–151

[TSYT97] Tayara, M. ; Soparkar, N. ; Yook, J. ; Tilbury, D.: Real-Time Data and Coordination Control for Reconfigurable Manufacturing Systems. In: *Real-Time Database and Information Systems, Research Advances.* Kluwer, 1997, S. 23–48

[Tur96] Turau, V. (Hrsg.): *Algorithmische Graphentheorie.* Addison-Wesley, 1996

[TVC00] Tomic, S. ; Vrbsky, S. V. ; Camp, T.: A New Measure of Temporal Consistency for Derived Objects in Real-Time Database Systems. In: *Information Sciences 124* (2000), S. 139–152

[UB92] Ulusoy, Ö. ; Belford, G. G.: Concurrency Control in Real-Time Database Systems. In: *ACM Conference on Computer Science* (1992), S. 181–188

[UB98] Ulusoy, Ö. ; Buchmann, A.: A Real-Time Concurrency Control Protocol for Main-Memory Database Systems. In: *Information Systems* 23 (1998), Nr. 2, S. 109–125

[VGH93] Vossen, G. ; Gross-Hardt, M.: *Grundlagen der Transaktionsverarbeitung.* Addison-Wesley, 1993

[WF92] Widom, J. ; Finkelstein, S. J.: The Starburst Rule System: Language, Design, Implementation and Applications. In: *IEEE Data Engineeering* 15 (1992), Nr. 1-4, S. 15–18

[WKO+84] Witt, D. J. D. ; Katz, R. H. ; Olken, F. ; Shapiro, L. D. ; Stonebraker, M. R. ; Wood, D.: Implementation Techniques for Main-Memory Database Systems. In: *Proceedings of the SIGMOD Conference* (1984), S. 1–8

[Wro97] Wrobel, H.: *Implementation of Deadline Scheduling and Deadline Inheritance into RTEMS and Port to the MC 68332 Based NF 300 Board*, Technische Universität München, Diplomarbeit, 1997

[WYP92] Wu, K. L. ; Yu, P. S. ; Pu, C.: Divergence Control for Epsilon-Serializability. In: Golshani, F. (Hrsg.): *Proceedings of the 8th International Conference on Data Engineering*, 1992, S. 506–515

[XR99] Xiong, M. ; Ramamritham, K.: Deriving Deadlines and Periods for Real-Time Update Transactions. In: *Proceedings of IEEE Real-Time Systems Symposium* (1999), S. 1–8

[YTCS98] Yook, J. ; Tilbury, D. ; Chervela, K. ; Soparkar, N.: Decentralized, Modular Real-Time Control for Machining Applications. In: *Proceedings of the American Control Conference* (1998), S. 844–849

[YWLS94] Yu, P. S. ; Wu, K.-L. ; Lin, K.-J. ; Son, S. H.: On Real-Time Databases: Concurrency Control and Scheduling. In: *Proceedings of the IEEE* 82 (1994), Nr. 1, S. 140–157

Danke

An dieser Stelle möchte ich mich bei allen bedanken, die zum Gelingen dieser Arbeit beigetragen haben, vor allem:

Herrn Prof. Färber für die Möglichkeit, diese Arbeit am Lehrstuhl für Realzeit-Computersysteme durchführen zu können, seine Betreuung und seine Unterstützung,

Herrn Prof. Broy für die Übernahme des Zweitgutachtens und die kritische Durchsicht einer Vorversion dieser Arbeit,

allen Mitarbeitern und Kollegen am Lehrstuhl und im Projekt FORSOFT, sowie den Mitarbeitern der Industriepartner, insbesondere Herrn Woitschach, für die stets angenehme Arbeitsatmosphäre.

Einen besonderen Dank möchte ich meinen Eltern aussprechen, die mir durch ihre stets uneingeschränkte Unterstützung in vieler Hinsicht meine Ausbildung und diese Dissertation erst ermöglicht und mich stets mit vollem Vertrauen gefördert haben. Ebenso möchte ich Sonja für ihre Geduld und die liebevoll-kritische Begleitung meiner Arbeit danken.