

Integritätszentrierter Datenbank-Entwurf

Udo W. Lipeck

Institut für Informatik, Universität Hannover
Lange Laube 22, D-W3000 Hannover 1
ul@informatik.uni-hannover.de

1 Zielsetzung

In diesem Beitrag wird die zentrale, phasenübergreifende Rolle verfolgt, die die Berücksichtigung von (semantischer) Integrität beim Entwurf von zuverlässigen Datenbank-Anwendungssystemen spielt bzw. spielen sollte. Während unter einem Datenbanksystem der Komplex aus anwendungsunabhängiger DB-Software und anwendungsspezifischem Datenbestand verstanden wird, schließt der Begriff Datenbank-Anwendungssystem (DBAS) auch Grundbausteine für Anwendungsprogramme wie vorbereitete Transaktionen, Trigger und Prozeduren ein. Die folgenden Abschnitte sollen zeigen, daß es beim DB-Entwurf neben der Modellierung von DB-Strukturen vor allem um die Aufstellung von Integritätsbedingungen (IBen) und deren Transformation in überwachbare Elemente geht.

2 Aufstellung von Integritätsbedingungen

Zunächst müssen beim (ersten) konzeptionellen Entwurf mittels eines semantischen Datenmodells solche Integritätsbedingungen explizit aufgestellt werden, die sich aus der Informationsbedarfsanalyse ergeben und sich nicht in Strukturen des Datenmodells umsetzen lassen.

Beispiel: Gegeben das Schema von Abbildung 1 in einem erweiterten Entity-Relationship-Modell (z.B. nach [EGHHLSE90]), das Kunden, Stammkunden mit eigenem Konto und Bestellungen von Waren beschreibt. Daß nur vorhandene Kunden vorhandene Waren bestellen dürfen, ist eine modellinhärente Eigenschaft des Relationship-Typs bestellt. Da das Modell Spezialisierungen anbietet, z.B. hier von Kunden zu Stammkunden, ist bereits ausgedrückt, daß die Menge aller Stammkunden eine Teilmenge aller Kunden bildet. Diese haben Konto als eigenes Attribut und erben die Kundenattribute KNr, KName, usw.

Hingegen muß die Restriktion, daß keine Bestellung von Stammkunden mit Kontostand unter -1000 zulässig ist, explizit angegeben werden (hier als prädikatenlogische Formel):

$$(1) \quad \forall s: \text{STAMMKUNDE } s.\text{Konto} < -1000 \Rightarrow \nexists w: \text{WARE } \text{bestellt}(s, w) \quad \square$$

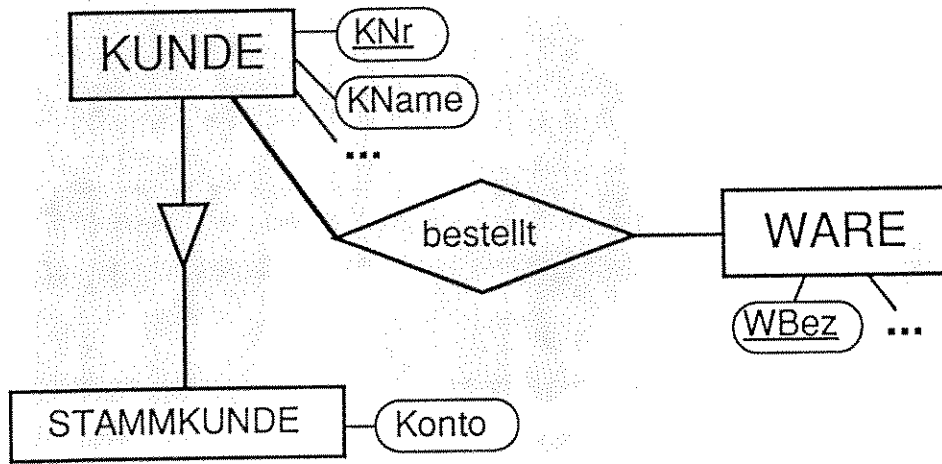


Abbildung 1:

In folgenden Entwurfsphasen, die typischerweise mit einem Modellwechsel einhergehen, etwa vom erweiterten Entity-Relationship-Modell über das einfache ER-Modell zum (implementierten) Relationenmodell, müssen infolge der Transformation von DB-Strukturen explizite IBen umformuliert oder neue explizite IBen zum Ausgleich für verlorengegangene strukturelle IBen ergänzt werden.

Beispiel: Die obige Spezialisierung von Kunden zu Stammkunden muß im Entity-Relationship-Modell durch einen n:1-Relationship-Typ ist dargestellt werden, und wenigstens die identifizierenden Attribute, hier KNr, müssen für den Teiltyp übernommen werden (siehe Abb. 2).

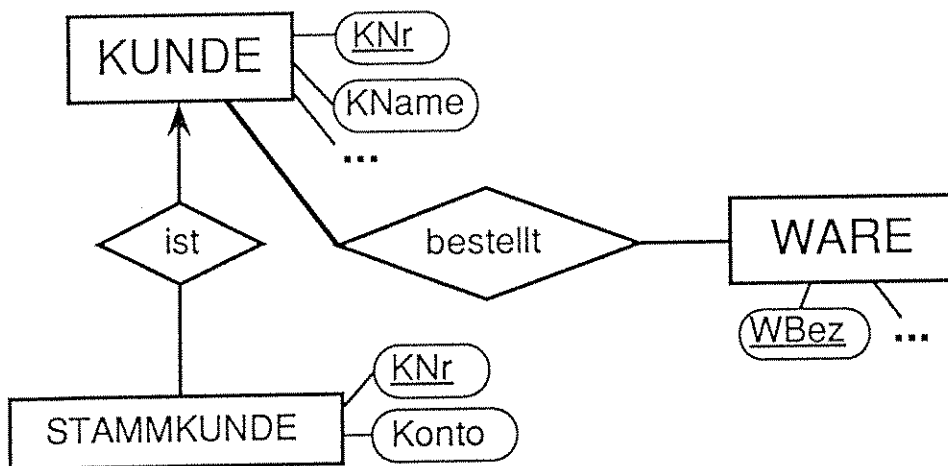


Abbildung 2:

Anstelle der inhärenten Teilmengenbeziehung ist nun explizit zu fordern, daß es zu jedem Entity vom Typ STAMMKUNDE ein Entity vom Typ KUNDE gibt, und daß nur Entities mit gleichem Schlüssel in Beziehung stehen.

