MC SYLLABUS 46.1

COLLOQUIUM DATABANKORGANISATIE

DEEL 1

P.M.G. APERS (red.)

MATHEMATISCH CENTRUM AMSTERDAM 1981

1980 Mathematics subject classification: 68-06, 68A05, 68B20, 68H05

ACM-Computing Reviews-category: 4.33

INHOUD

VOORWOORD

Het colloquium databank-organisatie, dat in samenwerking met het Mathematisch Centrum wordt georganiseerd, staat onder leiding van drs. P.M.G. Apers (Vrije Universiteit), prof. J.M. van Oorschot (PTT en Vrije Universiteit), prof.dr. J.A. van der Pool (IBM en TH Twente), drs. F. Remmen (TH Eindhoven) en prof.dr. R.P. van de Riet (Vrije Universiteit). In het eerste semester zijn er een zestal lezingen geweest.

De bijdrage van Van de Riet omvat een globaal overzicht van het huidige onderzoek op het gebied van databases, en een meer gedetailleerde beschrijving van de noodzakelijke uitbreidingen van de programmeertaal PLAIN om privacy en security problemen op te lossen.

Date behandelt in zijn bijdrage de begrippen locking en recovery gezien vanuit het standpunt van de applicatieprogrammeur. Vijf niveaus van consistentie worden ingevoerd; hoe hoger het niveau, des te lager de concurrency.

In de bijdrage van De Brock wordt een semantisch model van het conceptuele niveau van een database ingevoerd, gebaseerd op verzamelingenleer, met het doel een basis te leggen om de verschillende datamodellen te vergelijken.

Vandenbulcke bekijkt in zijn bijdrage het ontwerp van een database in een bedrijf. Behandeld wordt welke informatie verzameld moet worden om het conceptuele gegevensmodel op te stellen, en hoe dit model uiteindelijk omgezet kan worden in een fysiek model.

Het geheimhouden van bepaalde informatie in een database is het onderwerp van de bijdrage van De Jonge. Criteria worden ontwikkeld op grond waarvan het systeem de beslissing neemt een vraag te beantwoorden of niet.

Kleefstra behandelt in zijn bijdrage de structurele aspecten van een datamodel. Een definitie van het begrip datamodel wordt gegeven en ook een formalisme om de verschillende datamodellen te specificeren.

Wij danken het Mathematisch Centrum, en met name dr. J.C. van Vliet, voor de goede samenwerking.

P.M.G. Apers

DATABANKEN, ENKELE ONDERZOEKSASPECTEN

R.P. van de RIET
Vrije Universiteit, Amsterdam

1. INLEIDING

Als eerste voordracht in het colloquium over databanken is het gepast om een overzicht te geven van de huidige stand van zaken op het gebied van onderzoek van databanken.

Reeds spoedig na de aanvang van de voorbereidingen bleek echter dat dit doel te hoog gegrepen was voor een enkele voordracht. Om een indruk te geven van de hoeveelheid onderzoeksmateriaal vermelden we dat er vijf tijdschriften zijn die speciaal op het gebied van databanken worden uitgegeven, dat er elk jaar minstens drie conferenties op dit gebied gehouden worden en dat Mohan, in een overzichtsartikel [44] dat eind 1978 verscheen, 415 publicaties noemt die vrijwel allemaal na 1975 verschenen zijn.

In het tweede hoofdstuk zullen we aan de hand van dit artikel een kort overzicht geven, waarmee een summiere basis voor het colloquium gelegd is. Het onderwerp datamodellen zal dan nader uitgewerkt worden in het derde hoofdstuk. Hier gaan we in op enkele concepten die gebruikt worden om een model van de werkelijkheid te maken, zoals de abstractiebegrippen: classificatie, generalisatie en aggregatie. Tevens geven we een inleiding in het relationele model en signaleren we enkele daarmee samenhangende problemen.

In het laatste hoofdstuk bespreken we de wijze waarop een databank geprogrammeerd kan worden: programmeertalen, operaties en het gebruik van abstracte datatypen. Dit laatste gebeurt met behulp van de taal PLAIN [69], een taal die ontworpen en geimplementeerd wordt als gemeenschappelijk onderzoeksproject van onze vakgroep, daartoe in staat gesteld door Z.W.O., en een groep onder leiding van Wasserman van de universiteit van Californië te San Francisco en Berkeley.

Het zal ons er in het bijzonder om gaan te onderzoeken of het mogelijk is op veilige wijze in de definitie van databankmanipulaties vast te leggen wie wat met deze manipulaties mag doen (privacy) en hoe de interne

consistentie van de databank gewaarborgd kan worden.

2. EEN KORT OVERZICHT

Aan de hand van het artikel van MOHAN [44] zullen we in vogelvlucht een kijkje nemen op het databaselandschap met zijn bergen van gegevens, zijn talen om te programmeren en zijn databanken voor de vermoeide onderzoeker.

2.1. Datamodellen

In een databank gaat het om het vastleggen van gegevens van een model van een stukje werkelijkheid. De structuur hiervan wordt aan de ene kant vastgelegd omdat je er relaties uit de werkelijkheid mee wilt beschrijven, aan de andere kant moet het model vertaald kunnen worden naar bekende en beschikbare datastructuren.

De traditionele datamodellen zijn het hiërarchische datamodel, waarbij data in de vorm van boomstructuren is opgeborgen, het IMS systeem van IBM is hier het beste voorbeeld van, verder het netwerkmodel, waarbij data in de vorm van een meer algemener netwerk of graaf wordt gerepresenteerd, dit wordt algemeen als het DBTG model omschreven en tenslotte het relationele model ingevoerd door CODD [17]. Het laatste model zal uitvoerig in het volgende hoofdstuk worden besproken en is ook de basis van het laatste hoofdstuk.

Abstracties van objecten uit de werkelijkheid worden vaak entiteiten genoemd.

We noemen de namen van de volgende modellen: Het entity-relationship model van CHEN [16], het entity-property-association model van PIROTTE [48], het information space model van KOBAYASHI [36], het DIAM II model van SENKO [52], semantische netwerken van ROUSSOPOULOS en MYLOPOULOS [50], de infologische benadering van LANGEFORS en SUNDGREN [38], het functionele model van KERSCHBERG en SZETO [34].

Een nieuw datamodel, dat het mogelijk maakt om de verschillende rollen die entiteiten uit de werkelijkheid kunnen spelen te representeren, werd geintroduceerd door BACHMAN [8].

Tenslotte vermelden we hier twee studies van KLEEFSTRA [35] en LINDENCRONA-OHLIN [40], de eerste die het onderwerp fundamenteel aanpakt, en de tweede die een groot aantal modellen vergelijkt.

De studiegroep ANSI/SPARC, internationaal en heterogeen van

samenstelling, heeft baanbrekend werk verricht om tot standaardisatie van het databankwereldje te komen. Een belangrijk resultaat, dat tot nu toe bereikt is, is het CODASYL DBTG voorstel [63], waarin een streng onderscheid wordt gemaakt tussen drie verschillende datamodellen. Het conceptuele model dat de totale logische inhoud van de databank beschrijft, het externe model dat beschrijft hoe gebruikers tegen hun stukje van de databank aankijken en het interne model dat de dataopslagstructuren vastlegt.

Het vergelijken van het relationele model met andere, met name het DBTG netwerkmodel, is onderwerp van veel studie geweest. Kobayashi ontwierp een speciaal model om beide modellen te beschrijven [36] en Gerritsen ontwierp compilers om vragen die aan een databank in de relationele vorm werden gesteld om te zetten naar vragen in DBTG vorm en vice versa [27].

2.2. Flexibele interfaces

Omdat er steeds meer gebruikers van de diensten van een databank gebruik willen maken zijn er twee methoden ontwikkeld om dit mogelijk te maken. De ene is om de gebruiker een natuurlijke taal te laten gebruiken om zijn vragen in te stellen. Het RENDEZVOUS systeem is hiervan een voorbeeld; het werd door Codd ontwikkeld om de "domme" gebruiker in zijn eigen taal te laten converseren met het systeem [18]. De tweede methode is om een query (vraag), te laten formuleren in een programma. Soms is de programmeertaal aangepast aan dit gebruik, PLAIN is daarvan een voorbeeld; soms moet de query in de vorm van een aantal procedure-aanroepen worden geformuleerd, het gebruik van DL1 in PL/1 is daarvan en voorbeeld (zie DATE [20]); soms ook vindt pre-processing plaats, voorbeelden zijn SQL van System R [6] en EQUEL [58]. Het transformeren van een query naar een andere vorm, die qua rekentijd beter is, is een interessante bezigheid, waar we in het volgende hoofdstuk even op in zullen gaan.

2.3. Abstractie van databanken

Het idee van een abstracte datatype is dat het de structuur van de data, tezamen met de operatoren, die op deze data moeten werken, definieert. Op deze manier wordt de structuur nauwer met de operaties verbonden. Sterker, wat essentieel is voor de gebruiker, namelijk de operaties die hij kan uitvoeren, wordt de basis voor de structuur van de data. Eigenlijk doet voor die gebruiker de structuur er dan niet meer zo toe.

Voor hem gaat het er om wat de operaties doen en niet hoe ze het doen. In principe is het zo mogelijk om abstracte specificaties op te stellen voor de verschillende operaties los van de keuze van de datastructuren.

Diverse onderzoekers, zoals BRODIE [12], hebben abstracte datatypen gebruikt om een databank te beschrijven en de operaties hierop. Wij willen in deze voordracht de techniek van abstracte datatypen gebruiken om semantische integriteitsregels en protectieregels vast te leggen. Semantische integriteitsregels zijn regels waaraan de neergelegde gegevens moeten voldoen, terwijl ook de consistentie van de gegevens ermee wordt vastgelegd. Voorbeelden: leeftijd is een niet-negatief getal kleiner dan 150; elke persoon heeft één vader. Protectieregels beschrijven welke gebruikers recht hebben om welke operaties op welke gegevens uit te voeren. Voorbeelden: alleen de gebruiker zelf kan zijn paswoord veranderen; de computerbaas mag een gebruiker in de gebruikersfile toevoegen en weglaten.

2.4. Gelijktijdig gebruik van de databank door meerdere gebruikers

Omdat bij grote databanken meerdere gebruikers tegelijkertijd met de databank willen converseren, moeten speciale maatregelen genomen worden om een gegarandeerd niveau van consistentie te bereiken.