- (2) $\forall s: \text{STAMMKUNDE} \exists k: \text{KUNDE} \text{ ist}(s, k)$
- (3) $\forall s: \text{STAMMKUNDE} \forall k: \text{KUNDE} \text{ ist}(s, k) \Rightarrow d.\text{KNr} = k.\text{KNr}$

Außerdem muß IB (1) auf die neuen Strukturen angepaßt werden:

$$(1)' \quad \forall s: \text{STAMMKUNDE} \forall k: \text{KUNDE} (s.\text{Konto} < -1000 \wedge \text{ist}(s, k)) \\ \Rightarrow \exists w: \text{WARE} \text{ bestellt}(k, w)$$

Beim weiteren Übergang zum Relationenmodell muß auch der Relationship-Typ bestellt durch eine Relation BESTELLUNG mit passenden Fremdschlüsselbedingungen (referentielle Integritätsbedingungen) aufgelöst werden:

$$\text{Relationenschemata: KUNDE}(\underline{\text{KNr}}, \text{KName}, \dots), \text{WARE}(\underline{\text{WBez}}, \dots), \\ \text{BESTELLUNG}(\underline{\text{KNr}} \rightarrow \text{KUNDE}, \underline{\text{WBez}} \rightarrow \text{WARE})$$

Für IB (1) ergibt sich folgende Umformulierung:

$$(1)'' \quad \forall d: \text{STAMMKUNDE} \forall k: \text{KUNDE} (d.\text{Konto} < -1000 \wedge d.\text{KNr} = k.\text{KNr}) \\ \Rightarrow \exists b: \text{BESTELLUNG} \ b.\text{KNr} = k.\text{KNr}$$

Wenn als Datenmodell nur noch pure Relationen zur Verfügung stehen (was auf manche "relationalen" DBMS durchaus fast zutrifft), sind auch noch die Schlüssel- und Fremdschlüsselbedingungen explizit zu machen, z.B.:

$$(4) \quad \forall k, k': \text{KUNDE} \ k.\text{KNr} \neq k'.\text{KNr} \Rightarrow k \neq k'$$

$$(5) \quad \forall b: \text{BESTELLUNG} \ \exists k: \text{KUNDE} \ \exists w: \text{WARE} \\ b.\text{KNr} = k.\text{KNr} \wedge b.\text{WBez} = w.\text{WBez} \quad \square$$

Abbildung 3 gibt einen Überblick über Erst- und Folgemodellierungen. Strukturtransformationen induzieren ganze Schematransformationen, die korrekt sind, wenn die durch DB-Strukturen und IBen beschriebenen Mengen von DB-Ausprägungen (Zuständen) zueinander äquivalent sind. Im Gegensatz zur Erstmodellierung, die relativ zur Anwendungswelt nur validiert werden kann, können Folgeschritte durch den Einsatz formaler Spezifikationen für IBen prinzipiell auch verifiziert werden.

Um auch das dynamische Verhalten von Datenbank-Anwendungssystemen in den Entwurf einzubeziehen, sollten neben den klassischen statischen IBen dynamische, d.h. transitionale oder temporale IBen aufgestellt werden, die Bedingungen an Zustandsübergänge bzw. an Folgen von Zuständen beinhalten. Diese lassen sich mit Hilfe transitionaler oder temporaler Operatoren ("old/new", "always", "sometime", "before", usw.) formalisieren oder direkt durch sogenannte Transitionsgraphen darstellen [LS88, Lip89, Saa91].

Beispiel: Die folgenden Integritätsbedingungen beziehen sich auf das ER-Schema in Abbildung 4 und verwenden die Variablen k: KUNDE, w: WARE, l: LIEFERANT, t: date und p: decimal(6, 2).

- Statische IB (Restriktion von DB-Zuständen):

$$\forall k, l, w \text{ BESTELLUNG}(k, l, w) \Rightarrow \text{PREISLISTE}(l, w)$$

(Zu jeder Bestellung gibt es einen passenden Eintrag in der Preisliste.)

- Transitionale IB (Restriktion von Zustandsübergängen):

$$\forall k, l, w, p \ \text{old} \ \text{BPreis}(k, l, w) = p \Rightarrow \text{new} \ \text{BPreis}(k, l, w) = p$$

(Der für eine Bestellung gültige Preis darf sich nicht ändern.)

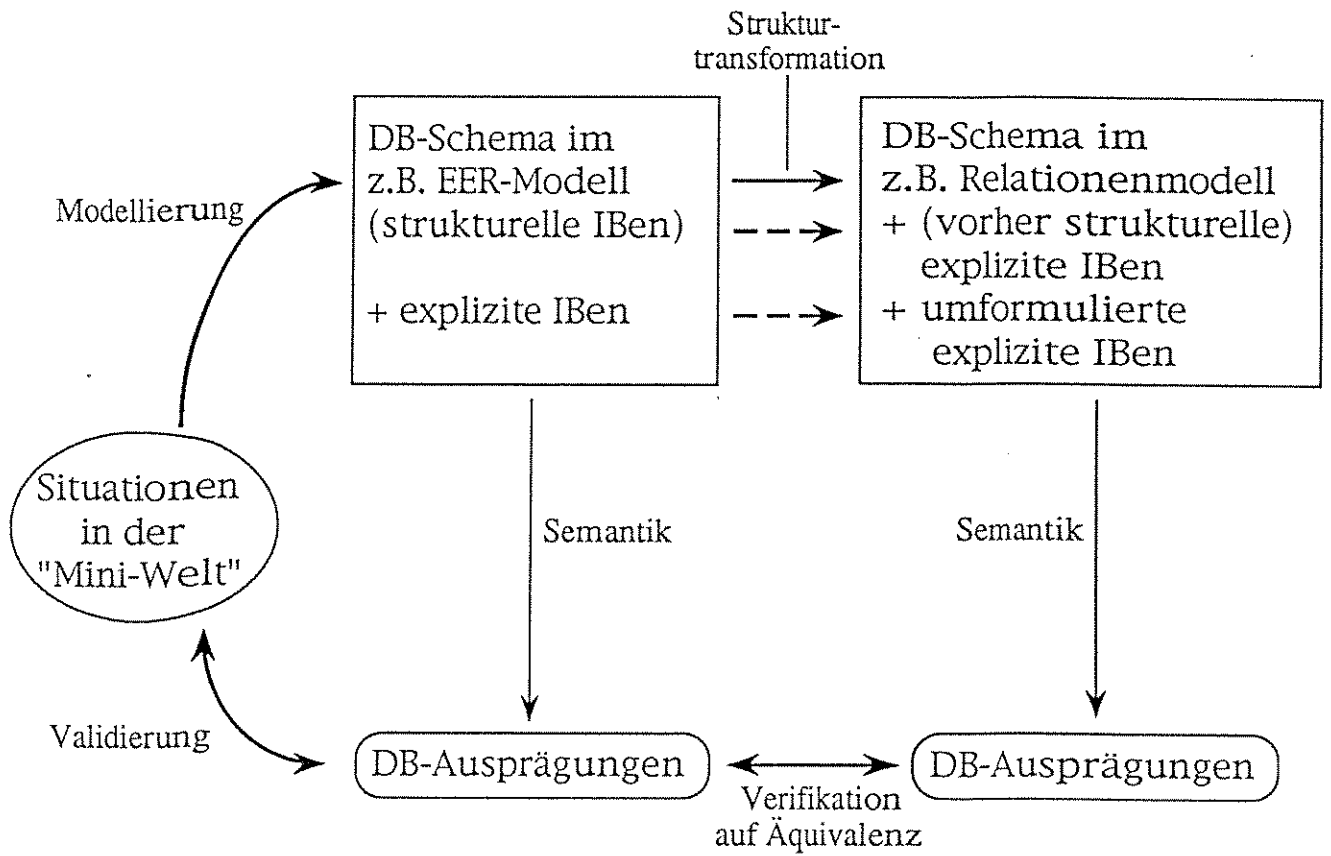


Abbildung 3:

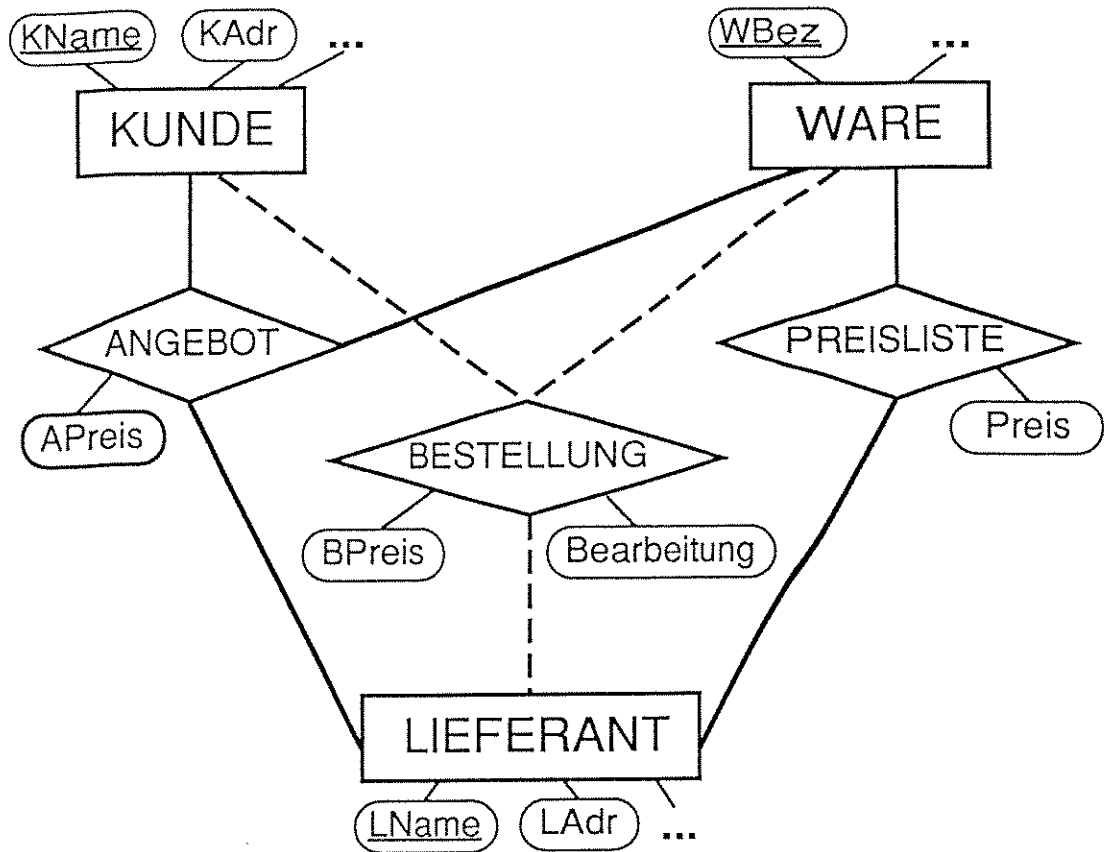


Abbildung 4:

- Temporale IB (Restriktion von Zustandsfolgen):

from ANGEBOT(k,l,w) **with** t := Heute

holds always ANGEBOT(k,l,w) **before** Heute > t + 30

(Wenn ein Angebot erstellt wird, bleibt es 30 Tage lang erhalten.)

Abbildung 5 enthält detailliertere temporale IBen, die Preise neuer Angebote durch die Preisliste begrenzen (A1), die Gültigkeit von Angeboten und Preisen präzisieren (B1), und Bindungen der für Bestellungen gültigen Preise angeben: zunächst wird der Preis aus dem Angebot, sonst aus der Preisliste übernommen (B1); solange dann die Bestellung läuft, bleibt der Bestellpreis unverändert (B2). Solche Anforderungen lassen sich nicht durch statische oder transitionale IBen ausdrücken.

A1: always-from ANGEBOT(k,l,w) holds
PREISLISTE(l,w)
and APreis(k,l,w) ≤ Preis(l,w)

A2: always-from ANGEBOT(k,l,w)
with t:= Heute, p:= APreis(k,l,w) **holds**
always ANGEBOT(k,l,w) **and** APreis = p **before** Heute > t+30
and from Heute > t+30 **holds not** ANGEBOT(k,l,w)

B1: always-from BESTELLUNG(k,l,w) holds
if ANGEBOT(k,l,w) **then** BPreis(k,l,w) = APreis(k,l,w)
else PREISLISTE(l,w) **and** BPreis(k,l,w) = Preis(l,w)

B2: always-from BESTELLUNG(k,l,w)
with p:= BPreis **holds**
BPreis = p **while** BESTELLUNG(k,l,w)

Abbildung 5:

Aus solchen temporalen Formeln können systematisch Transitionsgraphen konstruiert werden [LS87, Lip89], die Lebensläufe von DB-Objekten bzgl. der IBen beschreiben: die Knoten entsprechen Situationen in diesen Lebensläufen, die Kanten beinhalten (wechselnde) Bedingungen an DB-Zustände. Die Transitionsgraphen zu den obigen IBen sind in Abbildung 6 gezeigt.

Während dieses Vorgehen erlaubt, IBen deskriptiv zu formulieren und die zugehörigen Abläufe erst über eine (ggf. rechnerunterstützte) Graphkonstruktion abzuleiten, werden in manchen Anwendungen Objekt-Lebensläufe so klar vorliegen, daß Transitionsgraphen sofort angegeben werden können. Das dürfte z.B. auf die Übergänge zwischen den Bearbeitungszuständen einer Bestellung wie in Abbildung 7 zutreffen. □