Als meerdere gebruikers tegelijk willen lezen, dan is er niets aan de hand, behalve dat het systeem moet beslissen welke gebruiker hij in de eerst komende milli-seconde moet bedienen. Wanneer een van de gebruikers echter veranderingen wil aanbrengen dan moeten die stukken van de databank op slot worden gezet die veranderd moeten worden, zodat iemand die de data leest niet de ene keer dit en de andere keer wat anders leest.

Het op slot zetten heeft echter weer tot gevolg dat er een mogelijkheid tot "deadlock" ontstaat.

Om deadlock te voorkomen zijn er in principe twee mogelijkheden. De eerste is dat gebruikers van te voren moeten vertellen wat ze willen, zodat het systeem van te voren alle delen van de databank kan reserveren en aldus in staat is deadlock te voorkomen. Dit wordt deadlock prevention genoemd.

De tweede is dat het systeem de mogelijkheid heeft om stukken data die een gebruiker in zijn bezit heeft weer af te nemen en dit na alle veranderingen die die gebruiker heeft aangebracht weer ongedaan te hebben gemaakt, aan een andere gebruiker te geven. Dit wordt deadlock preemption genoemd.

Omdat de eerste methode de vervelende eigenschap heeft dat er grote delen van de databank gedurende lange tijd op slot gezet kunnen worden,

wordt veelal de tweede methode gebruikt. In dit verband wordt vaak het begrip transactie gebezigd, hetgeen een aantal elementaire handelingen op de databank definieert die stuk voor stuk weer ongedaan kunnen worden gemaakt.

Ook moet hier het zogenaamde two-phase commit protocol genoemd worden van GRAY [21,28], dat bepaalde eigenschappen aan transacties oplegt en wel zo dat een transactie bestaat uit eerst het aanvragen van locks en daarna in omgekeerde volgorde het weer vrijgeven van de locks. Transacties volgens dit protocol hebben de eigenschap dat ze, wanneer ze simultaan worden uitgevoerd, dezelfde resultaten opleveren als wanneer ze in een zekere volgorde na elkaar worden uitgevoerd.

Een geheel ander wezenlijk probleem is de mogelijkheid om na een ernstige systeemfout de databank weer in een goede conditie te krijgen. Dit is het zogenaamde recovery probleem. Er worden checkpoints ingevoerd die de toestand van de databank vastleggen op bepaalde welgekozen momenten. Op welke tijdstippen die momenten gekozen moeten worden is een interessant probleem. Zie bijvoorbeeld ANGELUCCI [2].

2.5. Herstructureren van een databank

Op grond van prestatiemetingen kan het blijken dat het nodig is de databank te herstructureren. Bijvoorbeeld dat er een extra toegangspad (index) wordt gemaakt om een snellere toegang te verkrijgen. Of dat rehashing nodig is omdat de primaire hashtabellen te klein blijken te zijn.

Echter, het kan ook betekenen dat tabellen gesplitst moeten worden of dat kolommen aan bestaande tabellen moeten worden toegevoegd. Ook is het mogelijk dat bepaalde hiërarchische structuren gewijzigd moeten worden. Dit laatste is in het bijzonder van belang voor IMS databanken. TAYLOR et al[55] hebben een systeem gemaakt, XPRS, dat uitgaande van een beschrijving van het huidige datamodel van de databank en een beschrijving van het nieuwe datamodel een PL/1 programma maakt dat met IMS statements de databank herstructureert.

2.6. Gespreide databanken

Doordat er openbare en privé dataverkeersnetten zijn, gaan steeds meer grote bedrijven met geografisch verspreide filialen er toe over hun computers te spreiden en bij de filialen neer te zetten. Deze computers worden echter met elkaar via de datanetten verbonden, zodat automatisch data die de ene computer nodig heeft en die in de andere computer aanwezig is wordt

overgestuurd.

Dit heeft aanleiding gegeven tot zeer veel studie en ook enkele implementaties van zogenaamde gespreide databanken, zoals bijvoorbeeld het SDD1 systeem [49] en het SIRIUS project [39].

Op de VU zijn we bezig met de implementatie van een gespreide versie van het Ingressysteem (zie THOMAS [61]). Ook worden optimalisatieproblemen bestudeerd in de zin van: als je verschillende computers hebt, wie moet dan wat doen om bijvoorbeeld de transporttijd te minimaliseren (zie APERS [3,4]).

Een prima introductie in de problematiek van gespreide databanken is het boek van BERNSTEIN, ROTHNIE en SHIPMAN [9], waarin een aantal belangrijke artikelen is opgenomen.

2.7. Databankmachines

Op hardwareniveau is veel gedaan op het gebied van databanken. Zo werd op de Universiteit van Florida het CASSM (Content Addressed Segment Sequential Memory) systeem gemaakt dat bestaat uit een rij van niet-numerieke microprocessoren die het mogelijk maken om met behulp van inhoud en context te adresseren in een grote databank (zie SU [59]). TODD [62] beschrijft de architectuur van hardware waarbij magnetisch bubble geheugen en een aantal microprocessoren worden gebruikt voor een relationeel databanksysteem.

Bij het RARES systeem van de Universiteit van Utah wordt een "head-per-track" techniek gebruikt om tuples uit relaties op een disk op te bergen (zie LIN [40]).

De RAP (Relational Associative Processor) machine, ontwikkeld op de Universiteit van Toronto is een cellulaire machine, waarbij elke cel kan rekenen, vergelijken en I/O kan doen en een sequentieel circulair geheugen gebruikt voor het opbergen van relationele gegevens (zie OZKARAHAN [46]).

HSIAO [32] heeft op de Ohio State Universiteit een data base computer (DBC) ontworpen waarvan als bijzonderheid vermeld kan worden dat het een ingebouwd protectiemechanisme heeft.

Slechts op twee onderwerpen uit de boven besproken lijst willen we in de twee volgende hoofdstukken ingaan: datamodellen en programmering van databankmanipulaties. De meeste andere onderwerpen worden door andere sprekers in dit colloquium behandeld.

Tenslotte verwijzen we nog naar het nieuwe boek van ULLMAN [64], waar

veel van de onderzoeksresultaten van de laatste jaren in beschreven zijn.

3. DATAMODELLEN

Aan de hand van drie artikelen willen we het onderwerp datamodellen en het ontwerpen ervan behandelen.

Het zijn het bekende rapport van de ANSI/X3/SPARC werkgroep [63], een artikel van SMITH en SMITH [56] en een artikel van BUBENKO [15].

3.1. Het concept van een informatiesysteem

Er zijn vele manieren om tegen een informatiesysteem aan te kijken dat met een computer gerealiseerd wordt. Eén manier is het systeem te beschouwen als een verzameling gegevens in de vorm van files, met als eigenschap dat bij zekere invoergegevens een bepaalde uitvoer wordt geproduceerd. Een meer toepassingsgerichte kijk is te zeggen dat het systeem een model representeert van een stukje toepassingsrealiteit. Dit model omvat beslissingsprocedures, voorspellingsregels, rekenregels, regels om veranderingen van toestanden aan te brengen, etc. Zulk een model is gemaakt om redelijk nauwkeurig de werkelijkheid na te bootsen, in het bijzonder wat betreft het rapporteren van transacties en gebeurtenissen van de toepassingswerkelijkheid.

Het ligt dan ook voor de hand dat een model van het informatiesysteem ontworpen moet worden alvorens tot de daadwerkelijke technisch-physische implementatie kan worden overgegaan. Het probleem hierbij is dat formele methoden om vanuit de werkelijkheid tot een formeel ontwerp te komen nauwelijks bestaan.

Desalniettemin is het toch noodzakelijk dat er op hoog niveau een model wordt vastgelegd waarin alle concepten beschreven staan. Dit zogenaamde conceptuele model bevat beschrijvingen van alle objecten, eigenschappen van objecten, relaties tussen objecten en de handelingen (operaties) die met die objecten moeten worden uitgevoerd.

Een conceptueel model heeft twee functies:

1. Het is een basis voor discussies met o.a. de gebruikers over de vraag hoe de werkelijkheid geabstraheerd moet worden en welke veronderstellingen en welke regels er gelden. Hierbij hoort dat de verschillende "views" die gebruikers op de databank zullen hebben geintegreerd moeten worden tot één model. Dit is wat de ANSI/SPARC groep het conceptuele

schema noemt.

2. Het is een basis om een efficiënte implementatie in termen van datastructuren van het databanksysteem mogelijk te maken.

Het ANSI/SPARC rapport onderscheidt drie niveaus van modellen. Het middelste niveau is als het ware centraal. Het bestaat uit een verzameling zogenaamde external schema's. Een external schema legt formeel vast welk gedeelte van het informatiesysteem voor welke gebruiker beschikbaar is en hoe hij er tegen aan kijkt. Het is als het ware een "view" van die gebruiker. De integratie van alle externe schema's wordt conceptual schema genoemd en beschrijft het gehele informatiesysteem als model van een stukje werkelijkheid (de "enterprise"). De vertaling van het conceptual schema in termen van datastructuren en physische opslagstructuren is een formeel model dat internal schema wordt genoemd.

Onderzoekers in kunstmatige intelligentie verdelen kennis vaak in concrete en abstracte kennis. het eerste bestaat uit feiten, beweringen over individuele objecten en relaties tussen de objecten. Een voorbeeld is een gegeven als "leverancier S levert product P aan project J". Dit is een concreet feit dat ook een relatie tussen de verzamelingen leveranciers, producten en projecten aangeeft. Zulke verzamelingen worden entity sets genoemd. De objecten zelf worden entiteiten genoemd.

Representatie in een computergeheugen kan in principe met behulp van files en tabellen geschieden.

Abstracte kennis die de semantische integriteitseisen omvat is kennis die ons in staat stelt logische conclusies te trekken. Bijvoorbeeld kunnen we uit de bovenstaande concrete kennis concluderen dat leverancier S het product P kan leveren en dat product P een onderdeel is van project J.

Ook kan de abstracte kennis ietsje concreter zijn, zoals:

. elke employé is een persoon
. elke dag is een deel van een maand
. een chauffeur is een employé die een transport moet verrichten.