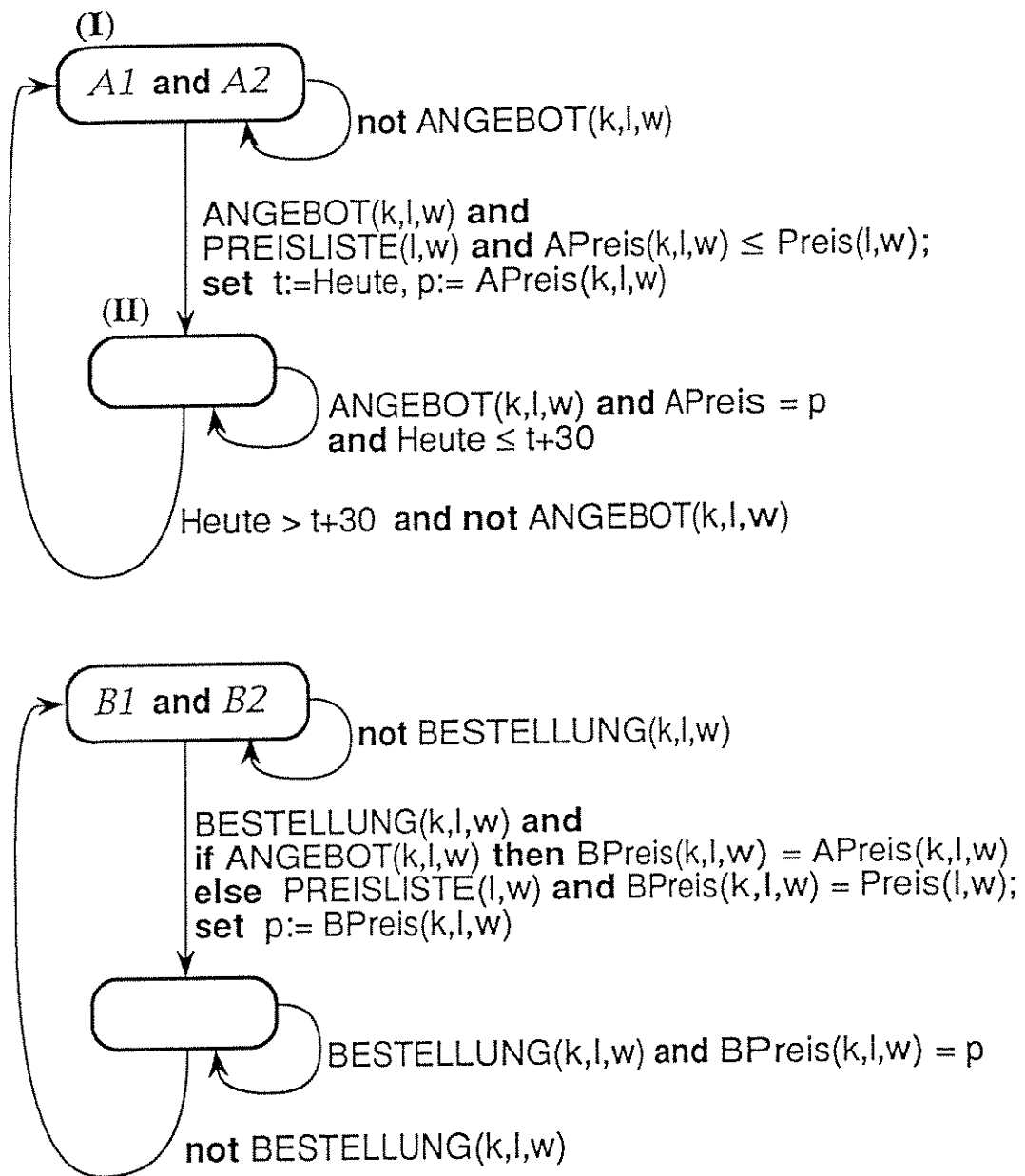


Abbildung 6:

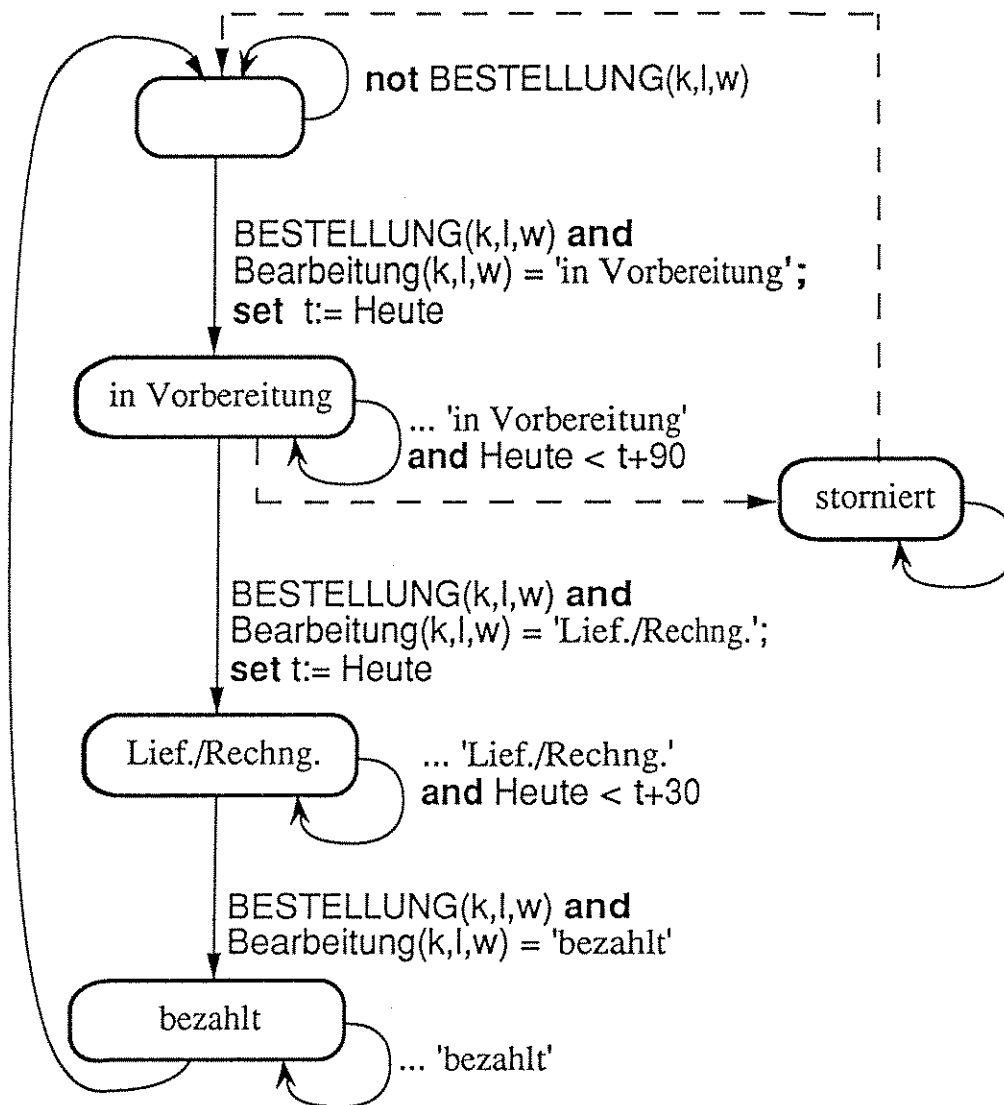


Abbildung 7:

3 Transformation von Integritätsbedingungen

Anschließend sind IBen so zu transformieren, daß zur Laufzeit überwachbare Elemente entstehen. Aus der Architektur eines DBAS in Abbildung 8 lassen sich die verschiedenen Ansatzpunkte zur Integritätsüberwachung ablesen.

Vorrangig und möglichst früh beim Entwurf sollten IBen in Strukturdefinitionen bzw. strukturelle IBen umgesetzt werden, die vom DBMS selbst kontrolliert werden. Unter dieses Prinzip fallen etwa die Reduktion von funktionalen Abhängigkeiten auf Schlüssel(kandidat)bedingungen und die Beseitigung von mehrwertigen Abhängigkeiten, wie sie in der klassischen Entwurfslehre für das Relationenmodell behandelt werden. Solche Transformationen lassen sich durchaus auf die Ebene semantischer Datenmodelle wie z.B. das ER-Modell übertragen, um bereits dort eingesetzt zu werden.

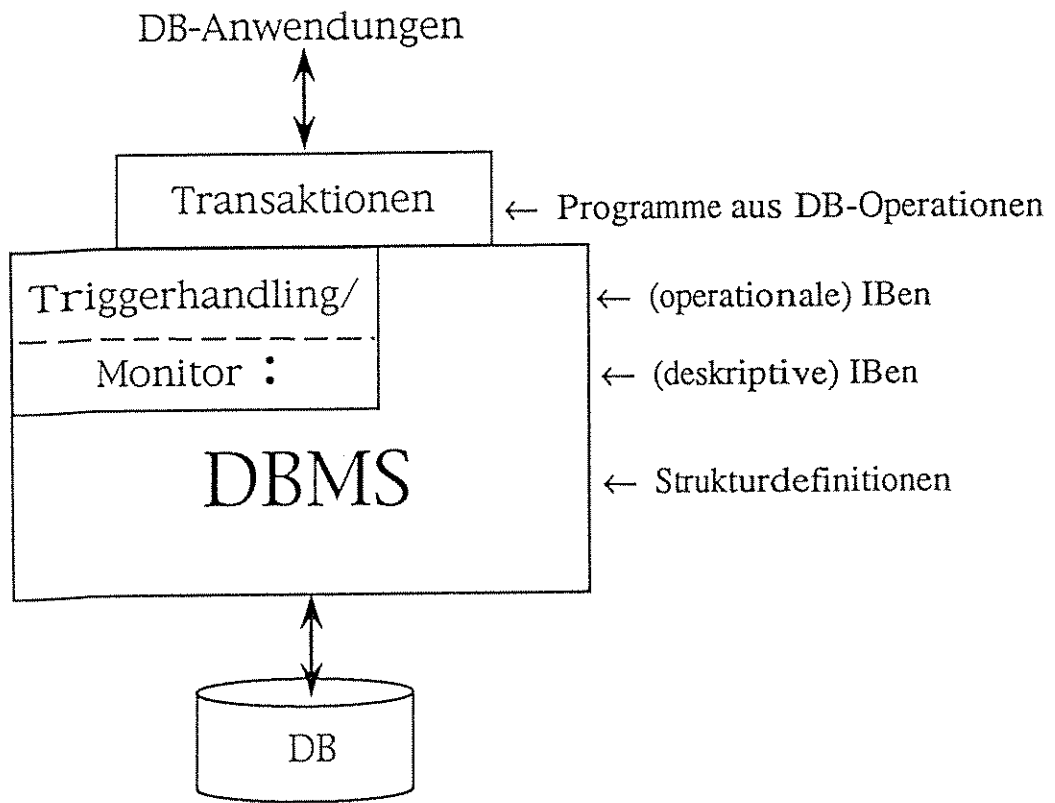


Abbildung 8:

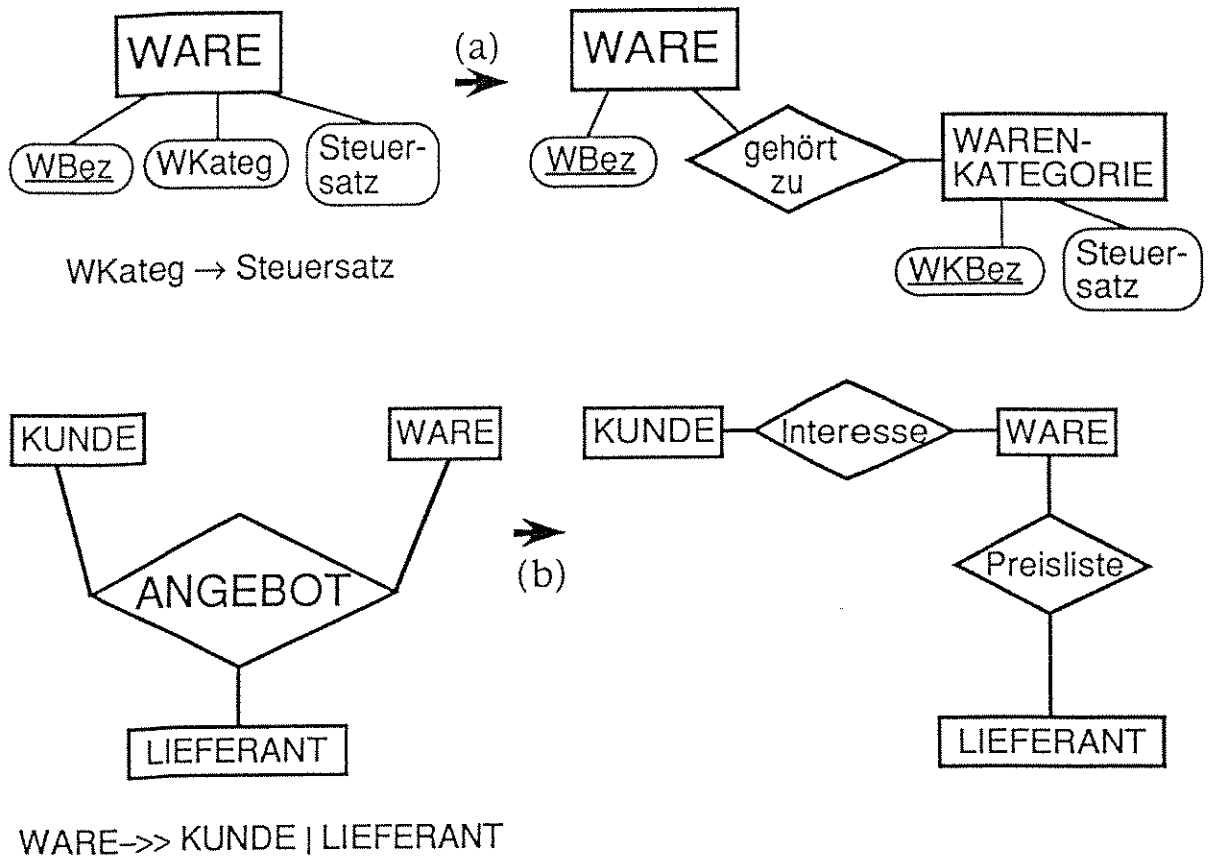


Abbildung 9:

Beispiel: Abbildung 9(a) zeigt die Umwandlung einer funktionalen Abhängigkeit, die zwischen Attributen des Entity-Typs WARE besteht, in eine Schlüsselbedingung für einen neuen Entity-Typ WARENKATEGORIE (entspricht der Umwandlung in 3.Normalform). Abbildung 9(b) zeigt die Beseitigung einer mehrwertigen Abhängigkeit innerhalb eines mehrstelligen Relationship-Typs durch Aufspaltung in binäre Relationship-Typen (entsprechend der 4.Normalform). □

Die übrigen expliziten statischen IBen bräuchten im Fall, daß das DBMS einen Integritätsmonitor enthält, nur in einer passenden Form deklariert zu werden. Heutige kommerzielle DBMS enthalten allerdings auch nach über 16 Jahren einschlägiger Veröffentlichungen (seit [EC75, HM75], vgl. auch [Da83]) immer noch keine Integritätssysteme, weil universelle Monitore die Transaktionsbearbeitung zu ineffizient machen würden. Inzwischen werden aber zunehmend Regeln (Ereignis-Bedingung-Aktionen-Regeln) bzw. Trigger (bei Vorkommen von DB-Operationen in Transaktionen ausgelöste DB-Prozeduren) angeboten und unterstützt — von Forschungsprototypen [Ko89, HCLM90, SK91] und inzwischen mit Einschränkungen auch in Sybase, Ingres und Oracle. Prinzipiell lassen sich damit Monitore simulieren, wenn jede DB-Operation nach folgendem Muster (am Ende einer Transaktion) die Prüfung jeder IB und ggf. ein Rollback auslöst:

```
trigger on < DB-Operationen >:  
if not < IB > then rollback
```

Wegen ihrer Mächtigkeit sollten solche Mittel jedoch sparsam und optimiert eingesetzt werden, um die ansonsten gute Performance eines DBMS nicht zu unterlaufen. Andererseits bieten Trigger die Möglichkeit, Integritätsverletzungen auch aktiv zu korrigieren. Damit wird die Transformation von IBen in aktive und effiziente Operationalisierungen zu einer Entwurfsaufgabe.