De voorbeelden geven aan dat abstracte kennis vaak betrekking heeft op collecties van objecten. Vaak ook worden algemene beperkende voorwaarden in de vorm van abstracte kennis gepresenteerd, zoals:

. op een willekeurige dag mag een employé slechts aan één project werken
. elke leverancier levert minstens twee onderdelen voor elk project

. een employé die aan project X werkt mag niet langer dan Y uur per maand aan project Z werken
. het salarisbudget van een maatschappij is de som van de salarisbudgetten van zijn afdelingen

Een door Z.W.O. gesteund onderzoek dat in onze vakgroep wordt uitgevoerd heeft betrekking op het geheim houden van geheimen als het databanksysteem aan de ene kant zo veel mogelijk vragen moet beantwoorden maar aan de andere kant geen geheimen prijs mag geven, daarbij gebruik makend van algemeen toegankelijke kennis, zoals de hier besproken abstracte kennis, en specifieke kennis van een gebruiker, die hij al vragend heeft verkregen [54].

3.2. Structuur van abstracte kennis

Abstracte kennis heeft betrekking op abstracties van individuele objecten. Het is uiteraard van belang deze kennis ook in de databank op te slaan. Daartoe is het noodzakelijk dat de abstractie van een verzameling objecten concreet wordt gemaakt.

In de terminologie van SMITH en SMITH [57] heet het abstraheren van individuele objecten classificatie ; hierbij wordt afgezien van de details die het object uniek maken, terwijl de eigenschappen die de objecten uit de collectie gemeen hebben naar voren worden gehaald. De laatste eigenschappen worden dan de eigenschappen van een nieuwe entiteit of type. Het voordeel is dat er zowel in intellectuele als in representationele zin efficiëntie is bereikt. Immers in plaats van bij elke leverancier te denken en te noteren dat hij aan elk project minstens twee onderdelen moet leveren kan dit nu één keer gebeuren.

Het onderkennen dat twee of meer verschillende types gezien kunnen worden als een ander nog algemener type is een vorm van abstractie die met de term generalisatie wordt aangeduid. Zo is de collectie van types: chauffeur, secretaresse en boekhouder te beschouwen als voorbeelden van het type employé. Allerlei eigenschappen van een employé kunnen nu éénmaal gedefinieerd worden, bijvoorbeeld dat alle employés één salaris hebben, dat ze allemaal één baas hebben, etc. Een bijkomend voordeel is dat door streng gebruik te maken van generalisatie, operaties streng gescheiden kunnen worden. Zo is er één salarisprogramma dat voor alle employées op dezelfde manier de sociale lasten en de belasting berekent, dat echter gebruik maakt van gegevens, die een speciaal voor de chauffeurs gemaakt programma bere-

kend heeft, voor de beloning van de overuren.

In een universiteit kunnen docenten en studenten ook als personen worden geadministreerd, die bijvoorbeeld allemaal het mededelingenblad ontvangen. Door een goed gebruik van het begrip generalisatie, kunnen de administraties van de studenten, die van de docenten, die van het overige personeel en die van alle bij de universiteit werkzame personen volledig van elkaar gescheiden worden, zodat privacyregels gehandhaafd kunnen worden.

Een ander belangrijk abstractiebegrip is aggregatie. Hier worden twee of meer typen bij elkaar gebracht om samen een nieuw type te vormen. Bijvoorbeeld ontstaat het nieuwe type lesrooster uit aggregatie van de types klas, les, zaal en tijdstip.

Volgens Smith en Smith zijn deze drie abstracties: classificatie, generalisatie en aggregatie de (enige) elementaire hulpmiddelen om structuur aan te brengen in het datamodel.

BRODIE [13] voegt hier nog het setbegrip aan toe, waarmee hij de mogelijkheid bedoelt om groepen objecten te kunnen aangeven. Een voorbeeld is een commissie van personen. De commissie heeft eigenschappen die de individuele personen niet bezitten, zoals een naam, een voorzitter en een secretaris. Het setbegrip kan als bijzonder geval van het begrip aggregatie worden beschouwd en is in zoverre interessant omdat het nauw aansluit bij het setbegrip van het DBTG datamodel.

Codd laat zien hoe de genoemde abstracties in het extented relationele model kunnen worden gerepresenteerd [19]. Wij zullen in het volgende hoofdstuk zien hoe dit met behulp van abstracte datatypen gerealiseerd kan worden.

3.3. Het relationele model

In 1970 introduceerde Codd het relationele datamodel. Concrete feiten betreffende de toepassingswereld worden voorgesteld door tupels in tabellen; de tabellen worden relaties genoemd. Elke relatie, zeg R, wordt gedefinieerd over een verzameling attributen X = {A1,...,An}, waarbij elk attribuut, Ai, op unieke wijze geassocieerd is met een domein, Dom(Ai), van waarden voor dat attribuut.

Een relatie R(X) bestaat aldus uit tupels (a1,...,an), waarbij ai een element is van Dom(Ai).

We nemen als voorbeeld een studentenadministratie, die we ook in de volgende paragraaf zullen gebruiken, waar tupels de volgende informatie vastleggen:

N: de naam van de student; we nemen aan dat de naam uniek is.
F: de faculteit waarin de student studeert; we nemen aan dat elke student slechts in één faculteit studeert.
M: de naam van de mentor van de student die zelf ook student is; er is één mentor per student.
V: de vakken die een student volgde.
C: de cijfers die hij daarbij haalde.
A: de accountnummers die de student als computergebruiker in gebruik heeft.
P: de paswoorden die hij hierbij gebruikt.

Voorbeeld:

STUDENT	N	F	M	V	C	A	P
	Jan	W&N	Piet	Alg	7	112	marie
	Jan	W&N	Piet	Mtk	8	112	marie
	Jan	W&N	Piet	Alg	7	303	mary
	Jan	W&N	Piet	Mtk	8	303	mary
	Piet	PHIL	Jan	WdW	4	000	000

Fig 1. De STUDENT relatie

We merken meteen op dat de relatie veel redundante gegevens bevat. De reden hiervoor is dat we een aantal zaken met elkaar gecombineerd hebben die niet zo veel met elkaar te maken hebben. Het is niet moeilijk om in te zien dat we de relatie kunnen splitsen in de volgende drie relaties:

ST	N	F	M
	Jan	W&N	Piet
	Piet	PHIL	Jan

,

TENT	N	V	C
	Jan	Alg	7
	Jan	Mtk	8
	Piet	WdW	4

,

COMP	N	A	P
	Jan	112	marie
	Jan	303	mary
	Piet	000	000

Fig 2. De relatie STUDENT gesplitst in ST, TENT en COMP

De oorspronkelijke relatie STUDENT kan weer gemakkelijk uit de relaties ST, TENT en COMP gereconstrueerd worden. We hebben hier te maken met een zogenaamde meervoudige afhankelijkheid (MVD), een begrip dat door FAGIN [22,23,24] aldus is aangeduid, maar waarvan schrijver dezes vindt [66,67] dat het beter met de term independency had kunnen worden aangeduid. De notatie hiervoor is:

N -->-> {V,C} | {A,P}

Fagin voerde dit begrip in als generalisatie van het begrip functionele afhankelijkheid (FD). Voorbeelden van FD's zijn:

N --> F
en N --> M.

De simpele betekenis hiervan is dat elke student slechts één faculteit en één mentor heeft. D.w.z. gegeven de waarde van N, dan zijn de waarden van F en M bepaald.

Om een datamodel te ontwerpen is het van groot belang de FD's en MVD's te kennen. Er zijn interessante theoriën ontwikkeld om uitgaande van een verzameling (basis) FD's alle FD's te vinden die er uit af te leiden zijn (zie ARMSTRONG [5] en BERNSTEIN [10]). De kunst is om de tabellen zo voordelig mogelijk te splitsen. De instrumenten voor de ontwerper zijn het groeperen van een aantal attributen in een relatie en het kiezen van een stel key-attributen per relatie. Met het laatste bedoelen we die attributen van een relatie die de tupels uniek bepalen, zoals N in ST.

Helaas moeten we diverse problemen signaleren:

1. Het splitsen is niet altijd uniek. Neem als voorbeeld dat we een student ook een geboortedatum G en een leeftijd L geven. We kunnen nu S = S(N,G,L) op twee manieren splitsen:

   S1 = S1(N,G) en T1 = T1(N,L)
   en S2 = S2(N,G) en T2 = T2(G,L)

   waarbij de tweede vermoedelijk de voorkeur zal hebben vanwege het jaarlijks bijwerken.
2. In bovenstaand voorbeeld hebben we de functionele afhankelijkheden:

   F1: N --> F {elke student in één faculteit}
   F2: N --> M {elke student heeft één mentor}
   F3: M --> F {ook elke mentor heeft één faculteit}

waaruit door gebruik te maken van transitiviteit de volgende FD volgt:

F4: N --> F {elke student heeft één faculteit van zijn mentor}

Volgens F1 hoort W&N bij Jan en volgens F4 hoort PHIL bij Jan, zodat F1 en F4 wezenlijk verschillend zijn. Het is echter niet eenvoudig formeel uit te maken dat F1 en F4 verschillend zijn.

3. Hoe representeer je andere semantische integriteitsregels, zoals bijvoorbeeld, dat een mentor direct of indirect niet zichzelf als mentor kan hebben, of dat een W&N student wel PHIL vakken mag (moet) volgen maar omgekeerd niet, zodat de combinatie

Kees PHIL Alg 5

als een foutieve kan worden herkend.

Voor nadere bijzonderheden moeten we naar de literatuur verwijzen en naar het volgende hoofdstuk.

4. DATABANKEN EN EEN PROGRAMMEERTAAL

4.1. Inleiding

In dit hoofdstuk zullen we een programmeertaal, PLAIN (Programming LAnguage for INteraction), bespreken waarmee het mogelijk is interactieve software te maken die o.a. ook geschikt is om systemen rond databanken te maken, zoals een personeelsadministratie of een ziekenhuisregistratiesysteem.

We zullen Abstract DataTypes (ADT's) gebruiken om daarmee de databank en de operaties er op te definiëren en om vast te leggen wie wat mag doen met de databank (toegangsprotectie).

We gebruiken het idee dat een programma in een omgeving werkt en daaruit expliciet objecten importeert en zelf ook weer een omgeving creëert voor andere programma's en objecten.

Na een introductie in de taal aan de hand van een programma dat op een Amerikaanse Universiteit zou kunnen worden gebruikt, willen we wat dieper ingaan op het gebruik van import, het creëren van objecten m.b.v. ADT's en hoe dit alles te gebruiken is om een protectieveilig systeem te maken. Tenslotte zullen we in dit hoofdstuk een geïntegreerd systeem bespreken op basis van de ingevoerde hulpmiddelen, dat op een Nederlandse Universiteit

met studenten en computerpersoneel zou kunnen draaien, en waarin enkele semantische integriteitseisen en protectieregels gecombineerd worden.