Sollte auch kein Triggerhandler zur Verfügung stehen, aber ein korrekt arbeitendes Datenbank-Anwendungssystem gefordert sein, bleibt nur die Möglichkeit, sämtliche Änderungs-transaktionen vorzubereiten, und darin die notwendige Integritätsüberwachung nach ähnlichen Methoden, aber bereits zur Entwurfszeit fest einzubauen.

Wesentlich für den Entwurf von möglichst effizienten Triggern zu DB-Operationen wie **insert/ delete/ update** im Relationenmodell ist es, kritische DB-Objekte und -Operationen für die jeweilige IB zu identifizieren, also Daten, die an IB-Verletzungen beteiligt sein können, und Operationen, die solche Verletzungen bewirken können. Dann kann die IB auf die kritischen DB-Objekte spezialisiert und vereinfacht werden; im Kontext einer Transaktion wäre sie zusätzlich auf die (dort bekannten) Parameter der DB-Operationen spezialisierbar. Für unkritische Operationen brauchen natürlich gar keine Trigger vorgesehen werden. So läßt sich die Anzahl der Prüfungen und insbesondere der dabei erforderlichen Datenbank-Zugriffe verringern. Das typische Muster sieht dann wie folgt aus:

```
trigger on < kritische DB-Operation[en] >:  
if <vereinfachte negierte IB>  
then < Reaktion >  
  - passiv: rollback  
  - aktiv: <Korrektur auf kritischen DB-Objekten>
```

Zulässig sind solche Vereinfachungen, weil bei jeder Transaktion vorausgesetzt werden kann, daß die IB im Zustand vor der Transaktion erfüllt war, so daß sie für invariante Teile der DB erfüllt bleibt. Die Spezifikation bzw. Programmierung von Triggern erfordert, daß auf die durch DB-Operationen betroffenen Daten zugegriffen werden kann; ein relationales DBMS muß also während der Ausführung einer Transaktion sogenannte Differenzrelationen *inserted*, *deleted*, *updated.old* und *updated.new* zu jeder Basisrelation *R* mitführen, die die insgesamt eingefügten, gelöschten und aktualisierten Tupel (letztere jeweils in der Ausprägung vor und nach der Transaktion) enthalten. Damit gilt dann:

$$\text{new } R = \text{old } R - \text{deleted } R - \text{updated.old } R + \text{inserted } R + \text{updated.new } R$$

Tatsächlich bieten kommerzielle Systeme inzwischen solche Hilfsrelationen an, meist in zusammengefaßter Form (*deleted*, *inserted* umfassen eventuell *updated.old* bzw. *updated.new*), aber führen diese wohl nur für einzelne, immerhin mengenorientierte DB-Operationen. Da jedoch erst Transaktionen integritätserhaltende Einheiten bilden sollen, ist für eine vollständige Integritätsüberwachung unverzichtbar, daß der Gesamteffekt einer Transaktion rekonstruiert werden kann, und daß Trigger erst am Ende einer Transaktion aktiviert werden können (wie hier angenommen).

Für Vereinfachungen, welche die durch IB und DB-Operation bestimmten kritischen DB-Objekte ausnutzen, muß typischerweise ein Korrektheitskriterium der folgenden Form gelten:

$$\begin{aligned} &(\text{old IB } \{ \text{IB im Vorzustand} \} \\ &\quad \wedge < \text{old/new-Axiom für DB-Operation} > \\ &\quad \wedge \text{new IB}' \{ \text{vereinfachte IB im Nachzustand} \}) \\ &\iff \text{new IB } \{ \text{IB im Nachzustand} \} \end{aligned}$$

Auf dieser Grundlage gibt es verschiedene, früher im Zusammenhang mit Integritätsmonitoren diskutierte Vereinfachungstechniken, die für den Entwurf von Triggern genutzt werden können; dazu gehören u.a. die Vorauswahl unkritischer DB-Operationen aufgrund von Quantifizierungen in IBen [LR84], die Substitution kritischer Daten für allgemeine Variablen [Ni82, HI85], sowie die Speicherung und differentielle Aktualisierung redundanter Daten [BBC80, Pa84, QW86, QS87]; vgl. auch [GV89].

Die Vorauswahl kritischer DB-Operationen wird im wesentlichen durch folgende Tabelle bestimmt (*R* Relation, *A* Attribut):

IB / Operation:	insert <i>R</i>	delete <i>R</i>	update(<i>A</i>) <i>R</i>	update(<i>B</i>) <i>R</i> (<i>B</i> ≠ <i>A</i>)
∀ <i>r</i> : <i>R</i> ... <i>r</i> . <i>A</i> ...	betroffen	nicht betroffen	betroffen	nicht betroffen
∃ <i>r</i> : <i>R</i> ... <i>r</i> . <i>A</i> ...	nicht betroffen	betroffen	betroffen	nicht betroffen
max{ <i>r</i> . <i>A</i> <i>r</i> : <i>R</i> } < <i>c</i>	betroffen	nicht betroffen	betroffen ¹	nicht betroffen
max{ <i>r</i> . <i>A</i> <i>r</i> : <i>R</i> } > <i>c</i>	nicht betroffen	betroffen	betroffen ²	nicht betroffen

In den Fällen ⁽¹⁾, ⁽²⁾ wäre die IB nicht von der Operation betroffen, wenn bekannt wäre, daß das Attribut *A* verringert ⁽¹⁾ oder erhöht ⁽²⁾ wird. Im allgemeinen werden aber aus ad-hoc-Transaktionen keine Informationen über die Argumente der Operationen gewonnen, so daß nur die gröbere Vorauswahl greift.

Die anderen **T**echniken werden an folgenden Beispielen erklärt:

Beispiel (Spezialisierung): Kritisch für die IB

$$\forall b: \text{BESTELLUNG} \exists k: \text{KUNDE } b.\text{KNr} = k.\text{KNr}$$

können höchstens die Operationen `delete KUNDE` und `insert BESTELLUNG` sein (Updates auf `KNr` entfallen, da Schlüsselattribut). Dazu garantieren folgende Trigger die Einhaltung der **IB**, in denen Spezialisierungen auf die gelöschten Kunden bzw. eingefügten Bestellungen berücksichtigt sind.

```
trigger on delete KUNDE
if  $\exists b: \text{BESTELLUNG} \exists k: \text{deleted KUNDE } b.\text{KNr} = k.\text{KNr}$ 
then rollback "Kunde hat noch Bestellung."
```

oder

```
trigger on delete KUNDE
delete {b: BESTELLUNG |  $\exists k: \text{deleted KUNDE } b.\text{KNr} = k.\text{KNr}$ }
```

und

```
trigger on insert BESTELLUNG
if  $\exists b: \text{inserted BESTELLUNG } \nexists k: \text{KUNDE } b.\text{KNr} = k.\text{KNr}$ 
then rollback "Kunde unbekannt"
```

oder

```
trigger on insert BESTELLUNG
insert KUNDE < Ergänzung der fehlenden Kundendaten > □
```

Bei der Beispielbedingung handelt es sich um eine referentielle IB, für die der SQL-Standard und heutige Systeme bereits passende Behandlungen vordefiniert anbieten, z.B. die Optionen "restricted" oder "cascading delete", die den obigen `delete`-Triggern entsprechen. (Für `insert` ist aber nur eine restriktive Behandlung vorgesehen.) Manuell erstellte Trigger hingegen erlauben bei Bedarf flexiblere Reaktionen wie Rückmeldungen an den Benutzer oder interaktive Ergänzungen fehlender Daten — vorausgesetzt, die Definitionssprache ist mächtig genug.

Beispiel (differentielle Aktualisierung redundanter Daten):

Gegeben seien die Relationenschemata

`ANG[ESTELLTER] (Name,...,AbtNr)`, `ABT[EILUNG] (Nr,...)`

sowie die IB, daß die Anzahl der Angestellten pro Abteilung 100 nicht übersteigt:

$$\forall \text{abt}: \text{ABT count}\{a: \text{ANG} \mid a.\text{AbtNr} = \text{abt.Nr}\} \leq 100$$

Um zur Prüfung der IB Zugriffe auf die Relation `ANG` möglichst zu vermeiden, wird die Relation `ABT` um ein Attribut `Anzahl` erweitert und die IB durch folgende Bedingungen ersetzt:

- (i) $\forall \text{abt}: \text{ABT } \text{abt.Anzahl} \leq 100$
- (ii) $\forall \text{abt}: \text{ABT } \text{abt.Anzahl} = \text{count}\{a: \text{ANG} \mid a.\text{AbtNr} = \text{abt.Nr}\}$

Zur Einhaltung der Beziehung (ii) ist folgender Trigger erforderlich (wiederum zur Ausführung am Transaktionsende):

```

trigger on insert | delete | update(AbtNr) ANG :
update {abt: ABT} set Anzahl := Anzahl
    - count{a: deleted ANG  $\cup$  updated.old ANG | a.AbtNr = abt.Nr}
    + count{a: inserted ANG  $\cup$  updated.old ANG | a.AbtNr = abt.Nr}

```

Die eigentliche IB (i) läßt sich dann durch einen Trigger absichern, der erst durch diese Nachführung des Attributs Anzahl angestoßen wird:

```

trigger on update(Anzahl) ABT :
if  $\exists$ abt: updated.new ABT abt.Anzahl > 100 then rollback □

```

Auch zu dynamischen IBen können ausgehend von einer Darstellung durch Transitionsgraphen systematisch Trigger aufgestellt und sogar vereinfacht werden. Dazu müssen die DB-Strukturen um eine Modellierung von "Situationen", die DB-Objekte in solchen Graphen durchlaufen, erweitert werden; dies sind i.w. die Knoten zusammen mit für die Überwachung zusätzlich benötigten historischen Informationen. Häufig lassen sich diese Situationen aber auch bereits aus vorhandenen DB-Strukturen ableiten. Die durch Trigger zu prüfenden und ggf. herzustellenden Bedingungen können den Kanten der Transitionsgraphen entnommen werden, sind aber nach den erreichten Situationen zu differenzieren.

Zu einer gegebenen temporalen IB φ mit Variablen ob_1, \dots, ob_k für DB-Objekte wird also eine neue Relation

$$\text{SITUATION}_{\varphi}(ob_1, \dots, ob_k, \text{node}, \langle \text{sonstige historische Daten} \rangle)$$

eingeführt. Darin werden DB-Objekte durch ihre Primärschlüssel und Knoten durch Nummern oder Bezeichnungen für Situationen, z.B. "in Vorbereitung", repräsentiert. Die Trigger sind nach folgendem Grundmuster aufgebaut, wobei jeder Knoten einen Kontext wie in Abbildung 10 habe.

```

trigger on < kritische DB-Operation[en] >
case old SITUATION $_{\varphi}$ .node
...
s : if not (out0 and new SITUATION $_{\varphi}$ .node = s)
    and not (out1 and new SITUATION $_{\varphi}$ .node = s1)
    ...
    and not (outn and new SITUATION $_{\varphi}$ .node = sn)
    then rollback | < Korrektur >
...

```

Um zu Spezialisierungen auf betroffene DB-Objekte zu kommen, lassen sich hier die Bedingungen, die beim Erreichen einer Situation gelten müssen und die aus den jeweiligen Eingangskanten abgelesen werden können, als vor einer Transaktion erfüllt voraussetzen. Im Vorzustand gilt also:

$$\text{old SITUATION}_{\varphi}.\text{node} = s \implies in_0 \vee in_1 \vee \dots \vee in_m$$

Falls die rechte Seite (die Disjunktion der Eingangsbedingungen) unter einer Operation op auf einem Objekt ob invariant bleibt oder mit $op(ob)$ unvereinbar ist, so ist $op(ob)$ für die

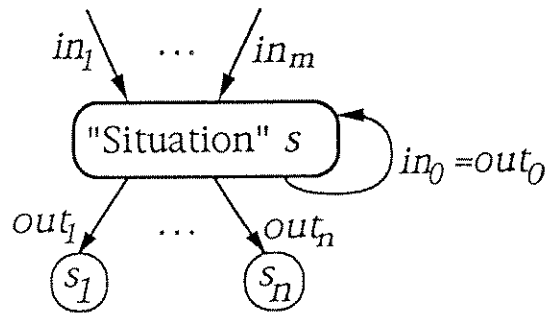


Abbildung 10:

IB φ und den Fall s unkritisch. (Wir setzen voraus, daß nur übliche temporale Formeln, die nicht auf den absolut nächsten Zustand Bezug nehmen, verwendet werden; die zugehörigen Transitionsgraphen sind dann iterationsinvariant, d.h. zu jeder Eingangskante in gibt es eine Schleife out mit $(in \Rightarrow out)$, so daß bei invarianter Eingangsbedingung die Situation erhalten bleibt [Li89].) Bleibt die Bedingung für beliebige Objekte invariant unter einer DB-Operation, ist die Operation für die gesamte IB unkritisch; ansonsten sind die zugehörigen Trigger durch Ausnutzung der teilweisen Invarianten zu vereinfachen, wie das folgende Beispiel zeigt. Eine ausführliche Beschreibung der Erzeugung von Triggern aus dynamischen IBen findet sich in [GL92].

Beispiel: Betrachte die IB (A1 and A2) aus Abb. 5 mit zugehörigem Transitionsgraph in Abb. 6; als DB-Strukturen werden die zum ER-Schema in Abb. 4 äquivalenten Relationenschemata sowie eine (einwertige) Relation Heute mit dem aktuellen Kalenderdatum verwendet. In diesem Beispiel können Situationen (I,II) durch vorhandene DB-Informationen charakterisiert werden, denn aus dem Graphen ist ablesbar, daß gilt:

$$\begin{aligned} \text{SITUATION}(k, l, w, I, \dots) &\Leftrightarrow \text{not ANGEBOT}(k, l, w) \\ \text{SITUATION}(k, l, w, II, \dots) &\Leftrightarrow \text{ANGEBOT}(k, l, w) \end{aligned}$$

Deshalb braucht keine zusätzliche Relation SITUATION eingeführt werden; benötigt wird allerdings als zusätzliche historische Information das Datum der Angebotserstellung, das hier als Attribut Datum in der Relation ANGEBOT untergebracht sei.