4.2. Een programmeertaal voor een databank

Om een databank te maken en ook te onderhouden heeft men programma's nodig die in een bepaalde programmeertaal zijn geschreven. De taalprimitiva die men in de programmeertaal nodig heeft hangen sterk af, althans zo is de situatie op dit moment, van de structuur van de databank. Zo hebben hiërarchisch databanken een eigen taal die voor het IMS systeem bestaat uit procedure-aanroepen vanuit een hogere (host) programmeertaal; de databank wordt beschouwd als hiërarchisch gestructureerde file.

De DBTG netwerk databanken beschikken over talen waarin het mogelijk is via de netwerkstructuur door de databank te "navigeren", om een term van BACHMAN [7] te gebruiken. Voor nadere details moeten we naar de literatuur verwijzen; zie bijvoorbeeld DATE [20].

Codd heeft zijn relationele datamodel voorzien van twee talen; de een is gebaseerd op operaties op de hele relaties, waarbij de programmeur precies moet aangeven welke operaties op welke relaties moeten worden toegepast, het heet de algebraïsche methode, de ander geeft de programmeur de mogelijkheid te definiëren aan welke voorwaarden de tupels moeten voldoen die hij selecteert en combineert. In dit laatste geval moet het systeem maar uitmaken welke operaties uitgevoerd moeten worden. Dit wordt de calculusmethode genoemd. PLAIN is primair gericht op de eerste aanpak, hoewel het ook zeer goed mogelijk is relaties of subcollecties van relaties te doorlopen op zoek naar bijzondere tupels.

In PLAIN gaan we uit van relaties met unieke tupels (dus geen duplicaten in de relatie); voor de definitie verwijzen we naar het vorige hoofdstuk. We nemen als voorbeeld de relaties ST en TENT van dat hoofdstuk, zie fig. 2.

Op één relatie zijn twee operaties gedefiniëerd, namelijk selectie en projectie, wat feitelijk neerkomt op een horizontale en een verticale uitsnijding van de relatie.

1. De selectie werkt als volgt:
   Stel we willen de gegevens van alle studenten die in de W&N faculteit studeren. In PLAIN schrijven we dan:

```
m1:= ST where F = 'W&N'
```

Het gevolg is dat m1 nu een deelcollectie van tupels van ST aanduidt. In veel relationele systemen, zoals bijvoorbeeld PASCAL/R [51], kent men alleen variabelen van het type relatie, zodat bovenstaand statement resulteert in het copiëren van de gekozen tupels uit ST. Om dit copiëren te voorkomen kan men in System R [6] m1 als een "view" definiëren, d.w.z. een recept dat nog niet wordt uitgevoerd, maar dat straks, als er echt resultaten moeten komen, wordt uitgevoerd. In de taal van System R, SQL genaamd, hadden we iets opgeschreven als:

```
LET M1 BE
    SELECT N M
    FROM TENT
    WHERE F = 'W&N'
```

Om het maken van copiën te voorkomen is het in PLAIN mogelijk m1 van het type marking te definiëren maar dan wel een marking die afhangt van ST, als volgt:

```
var m1: marking of ST
```

Er wordt een datastructuur gemaakt die het mogelijk maakt de geselecteerde tupels snel terug te vinden.

We zullen voor het vervolg aannemen dat de variabelen m1, m2, ..., m7 alle van het type marking zijn.

2. De projectie is een operatie waarmee we bepaalde attributen van een relatie of marking kunnen aanwijzen waarin we geïnteresseerd zijn. Zo zullen de tupels van m1 allen de waarde 'W&N' als F-attribuut hebben, zodat alleen de N en M attributen ons zullen interesseren. In PLAIN noteren we dit als volgt:

```
m2:= m1 => (N, M)
```

De projectie had ook met de selectie gecombineerd kunnen worden:

```
m2:= ST where F = 'W&N' => (N, M)
```

dan ontstaat in één klap de informatie welke mentoren welke studenten in de W&N faculteit begeleiden. Willen we alle mentoren kennen die studenten in de W&N faculteit begeleiden dan hoeven we slechts de volgende m3 te printen:

m3:= m2 => (M)

Overigens kunnen markings, in tegenstelling tot relaties, wel duplicaat-tupels bevatten. In het geval van m3 is dat ook zeer zinvol, omdat een mentor best meerdere keren kan voorkomen, hetgeen betekent dat hij of zij meerdere studenten in de W&N faculteit begeleidt.

3. De join operatie wordt tussen twee relaties of markings gedefinieerd. Stel we willen het verband weten tussen de vakken die een student heeft gevolgd en de faculteit waarin hij of zij studeert en dat voor alle studenten. Met andere woorden, gegeven een faculteit f en een vak v, zijn er studenten met naam n, mentoren m en cijfers c zó dat <n,f,m> element is van ST en <n,v,c> element is van TENT, waarbij de n in beide tupels dezelfde moet zijn. In PLAIN noteren we dit als volgt:

m4:= ST.N join TENT.N

Het resultaat van deze (algebraïsche) operatie is een marking bestaande uit combinaties van tupels (<n,f,m,v,c>) van ST en van TENT, zonder dat overigens weer copieën gemaakt worden. Als er een assignment aan een relatievariabele plaats vindt dan worden die copieën wel gemaakt.

De relatie STUDENT van paragraaf 3.3 zou als volgt geconstrueerd kunnen worden:

STUDENT:= m4.N join COMP.N

Wederom kunnen we de join met de projectie combineren:

m5:= ST.N join TENT.N => (F,V)

of als we meer in het verband tussen mentoren die W&N studenten begeleiden en de cijfers van hun studenten geïnteresseerd zijn, dan noteren we:

m6:= m2.N join TENT.N => (M,C)

Deze constructie, waarbij twee databankoperaties in één statement voorkomen, is in PLAIN de meest ingewikkelde die een programmeur kan opschrijven. Als hij bijvoorbeeld wil weten welke mentoren uit m6 studenten begeleiden met onvoldoende cijfers, dan noteren we dit als volgt:

m7:= m6 where C < 6 => (M)

In PLAIN is het niet toegestaan om alles in één keer op te schrijven. Het

is nodig dat de programmeur alle operaties stap voor stap voorschrijft. In een taal als SQL wordt het maken van ingewikkelde constructies aangemoedigd. De reden is dat de programmeur het aan het systeem moet overlaten om de operaties geschikt te kiezen, die dan gebruikmakend van allerlei inside-informatie de beste stategie kan kiezen. In SQL zou m7 als volgt opgeschreven moeten worden.

```
SELECT M
FROM ST TENT
WHERE ST.F = 'W&N'
  AND ST.N = TENT.N
  AND TENT.C < 6
```

Zou de programmeur de "view" M2 gebruikt hebben, dan had hij met het volgende statement hetzelfde resultaat bereikt:

```
SELECT M
FROM M2 TENT
WHERE M2.N = TENT.N
  AND TENT.C < 6
```

Het systeem zou overigens intern de M2 vervangen door de eerder gedefinieerde "view" en dezelfde statement gaan uitwerken als die we boven voor m7 vonden. De bedoeling van het systeem is namelijk om zo laat mogelijk tot daadwerkelijke uitvoering van het statement over te gaan. Dit maakt het namelijk aantrekkelijk om een optimalisator, die "query optimalisator" wordt genoemd, aan het werk te zetten . Deze zoekt de beste manier uit om de gestelde vraag te beantwoorden. In bovenstaand voorbeeld zal het er namelijk van afhangen of er extra toegangswegen beschikbaar zijn op de F en C attributen en wat de groottes zijn van de relaties ST en TENT. Op dit optimalisatieterrein wordt veel onderzoek verricht. We verwijzen hier naar PALERMO [47], WONG en YOUSSEFI [71] en GRIFFITHS [29].

In PLAIN is bewust een andere weg bewandeld. De reden is dat, hoewel het idee van de optimalisatie door het systeem uitstekend is, het optimalisatieproces zeer complex kan worden, met alle kwalijke gevolgen van dien. Bovendien weet de gebruiker natuurlijk heel wat van zijn databank af en hij kan die kennis inderdaad gebruiken, waarbij het systeem hem in zoverre tegemoet komt dat in het geval van een join het systeem zelf de slimste methode kiest. Tenslotte is PLAIN bedoeld voor een klein systeem, waarin zuinig met code omgesprongen moet worden.

4.3. Behandelen van excepties in PLAIN

Bij het schrijven van interactieve programma's moet men meestal verdacht zijn op bijzondere situaties, zoals fouten bij invoer en uitvoer (i/o error). In PLAIN kan men hierop anticiperen door het exceptiemechanisme. Onder een exceptie verstaan we een bijzondere situatie. Deze kunnen standaard zijn ontstaan door een i/o error of delen door nul, maar ze kunnen ook door de programmeur zelf worden ingevoerd, zoals stack-overflow of, zoals wij ze i.h.b. zullen gebruiken, protectiefouten.

Men kan de reactie R op een exceptie e in een statement S aangeven door achter het statement te schrijven:

```
![e:R]
```

waarbij e de naam is van de exceptie en en R de naam van een zogenaamde handler procedure. Bijvoorbeeld:

```
write i,j ![ioerr:reageer_op_io_error];
a:= b/c ![zerodivide:pas_op_c_is_nul];
stapel.push(st,x) ![stack_overflow:stapel_is_vol]
```

met de volgende, wellicht niet zo erg realistische handler procedures:

```
handler reageer_op_io_error;
begin clear {dit zet de exceptie af};
    {waarschuw de operateur}
end;
handler pas_op_c_is_nul;
imports c: modified {zie volgende paragraaf};
begin clear ; write 'c is nul'; c:=1;
    retry {hiermee wordt het statement waarin de
           exceptie optrad nog eens uitgevoerd}
end ;
handler stapel_is_vol;
begin write 'stapel is vol; weinig aan te doen';
    {door niet clear te doen wordt op een hoger
     niveau de exceptie ook gesignaleerd}
end;
```

Men kan zelf excepties aanzetten en er zoals boven beschreven mee werken door een signal statement. Bijvoorbeeld

```
signal stack_overflow
```

in de proceduredeclaratie van push:

```
if top >= size then signal stack_overflow
```

4.4. Import van variabelen en procedures binnen een programma

In PLAIN, zoals ook in sommige andere hogere programmeertalen als Euclid [37], kan men meer expliciet de scope van variabelen en procedures aangeven om de programmeur beter tegen zichzelf te beschermen. In principe is de scope van een variabele of procedure gedefiniëerd zoals in ALGOL 60, dus d.m.v. de geneste blokstructuur, echter als een globale variabele of procedure binnen een procedure wordt gebruikt dan moet hij expliciet d.m.v. een imports declaratie worden ingevoerd binnen die procedure. Bijvoorbeeld:

```
program P;
var i:integer;
    procedure Q;
    imports R: invoked ; i: readonly ;
    begin  ... i ... R ...  end ;
    procedure R;
    begin  ...  end ;
begin  ... i ... Q ... R ...  end
```

Ook kunnen bepaalde restricties worden aangegeven. Als we willen dat i niet in R maar wel in Q geïmporteerd mag worden, kunnen we dit als volgt aangeven:

```
var i:integer restricted to Q;
```

Mocht een procedure, zoals R, die zelf geen gebruik maakt van i, een andere procedure, bijvoorbeeld Q, aanroepen die wel i importeert, dan moet R hem ook importeren. Dit is nodig om de compiler uitspraken te kunnen laten controleren als: deze procedure laat de waarde van i onveranderd. Bijvoorbeeld kan de compiler signaleren dat in het volgende programma onbedoeld de waarde van i door R kan worden veranderd, hoewel dit niet de bedoeling was:

```
program P;
var i:integer restricted to Q;
    procedure Q;
    imports i: modified ;
    begin  ... i:= i+1; ...  end ;
    procedure R;
```

```
        imports Q: invoked ;
        var i:integer;
        begin .... i ... Q ... end ;
    begin ... R ... end.
```

Bovendien zien we, door de eis te stellen dat alle variabelen en procedures die direct of indirect worden gebruikt expliciet geïmporteerd moeten worden, dat de schijnbare ambiguïteit van de betekenis van i in de bovenstaande procedure Q wordt opgeheven omdat bij een goede declaratie van R, waarin i wel geïmporteerd wordt, de declaratie van de locale variabele op een foutmelding zal uitlopen.

Anticiperend op paragraaf 4.7., zouden we het importsmechanisme als volgt kunnen beschrijven:
Noem variabelen en procedures objecten. Een programma of een procedure P is in staat, d.m.v. declaratie, objecten te maken; het zijn precies die objecten die door het programma of de procedure zelf worden gedeclareerd en niet door locale sub-procedures. We zeggen dat al deze objecten P als vader hebben. Kennelijk heeft elk object één vader, namelijk het programma of de procedure waarin het gedeclareerd wordt.

De region van een programma of procedure bestaat nu uit die objecten waarvan hij de vader is en die objecten die door importering er aan zijn toegevoegd. We duiden de region van P aan met reg(P).

Het zijn precies de objecten in reg(P) die in P gebruikt mogen worden. Voor de procedure P geldt dat het zichzelf kan aanroepen, hoewel P zelf niet in reg(P) is opgenomen; we hebben dit gedaan om overeenstemming te krijgen tussen de importeringsregel hier voor procedures en variabelen binnen een programma en de importeringsregel voor objecten buiten een programma, een regel die we in een volgende paragraaf zullen bespreken.

De omgeving van een object O, aangeduid met omg(O), is de region van zijn vader.

De importeringsregel zegt nu dat een object O geïmporteerd kan worden in de region van een procedure P, als het in de omgeving van P zit, mits niet expliciet verboden d.m.v. een restrictie. Bij de importering wordt het aan reg(P) toegevoegd.

Voor het eerste programma in deze paragraaf geldt bijvoorbeeld:

de vader van i, Q en R is P
reg(P) = {i, Q, R}

reg(Q) = {i, R}
reg(R) = Ø
omg(Q) = {i, Q, R}
omg(R) = {i, Q, R}

Merk op dat het de compiler is die de feitelijke importering uitvoert, tijdens de syntactische analyse van het programma. Dit is in tegenstelling tot het importeren van objecten buiten een programma, zoals besproken zal worden in paragraaf 4.7., waar het een run-time operatie zal blijken te zijn.

4.5. Een voorbeeld van een programma

In deze paragraaf zullen we een programma demonstreren om een algemene indruk van PLAIN te krijgen. Het betreft operaties op een databank in een typisch Amerikaanse Universiteit. Het voorbeeld is overgenomen uit het PLAIN rapport [69]. Het geeft een goed inzicht in de mogelijkheden van PLAIN op het gebied van databankmanipulaties.

```
program honor_roll;
{This program makes use of a database of students, faculty, and
courses in a university. It produces a report of the names,
identification numbers, and grade point averages for all students
with a grade point average of 3.5 or better, grouped by
departmental major, for a specified set of majors.
Information on students is kept in relation student.
Information on courses is kept in relation course. Information on
student enrollment in courses, along with grades, is kept in
relation enroll.
Grades are stored in a field of two characters, an alphabetic
character optionally followed by a '+' or '-'. The value of
grades is as follows: 'A'=4, 'B'=3, 'C'=2, 'D'=1, and 'F'=0. A
'+' appended to a 'B', 'C', or 'D' adds 0.3 to the value; a '-'
appended to an 'A', 'B', 'C', or 'D' subtracts 0.3. Students may
also be assigned the grades of 'I', 'S', and 'U', which are not
included in the average. Courses in which the student is
presently enrolled will have an empty grade field. No courses may
be offered for a variable number of credits. A student may enroll
in a course for credit more than once.}
```

```
external {the following objects are imported from outside}
var student: relation [key idnum] of sinfo;
    course: relation [key dept, number] of cinfo;
    enroll: relation [key dept,  number,  section,  term, pupil]
                        of signup;
end external;
const ndept = 8 {number of departments to be checked on this run};
type  dabbr = char [4];
      cinfo = record
                dept: dabbr;
                number: char[4]
                  {1 to 3 digits, possibly followed  by a letter};
                title: string;
                credits: 0..8;
                descrip: string;
                incharge: string
              end record;
      sinfo = record
                idnum: char [8];
                name: string;
                address: string;
                college: char [2];
                year: 0..5
                  {0 for special student;  5  for  grad student};
                major: dabbr
              end record;
      signup = record
                 dept: dabbr;
                 number: char [4];
                 section: 1..60;
                 term: char [3] {sample: W80};
                 pupil: char [8];
                 grade: char [2]
               end record;
      var depts: array [1..ndept] of dabbr :=
        ('math','phys','chem','eecs','biol','hist','engl','econ');
      procedure honors (department: dabbr);
      {honors accepts a department name as input and computes the
```

```
grade point average for all students in that department.
For each student, it searches enroll to find the courses in
which that student is enrolled, computing a grade point
average for those courses where a letter grade of A,B,C,D,
or F is assigned.
 Relation good_student is created on each call to honors.}
imports student, enroll, course: readonly;
var ncred: integer {ncred is the total number of credits taken};
    entry: boolean;
    gpa, val, points: float;
    good_student: relation [key idnum] of
                        dnum: char [8];
                        name: string;
                        grade_avg: float
                  end relation;
    st: marking of student (idnum, name);
    temp: marking of enroll (dept, number, grade);
begin
   st := student where major = department => (idnum,name);
   foreach s in st
   loop <by_student> {this loop iterates over the tuples of st}
     {find that student's enrollments}
       temp := all enroll where pupil = s.idnum =>
         (dept,number,grade);
           {temp is a marking that holds student's courses and
           grades; duplicates are preserved since students may
           take a course more than once}
       ncred := 0 points := 0.0;  {initialize for each student};
       foreach t in temp
       {iterate over temp with attributes dept, number, and grade}
       loop <by_enroll>
           entry := true;
             {entry is true if there is an averageable grade}
           case t.grade[1] of
               when 'A': val := 4
               when 'B': val := 3
               when 'C': val := 2
               when 'D': val := 1
```

```
                when 'F': val := 0
                else: entry := false
            end case;
            if entry then
                if (val>=1) then
                    if val < 4 & (t.grade[2]='+') then
                       val := val + 0.3
                    else if (t.grade[2]='-') then
                       val := val - 0.3
                    end if
                end if;
                     {given the department and course number,
                     look in the relation course to find the
                     number of credits for that course,
                     totaling up the number of credits ncred}
                course% := course [t.dept,t.number];
                {course% now denotes the selected tuple}
                if course% ~= nil then
                    ncred := ncred + course%.credits;
                         {now add up the total number of grade
                         points, multiplying credits by grade
                         value}
                    points := points + course%.credits*val
                end if
            end if
        repeat <by_enroll>;
        gpa := points/ncred;
        if gpa >= 3.5 then
            {form a record with id number, name, and grade
            point average, then append it to relation
            good_student}
            good_student :+ [<s.idnum,s.name,gpa>]
        end if
    repeat <by_student>;
    foreach gs in good_student
    loop
        write gs.name,gs.idnum,gs.grade_avg,'\N';
    repeat;
```

```
      end honors;
begin
   foreach index increasing in [1..ndept]
   loop <by_dept>
       write depts[index], '\N';
       honors (depts[index])
   repeat <by_dept>
end honor_roll.
```

In de procedure honors hadden we ook een join kunnen gebruiken om een iets andere temp uit te rekenen en wel als volgt:

```
st:= student where major = department => (idnum,name);
temp:= enroll.pupil join st.idnum =>
          (idnum,dept,number,grade);
```

vervolgens moet de initialisatie voor alle studenten worden uitgevoerd in een korte loop, waarna in één repetitie, over alle tupels in temp, points en ncreds voor alle studenten (bijvoorbeeld in een tijdelijke relatie) berekend kunnen worden. Deze opzet is duidelijk ingewikkelder dan de structuur van het gedemonstreerde programma, dit demonstreert dat het niet altijd voordeliger is om de programmeur verre te houden van de elementaire operaties, zoals in SQL.

Een volgende opmerking betreft excepties die in het programma ontbreken. Als de programmeur er rekening mee wil houden dat delen door nul wel eens verkeerd kan gaan en dat een i/o operatie niet altijd goed afloopt, dan kan hij de volgende veranderingen aanbrengen in het programma: Het statement

```
gpa:= points/ncred
```

veranderen in:

```
gpa:= points/ncred ![zerodivide:pzero]
```

en het statement

```
write ...
```

veranderen in:

```
write ... ![ioerr:write_handler]
```

Verder moet hij handler procedures pzero en write_handler aan zijn programma toevoegen, bijvoorbeeld gedeclareerd zoals in de paragraaf over

excepties besproken is.