Die Operationen delete/update ANGEBOT erweisen sich als unkritisch für die Situation (I), ebenso insert ANGEBOT im Fall (II). Für die anderen Kombinationen sind aber Trigger erforderlich, z.B. die folgenden:

- [I] trigger on insert ANGEBOT :
- if $\exists a$:inserted ANGEBOT $\nexists p$:PREISLISTE
- $a.l = p.l$ and $a.w = p.w$ and $a.APreis \leq p.Preis$
- then rollback
- else update { a :inserted ANGEBOT} set $a.Datum := \text{Heute}$
- [II] trigger on delete ANGEBOT :
- if $\exists a$:deleted ANGEBOT $\text{Heute} \leq a.Datum + 30$
- then rollback
- trigger on update(APreis) ANGEBOT : rollback

trigger on update Heute :

delete {a: ANGEBOT | Heute > a.Datum + 30}

□

4 Ausblick

Insgesamt kann die Integritätserhaltung durch Trigger als genauso zuverlässig, aber effizienter und flexibler als eine Überwachung durch einen universellen Monitor beurteilt werden; dies trifft sogar auf referentielle IBen zu, bei denen die inzwischen angebotenen Unterstützungen für manche Anwendungen zu starr sein dürften. Voraussetzung ist natürlich, daß Trigger systematisch (nach den diskutierten Regeln) entworfen werden, und daß das implementierte Triggerhandling durchschaubar und flexibel arbeitet.

Entwurf und Optimierung von Triggern eignen sich größtenteils für eine automatische Unterstützung [CW90], und zwar mindestens durch Führung des Entwerfers entsprechend den Entwurfsregeln sowie bestenfalls durch Einsatz automatischer Beweistechniken wie etwa Resolutionsverfahren. Eine Interaktion mit dem DB-Entwerfer wird aber unverzichtbar bleiben. Außerdem werden Analysewerkzeuge benötigt, um Zyklen in den Triggern zu entdecken.

Durch Trigger können alle, auch ad-hoc programmierte Transaktionen integritätserhaltend gemacht werden. Wenn das DBMS kein Triggerhandling unterstützt, müssen solche Transaktionen vom Entwerfer vollständig vorbereitet werden. Bei der Programmierung kann dies durch den Abschluß von DB-Operationen gegen Triggerfolgen passieren; bei der Spezifikation empfiehlt sich eher eine direkte Kombination von Transaktionspezifikationen mit Integritätsbedingungen bzw. Transitionsgraphen in einer transitionalen Logik [Li89, Li90]. In allen Fällen lassen sich IBen durch Ausnutzen des Transaktionskontextes, insbesondere von Invarianten unter einer Transaktion, vereinfachen. Auch der Transaktionsentwurf ist automatisch und interaktiv unterstützbar [ICE90].

Literatur

- [BBC80] P.A. Bernstein, B.T. Blaustein, E.M. Clarke: Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data. In *Proc. 6th Int. Conf. on Very Large Data Bases*, 126-136, 1980
- [CW90] S. Ceri, J. Widom: Deriving Production Rules for Constraint Maintenance. In D. McLeod, R. Sacks-Davis, H. Schek (eds.), *Proceedings of the 16th Int. Conf. on Very Large Data Bases*, 566-577, Morgan Kaufmann Publishers, 1990.
- [Da83] C. Date: *An Introduction to Database Systems, Vol. II*, Addison-Wesley, 1983.
- [EC75] K. Eswaran, D. Chamberlin: Functional Specifications of a Subsystem for Data Base Integrity. In *Proc. Int. Conf. on Very Large Data Bases 1975*, 48-68, 1975.
- [EGHHLSE90] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, H.-D. Ehrich: *Conceptual Modelling of Database Applications Using an Extended Entity-Relationship Model*. Informatik-Bericht 90-05, TU Braunschweig 1990

- [GL92] M. Gertz, U.W. Lipeck: Deriving Integrity Maintaining Triggers from Transition Graphs. Institut für Informatik, Univ. Hannover 1992 (submitted for publication)
- [GV89] G. Gardarin, P. Valduriez: *Relational Databases and Knowledge Bases*. Addison-Wesley, Reading (Mass.), 1989.
- [HCLM90] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson: Starburst Mid-Flight: As the Dust Clears. *IEEE Trans. Knowledge and Data Engineering 2 (1990)*, 143-160.
- [HI85] A. Hsu, T. Imielinski: Integrity Checking for Multiple Updates. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 152-168, 1985.
- [HM75] M. Hammer, D. McLeod: Semantic Integrity in a Relational Database System. In *Proc. Int. Conf. on Very Large Data Bases*, 25-47, 1975.
- [ICE90] Projektgruppe ICE (Integrity-Centered Environment): Werkzeuge für den integritätszentrierten Entwurf von Datenbank-Anwendungssystemen. Interner Bericht, Fachbereich Informatik, Universität Dortmund 1990.
- [Ko89] A.M. Kotz: *Triggermechanismen in Datenbanksystemen*. Informatik-Fachbericht 201, Springer, Berlin 1989.
- [Li89] U. W. Lipeck: *Dynamische Integrität von Datenbanken: Grundlagen der Spezifikation und Überwachung*. Informatik-Fachbericht 209, Springer, Berlin 1989.
- [Li90] U. W. Lipeck: Transformation of Dynamic Integrity Constraints into Transaction Specifications. *Theoretical Computer Science 76 (1990)*, 115 - 142.
- [LR84] T.-W. Ling, P. Rajagopalan: A Method to Eliminate Avoidable Checking of Integrity Constraints. In *Proc. IEEE Trends & Applications Conference: Making Database Work*, 60-69, 1984.
- [LS87] U.W. Lipeck, G. Saake: Monitoring Dynamic Integrity Constraints Based on Temporal Logic. *Information Systems 12 (1987)*, 255-269.
- [LS88] Lipeck, U.W. and Saake, G.: Entwurf von Systemverhalten durch Spezifikation und Transformation temporalen Anforderungen. *Proc. GI Jahrestagung, Band II (R. Valk, Hrsg.)*, 449-463, Informatik-Fachbericht 188, Springer, Berlin 1988
- [Ni82] J.-M. Nicolas: Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica 18 (1982)*, 227-253.
- [Pa84] R. Paige: Applications of Finite Differencing to Database Integrity Control and Query/Transaction Optimization. In: *Advances in Database Theory Vol.2 (H.Gallaire, J.Minker, J.Nicolas, eds.)*, Plenum Press, New York 1984.
- [QS87] X. Qian, D. R. Smith: Integrity Constraint Reformulation for Efficient Validation In *Proc. 13th Int. Conf. on Very Large Data Bases*, 417-425, 1987
- [QW86] X. Qian, G. Wiederhold: Knowledge-based Integrity Constarint Validation. In *Proc. 12th Int. Conf. on Very Large Data Bases*, 3-12, 1986
- [Sa91] G. Saake: Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering 6 (1991)*, 47-73.
- [SK91] M. Stonebraker, G. Kemnitz: The Postgres Next-Generation Database Management System. *Communications of the ACM 34:10 (October 1991)*, 78 - 92.