4.6. Abstracte Data Typen (modules)

Aan de hand van een standaardvoorbeeld voeren we nu het begrip Abstracte Data Type, ook wel "encapsulated data type" of module in. Het voorbeeld betreft de bekende datastructuur stapel, of stack. We definiëren het volgende type: (overgenomen uit het PLAIN rapport [69])

```
type stack [max:integer] = module
    exports
        function empty (x:stack): boolean;
        procedure push (val: integer -> x:stack);
        procedure pop (-> x:stack, val:integer);
        function top(x:stack): integer;
        exception stkfull, stkempty;
    rep record
            stktop: 0..max;
            elements: array [1..max] of integer;
        end record;
    ops
        function empty (x:stack): boolean;
        begin empty := x.stktop = 0 end;
        procedure push (val: integer -> x:stack);
        exception stkfull;
        begin
            if x.stktop >= max then signal stkfull
            else
                x.stktop := x.stktop + 1;
                assert (x.stktop >= 1) &  (x.stktop  <= max);
                x.elements[x.stktop] := val;
            end if
        end;
        procedure pop (-> x:stack, val:integer);
        imports stack.empty: invoked;
        exception stkempty;
        begin
            if stack.empty(x) then signal stkempty
            else
```

```
                    assert (x.stktop >= 1) &  (x.stktop  <= max);
                    val := x.elements[x.stktop];
                    x.stktop := x.stktop - 1;
                end if
            end;
            function top(x:stack): integer;
             { push and pop guarantee that 0<=stktop<=max }
            imports empty: invoked;
            exception stkempty;
            begin
                if empty(x) then signal stkempty
                else top:= x.elements[x.stktop]
                end if
            end;
    begin  { initialize stack }
            stktop := 0
    end module;
```

Iemand kan nu het volgende programma maken:

```
program P1;
type stack ... {zoals boven gegeven}
exception stkfull,stkempty;
var s:stack[1000]; t:stack[2000];v,w:integer;
handler stapel_is_vol;
begin write 'stapel is vol' end ;
handler stapel_is_leeg;
begin write 'stapel is leeg' end ;
begin
       ... stack.push(123,s) ![stkfull:stapel_is_vol]; ...
       ... stack.push(321,t) ![stkfull:stapel_is_vol]; ...
       ... stack.pop(s,v) ![stkempty:stapel_is_leeg]; ...
       ... stack.pop(t,w) ![stkempty:stapel_is_leeg]; ...
end.
```

We hadden ook een type stack uit de omgeving van het programma kunnen halen als het in een bibliotheek aanwezig zou zijn. In plaats van expliciet de declaratie uit te schrijven had het programma dan de volgende external declaratie moeten hebben:

```
program P2;
external type stack
end external ;
{en verder precies hetzelfde te beginnen bij de var declaratie}
```

Tenslotte hadden we ook, in plaats van de variabelen s en t in het programma te creëren, bestaande variabelen s en t uit de omgeving kunnen halen, vooropgesteld dat ze er zouden zijn. Het programma zou er dan als volgt uit gaan zien:

```
program P3;
external type stack;
    var s,t:stack;
    exception stkfull,stkempty
end external ;
var v,w: integer;
{en verder hetzelfde als in P2}
```

Het is natuurlijk niet zo verstandig om stapelobjecten te importeren omdat ze misschien ook wel door anderen gebruikt kunnen worden. Het kan echter gebeuren dat twee programma's juist door middel van een externe stapel willen communiceren. In principe kan dat dus. Bovendien gaat het ons er ook niet om met externe stapels te gaan spelen, maar wel om met externe objecten van het soort databank. Objecten die wel extern moeten zijn, want als ze alleen locaal binnen een programma zouden bestaan, waren ze niet zo erg nuttig; immers met het verdwijnen van het programma zouden ook zij verdwijnen. In de volgende paragraaf zullen we nader ingaan op het importeren van externe objecten. We willen nagaan hoe een onschuldige vorm van protectie ingebouwd zou kunnen worden als we een externe stapel gebruiken. Stel we hadden de regel dat een stapel van het soort stack alleen overdag gebruikt mag worden. We kunnen dan in de declaratie een exceptie

```
protectie_fout
```

opnemen en aan de procedures van stack de volgende test toevoegen:

```
if abs(time - 14) > 6 then signal protectie_fout
```

Alleen, het zou niet zo erg beveiligd zijn als de programmeur in zijn programma de standaardprocedure time (die geacht wordt de tijd in uren van 0 tot 24 aan te geven) zou kunnen herdeclareren. Daarom is het importeren,

niet alleen van het type, maar ook van de objecten zelf een eis voor een protectieveilig systeem.

Hoe we er voor kunnen zorgen dat een bepaald object, zoals de stapel s, alleen door een bepaalde gebruiker geimporteerd en gemanipuleerd kan worden, hetgeen met objecten als databanken en gebruikers als bankemployé's voor de handhaving van privacyregels van groot belang is, zullen we ook zien in een volgende paragraaf.

4.7. Het importeren van objecten van buiten een programma

In deze paragraaf gaat het over objecten die buiten een programma leven zoals files, standaardprocedures en databanken. In het bijzonder gaat het om de vraag hoe we tegen het importeren van deze objecten in programma's aankijken. Daarbij moeten we het begrip programma ook nog wat ruimer nemen dan gebruikelijk, omdat we in de volgende paragraaf Abstracte DataTypen (ADT's) zullen tegenkomen waarmee objecten gecreëerd kunnen worden die zich een beetje als een programma gedragen.

Objecten door ADT's gedefiniëerd hebben de volgende levenscyclus: Op het moment dat een object, zeg S, gecreëerd wordt door een ander object, zeg T, wordt een initialiseerprocedure uitgevoerd. Als gevolg daarvan worden wellicht andere objecten gecreëerd. De verzameling van deze objecten wordt region van S genoemd, aangeduid met reg(S), en voor elk van deze objecten is S de vader.

De verzameling reg(S) wordt verder uitgebreid met de objecten die geïmporteerd worden tijdens het uitvoeren van de initialiseerprocedure.

Als de initialiseerprocedure afgelopen is, blijft het object S bestaan en tevens al de objecten die op dat moment in reg(S) zitten.

Uitvoering van operaties op S kan tot gevolg hebben dat er nieuwe objecten aan reg(S) worden toegevoegd, of dat er objecten uit verdwijnen. Het is dus duidelijk dat reg(S) een nogal dynamisch variërende verzameling van objecten is. Pas als de vader van S, T dus, besluit om S op te heffen of zelf opgeheven wordt, verdwijnt S en ook alle objecten waarvan hij de vader is, verdwijnen daarmee. Merk op dat we niet zeggen: "alle objecten van reg(S)", het kan immers zijn dat sommige van die objecten van elders geïmporteerd zijn.

In het vervolg zullen we een werkende procedure, aangeroepen tijdens de initialiseringsfase van een object of als operatie, een proces noemen.

Evenals in de voorgaande paragraaf, definiëren we ook nu weer wat de omgeving is van het object S, aangeduid met omg(S), zijnde de region van

zijn vader T. Dus

$$omg(S) = reg(vader(S)).$$

Omdat er tenslotte één stamvader moet zijn, voeren wij deze ook in onder de naam ALPHA; voor dit object geldt dat hij vader van zichzelf is. Het zou ook als operating systeem kunnen worden aangeduid.

Tenslotte kunnen we het importeren van een object O door een object S definiëren als het invoeren van het object O binnen reg(S), onder de voorwaarde dat, op het moment van invoer, O in omg(S) moet zitten.

Als O op het betreffende moment niet in omg(S) zit, dan zal de verdere executie van de operatie (initiëel of niet) worden opgeschort. Wat er daarna moet gebeuren is voor ons nog niet duidelijk. Een foutmelding is een eenvoudige mogelijkheid; wachten tot het object er is biedt echter ook interessante perspectieven. We wijzen hier op relaties met het monitor-begrip van HOARE [30].

Naast de bekende operaties op files (lezen en schrijven), programma's op files (uitvoeren) en standaardprocedures (aanroepen) kan een proces van een object S met de objecten O die in zijn region zitten de volgende handelingen uitvoeren:

- Als O een ADT is kan hij een object van dit type creëren, waarmee hij alle in de ADT aangegeven operaties kan uitvoeren. Dit object komt dan in reg(S) terecht.
- Als O een object is, gecreëerd door een ander object, dan kan het proces er alle operaties op toepassen die in het bijbehorende ADT zijn gedefiniëerd.
- Als O een relatie is dan kunnen er alle bekende relatie-operaties op worden toegepast, zoals selectie, projectie en join, maar ook veranderingen van de relatie (updates). Er wordt hier dus geen bijzondere protectie gegeven. Die moet gedefiniëerd worden op een niveau hierboven.

De enige protectie die door het systeem wordt gegeven is een consequent toepassen van de importeringsregel: alleen datgene wat in de directe omgeving zit mag geïmporteerd worden.
Wij hebben het vermoeden dat dit protectiemechanisme voldoende is om veilige systemen te ontwerpen.

Door Z.W.O. medewerker drs. M.L. Kersten wordt momenteel een mathematisch model onderzocht waarin het begrip "veiligheid" nader gedefiniëerd wordt, zodat enerzijds duidelijker wordt wat veiligheid betekent en zo dat

het anderzijds mogelijk wordt van programma's, die van de hier beschreven concepten gebruik maken, vast te stellen of ze inderdaad veilig zijn.

Om een indruk te krijgen van de hier geboden mogelijkheden zal er in de laatste paragraaf van dit hoofdstuk een uitgebreid voorbeeld van een systeem met te beveiligen personeelsadministraties en studenten, die graag het systeem willen breken, worden behandeld.

4.8. De beschermde stapel

We zullen in deze paragraaf nagaan hoe, met het simpele protectiemechanisme van de vorige paragraaf, het gebruik van de stapel s, geïntroduceerd in paragraaf 4.6., afdoende te beschermen is.

Stel we willen de stapel s, voorkomend in een bibliotheek, zo definiëren dat de operatie push alleen gebruikt kan worden door een gebruiker met de naam "John", terwijl een andere gebruiker, met de mooie naam "Mary", de operatie pop mag gebruiken en we willen het systeem zo structureren dat ze elkaars operaties niet kunnen gebruiken. De stapel wordt hier in een typische producer-consumer situatie gebruikt.

Zowel John als Mary worden geacht achter een terminal te zitten. Ze worden eenmalig geïdentificeerd door informatie op te vragen die alleen zij worden verondersteld te kennen, zoals een geboorteplaats en de meisjesnaam van de moeder (dit is in de USA een standaardmethode voor identificatie, naast uiteraard een handtekening).

We nemen hier aan dat Mary niet over de kennis beschikt om zich te vermommen als John.

Verder nemen we aan dat John en Mary programma's op een file hebben staan onder de namen John's_program, respectievelijk, Mary's_program.

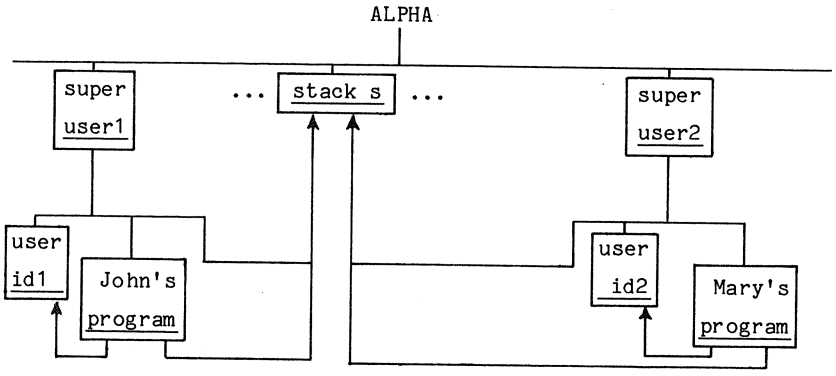
De idee is nu om een abstract data type (ADT) user te definiëren waarmee objecten gemaakt kunnen worden, die we userid 's zullen noemen, die de gebruiker identificeren en wel zo dat we in stack kunnen vragen of de gebruiker van s inderdaad John is.

Tijdens het creëren van een userid is er communicatie nodig tussen het nieuw gecreëerde user object en de gebruiker achter de terminal om de identificatie uit te voeren. Bovendien is er een databank met informatie over de gebruikers. Ook deze databank wordt geacht een object te zijn, gedefiniëerd in een ADT, userdb genaamd. Het betreffende object geven we de naam udb. In userdb treffen we als interne representatie een relatie aan waar alle gegevens in zitten. Omdat alle objecten m.b.v. een ADT

gedefiniëerd zijn, kan men slechts voorgeprogrammeerde operaties op die objecten toepassen (ook dit is een wezenlijk aspect van de hier voorgestelde beveiliging).

Tenslotte is er in ons model nog een ADT superuser. Als een gebruiker inlogt, verwachten we dat het operating systeem, in de vorige paragraaf ALPHA genoemd, een object van het type superuser creëert. Dit object, uniek voor elke gebruiker, zal eerst alle objecten van ALPHA importeren, het zal vervolgens een userid voor de gebruiker creëren en tenslotte het programma van de gebruiker in werking zetten.

Dit programma zal dus werken in een omgeving waar alle objecten van ALPHA én de zojuist gecreëerde userid inzitten en niet meer. Merk op dat we gezegd hebben dat er voor elke gebruiker een superuser gecreëerd wordt en dat deze, op zijn beurt, een unieke userid creëert. We zijn er dus zeker van dat in de omgeving van John's program de userid van Mary niet voorkomt en omgekeerd. Dit betekent dat Mary's program de userid van John niet kan importeren. In een plaatje ziet het er als volgt uit:



Zowel John's als Mary's programma kunnen dus de stack s importeren; maar, zoals opgemerkt, kunnen ze elkaars userid niet importeren.

We zullen nu eerst zien hoe John's_program er uit ziet:

```
program John's_program;
external type stack, user;
  var s:stack; u:user;
  exception stkfull,stkempty,access_violation;
end external ;
handler stapel_is_vol;{zie vorige paragraaf}
handler stapel_niet_beschikbaar;
```

```
begin clear ; write 'probeer een andere stapel' end ;
var v,w:integer;
begin
      ... stack.push(u,123,s)![stkfull:stapel_is_vol;
                  access_violation:stapel_is_niet_beschikbaar];
      ... stack.push(u,234,s)![stkfull:stapel_is_vol];
                  access_violation:stapel_is_niet_beschikbaar];
      ...
end.
```

Het programma is bijna hetzelfde als het programma P3 in de vorige paragraaf, alleen er is één opmerkelijk verschil, namelijk, de operaties push en pop hebben er een parameter bijgekregen: u, de geïmporteerde userid. Merk verder op dat Mary's program er eender uit zou kunnen zien, en dat ze met name dezelfde identifier voor de door haar programma geïmporteerde userid moet kiezen. De reden is dat elke superuser dezelfde identifier voor de gecreëerde userid's kiest. Dit zou ook best anders kunnen, bijvoorbeeld dat de gebruiker eerst aan de superuser opgeeft onder welke naam hij zijn userid wil kennen, zoals je dat ook kunt doen met files. Wij hebben hier echter voor de eenvoudigste opzet gekozen waarbij opzettelijk alles zo openbaar mogelijk is om te benadrukken dat de enige beschikbare protectie de importeringsregel is.

Alvorens de ADT user te geven, laten we eerst zien hoe de protectie van de stack er uit ziet. De definitie is bijna dezelfde als in de vorige paragraaf. De verschillen zijn dat er ten eerste een exceptie access_violation bij is gekomen en dat, ten tweede, de procedures er allen een extra parameter bijgekregen hebben: u van het type user, waarop dan in de body getest wordt. Het type user wordt expliciet geïmporteerd als extern type. Wij laten hier alleen de gemodificeerde push operatie zien.

```
type stack [max:integer] = module
exports {hetzelfde als in de vorige paragraaf plus}
   exception access_violation
extern type user
end extern ;
rep {hetzelfde als vroeger}
ops
  procedure push(u:user;val:integer->x:stack);
```

```
    imports user,identification : invoked ;
    exception stkfull, access_violation;
    begin
      if user.identification(u)~='John' then
        signal access_violation
        else if x.stktop >= max then signal stkfull
        else
           x.stktop:=x.stktop + 1;
        assert x.stktop>=1&x.stktop<=max;
           x.element[x.stktop]:= val
        end if
      end if
    end ;
    {...}
  end module ;
```

We zien dus dat er een operatie identification van user wordt gebruikt door stack. Deze operatie moet voor gebruiker John de string 'John' opleveren en voor Mary de string 'Mary', terwijl andere gebruikers weer andere strings krijgen, zoals de baas van het rekencentrum de string 'computer_boss'.

We gaan nu in op de definities van user en userdb. De belangrijkste fase in het leven van een object van type user, een userid dus, is ongetwijfeld het begin. Dan namelijk wordt vastgesteld of de terminalgebruiker gerechtigd is om de computer te gebruiken en er wordt vastgesteld wie hij is. Daartoe moet user een check operatie van userdb gebruiken omdat immers de informatie over gebruikers is opgeslagen in udb, een object van type userdb. Zowel userdb als udb moeten dan ook geïmporteerd worden.

Er is nog een object dat geïmporteerd moet worden, namelijk de zeer speciale userid uu. Wat is er aan de hand: Om vast te stellen dat de gegevens die een gebruiker verstrekt correct zijn, moet user de operatie check aanroepen van userdb, maar deze operatie is ook beschermd en wil alleen maar gegevens verstrekken aan programma's die een userid meegeven, waarvan de identificatie de string 'ALPHA' oplevert. Dus user moet ook zelf beschikken over een userid. Uiteraard kan deze niet op de standaardmanier gemaakt worden, omdat je dan in een vicieuze cirkel zit. We nemen daarom het bestaan aan van een object, genaamd uu, dat er altijd al was (net zoals het operating systeem ALPHA trouwens) en waarvoor geldt dat

```
          user.identification(uu) = 'ALPHA'
```

De definitie luidt nu aldus:

```
type user = module
exports function identification(v:user):string;
exception unsuccessful_login;
external type userdb;
var udb:userdb; uu:user
end external ;
rep nme:string end record ;
function identification(v:user):string;
begin identification:= v.nme end ;
procedure init(->n:string);
  imports userdb.check: invoked ;
  var c:integer:= 0;
    un,ug,mmn:string;
  begin
    loop
      if c = 0 then write
       'type naam, geboorteplaats en moeders meisjes naam'
      else write 'probeer het nog een keer'
      end if ;
      read un,ug,mmn;
      if userdb.check(uu,udb,un,ug,mmn) then exit
      else c:=c+1; if c>4 then exit end if
      end if
    repeat ;
      if c>4 then signal unsuccessful_login
      else nme:=un
    end if
  end init;
begin init (nme) end module ;
```

De ADT user hangt nog slechts af van de ADT userdb, zodat het een goed idee is om de definitie hiervan nu te geven. Straks komt superuser dan aan de beurt.

In userdb vinden we de echte databank in de vorm van de relatie user-rel. In de practijk zullen er een groot aantal operaties in userdb

gedefiniëerd zijn, zoals: check, voeg_een_gebruiker_toe, laat_een_gebruiker_weg, verander_iets, inspecteer_geboorteplaats, etc. Al deze operaties zullen ingebouwde protectieregels hebben. Wij zullen slechts de eerste twee laten zien om de definitie van user geheel compleet te hebben.

```
type userdb = module
exports
  function check(u:user;d:userdb;n,g,m:string):boolean;
  procedure add_new_user(u:user;n,g,m:string->d:userdb);
  exception access_error,wrong_addition;
external type user
end external ;
rep relation [ key name] of
       name, birthplace, mothers_maiden_name:string
         ...
      end relation;
ops
  function check(u:user;d:userdb;n,g,m:string):boolean;
  imports user.identification: invoked ;
  begin
    if user.identification(u)~= 'ALPHA' then check:=false
    else d%:= d[n];
      if d% ~= nil then
      check:=(d%.birthplace = g)&(d%.mothers_maiden_name = m)
      else check:= false
      end if
    end if
  end check;
  procedure add_new_user(u:user;n,g,m:string->d:userdb);
  imports user.identification: invoked ;
  begin
    if user.identification(u) ~= 'computer_boss' then
      signal access_error
    end if ;
     if d[n] = nil then
      {check semantische integriteitseisen}
       d:+ [<n,g,m>]
      else signal wrong_addition
```

```
      end if
    end add_new_user;
    begin userdb:+[<'ALPHA','omega','beedhe'>];
      userdb:+[<'computer_boss','tjaard','jitske'>]
      {deze tupels worden in userdb gezet om te voorkomen dat
       iemand anders met deze namen in userdb kan komen}
    end module ;
```

Tenslotte definiëren we de ADT superuser. Een object van dit type wordt door het operating systeem gecreëerd als zich een nieuwe gebruiker aandient. Het nut van de superusers is hoofdzakelijk dat ze een voor elke gebruiker eigen omgeving creëren, waarin alle objecten en typen en files en standaardprocedures geïmporteerd worden, zodat het gebruikersprogramma ze ook kan importeren, maar waar geen andere userid's geïmporteerd worden dan alleen de speciale userid uu. Mary's_program kan dus niet bij John's userid komen.

```
type superuser = module
  exports exception access_threat;
  external type user,stack,userdb,...;
    var s:stack;udb:userdb;uu:user;...;
    file John's_program, Mary's_program,...;
    exception unsuccessful_login;
  end external ;
  {geen rep gedeelte}
  procedure try_to_run;
  imports improper_usage: invoked ;
  var u:user![unsuccessful_login:improper_usage];
    {nu bestaat de userid als de login succesvol is geweest}
    s:string;
  begin
    write 'naam van het te executeren programma';
    read s;
    {de volgende operatie bestaat in PLAIN nog niet maar is
     voor dit soort spelletjes onontbeerlijk}
    execute (s)
  end module ;
```

De oplettende lezers hebben inmiddels misschien opgemerkt dat de omgeving

van Mary's_program de objecten udb en uu bevat. Er is dus schijnbaar geen bescherming van udb, want Mary zou udb en uu kunnen invoeren. In feite valt het wel mee. Immers de enige userid's die Mary's program kan meegeven aan udb zijn haar eigen userid, maar daar kan ze alleen de operatie pop mee uitvoeren, of uu, maar de enige operatie die daarmee uit te voeren is, is check. Dus Mary is in staat te checken of haar eigen gegevens juist zijn. Wel, dat is een plezierige bijkomstigheid. Er zijn landen waar de privacyregels zo zijn dat dat nu juist mogelijk moet zijn. Mary zou systematisch geboorteplaatsen en moeder's meisjes namen in haar programma kunnen genereren om aldus achter de gegevens van John te komen. Deze situatie is volkomen vergelijkbaar met de wijze waarop in UNIX de paswoorden worden beveiligd. Geïnspireerd door de vuistregel dat files toch niet te beschermen zijn, al was het alleen maar doordat iemand een stuk papier laat slingeren, hebben de ontwerpers de paswoordfile volledig publiek bezit verklaard. Echter om de paswoorden zelf te lezen moet men een algoritme toepassen die zo langzaam is dat men practisch kan zeggen dat hij niet bestaat [45].

Overigens, als men het toch wat gevaarlijk vindt om udb aan alle gebruikers te geven, kan men de ADT voor superuser iets wijzigen. We gebruiken de operatie disconnect a; die het geïmporteerde object a uit de region van het programma verwijdert. De body van de procedure try_to_run zou dan kunnen beginnen met het statement

```
disconnect udb,uu;
```

In feite was het object uu ingevoerd om de udb ook echt te beschermen, zoals hier boven in de tweede versie aangegeven. Zouden we het alleen met de eerste versie doen, dan kunnen we het ook zonder uu af, alleen dan kan userdb.check geen controle uitoefenen op zijn eigen gebruik.

Aan het einde van deze paragraaf gekomen, constateren we dat het mogelijk is gebleken met de hulpmiddelen die PLAIN nu heeft, i.c. module = ADT, aangevuld met een eenvoudige importeringsregel voor externe objecten, een systeem te maken waarin èn de protectieregels in de modulen zelf worden ingebouwd èn de definitie van de userid's op simpele wijze mogelijk is. We verwachten in staat te zijn om de veiligheid van zo'n systeem inderdaad te kunnen bewijzen. In ieder geval geven we in de volgende paragraaf een voorbeeld waarin de semantische integriteitsregels en de protectieregels volledig zijn geïntegreerd. Het betreft een uitwerking van het voorbeeld

van paragraaf 3.3 uitgebreid met een persoonsadministratie en een aantal regels.

Wat betreft de implementatie van de importeringsregel merken we op dat deze simpel kan zijn. Immers het is voldoende van ieder object bij te houden wie zijn vader is, en in twee lijsten wie zijn zonen zijn en welke objecten geïmporteerd zijn. De beide lijsten zullen i.h.a. kort zijn omdat ze expliciet in de ADT moeten worden vermeld; namelijk in de vorm van declaraties en de lijst van de externals. Onze methode maakt het mogelijk om vrijwel de gehele protectie bij de modules zelf te leggen, hetgeen in tegenstelling is tot bijvoorbeeld een capability systeem, zoals gebruikt in HYDRA [33,72], waar de protectie geheel door (de kernel van) het operating systeem wordt verzorgd. Een vergelijking wat betreft efficiëntie kunnen we hier niet geven; we zijn echter overtuigd dat onze methode veel flexibeler is.

Een methode die het ook mogelijk maakt dat protectie in de modules zelf wordt gedefinieërd is gebruik te maken van door het operating systeem gedefinieerde en gecreëerde userid's. Door de compiler wordt dan elke aanroep van een systeemprocedure door een gebruikersprogramma veranderd door er een extra actuele parameter, namelijk de userid van deze gebruiker, aan toe te voegen. Dit vereist dat er een extra elementair type "user" in PLAIN bijkomt. Het is dan niet meer mogelijk (nodig) om een ADT user te definiëren. Het operating systeem gebruikt een vaste procedure om een gebruiker te identificeren. Het voordeel is dat we dan de importeringsregel niet nodig hebben. Wij zijn van mening dat ook dit leidt, zij het minder dan de zojuist besproken methode, tot een weinig flexibel systeem.

Overigens, het moge duidelijk zijn uit de voorgaande opmerkingen dat nadere studies over fundamentele vragen (wat is veilig?) en implementatieproblemen en ook vergelijkingen met andere systemen noodzakelijk zijn.

4.9. Een voorbeeld van een beveiligd systeem

Als slot van dit hoofdstuk willen we in deze paragraaf een voorbeeld geven waarin alle mechanismen die we besproken hebben worden toegepast: ADT's, geprogrammeerde databankmanipulaties, semantische integriteitseisen, abstractie in de vorm van generalisatie, protectie en privacy.

Het voorbeeld is een stukje van een zeer hypothetisch databanksysteem in een universiteit. Gedeelten hebben we reeds eerder gezien.

De universitaire administratie bestaat uit een aantal afdelingen, waarvan wij er een aantal zullen bespreken. Personeelsleden van deze afdelingen, aangeduid met de term medewerker, hebben alle toegang tot de computer en tot die delen van de databank, waartoe zij, volgens de onderstaande specificaties, gerechtigd zijn. Daartoe worden afdelingsidentifiers gebruikt, zoals de identifier 'John' in de vorige paragraaf, nu echter niet van type string maar van scalartype dept. Ook andere personen hebben toegang tot de computer, zoals studenten.

Wat betreft de notatie: R(A,B,C) betekent de relatie met attributen A, B en C, waarvan A de key-attribuut is. De regels voor de diverse afdelingen zijn de volgende:

STafd met identifier ST: medewerkers kunnen de gegevens van studenten, opgeslagen in ST(N,F,M,P_nr), bekijken en veranderen. N is de unieke naam van de student, F de faculteit en M de mentor van de student. P_nr is het door PERSafd uitgedeelde unieke persoonsnummer van de student.

De eisen zijn dat elke student ook in PERS zit (m.a.w. PERS is een generalisatie van ST) en dat geen mentor, direct of indirect, mentor is van zichzelf.

TENTafd met identifier TENT: medewerkers kunnen de tentamenresultaten van studenten opgeslagen in TENT(N,V,C), bekijken en veranderen. N is de naam van de student, V de naam van het vak en C het door de student behaalde cijfer.

De eisen zijn dat student met naam N moet bestaan in ST en dat het vak in combinatie met de faculteit van de student toegestaan is. Er is een relatie TVFC(F,V) die de Toegestane Vak-Faculteit Combinatie aangeeft. Verder mag een TENT medewerker de faculteit opvragen van een student in ST.

COMPafd met de identifier COMP: medewerkers mogen de gegevens over het computergebruik, opgeslagen in COMP(P_nr,PW,PV), bekijken en veranderen. P_nr is het door de PERSafd uitgegeven persoonsnummer van de persoon en PW is diens paswoord. De betekenis van PV wordt later besproken.

De eisen zijn dat P_nr moet bestaan; verder mag een COMP medewerker de afdelingsidentifier en het adres van een persoon uit PERS opvragen. Voor een willekeurige persoon van de universiteit geldt verder dat, als hij in COMP zit, hij zijn paswoord mag veranderen. Echter, om eventueel wangedrag op te kunnen sporen, wordt in een apart veld, PV, van COMP

bijgehouden hoe vaak hij dit deed. Bovendien wordt de verandering alleen dan geëffectueerd als de persoon achter de terminal zijn huidig paswoord kan opgeven.

PERSafd met identifier PERS: medewerkers kunnen gegevens over alle personen, werkzaam of studerend bij de universiteit, opgeslagen in PERS(P_nr,N,AFD,ADR), bekijken en veranderen. P_nr is het uniek bepaalde persoonsnummer, N is de naam, AFD de afdelingsidentifier van de afdeling waar de persoon werkt, waarbij voor studenten die niet in een afdeling werken, STUD wordt ingevuld (we gaan hier niet in op het probleem dat iemand in meer dan één afdeling kan werken) en ADR is het adres van de persoon.

De eisen zijn dat elke persoon ook voorkomt in minstens één van de volgende databanken: ST, DOC en TAS, waarbij de laatste twee hier niet verder behandeld zullen worden.

Om medewerkers van de afdelingen te identificeren, gebruiken we de boven aangegeven identifiers. Soms zijn ook interne procedure-aanroepen nodig, als bijvoorbeeld de procedure update van de STADT de procedure update van de PERSADT aanroept. Ook dan vindt er uiteraard access-controle plaats. In plaats van de echte userid door te geven, wordt een speciale userid doorgegeven die uniek is voor elke ADT. Dit heeft het voordeel dat een procedure ook nog weet vanwaar hij aangeroepen wordt. De afdelingsidentifier is in zo'n geval: STop, TENTop, COMPop, PERSop, DOCop en TASop. Zoals we zagen hebben studenten als identifier STUD. Al deze identifiers samen vormen de scalartype dept.

Om ruimte te besparen zijn de procedures, inspect, add, delete en change samen genomen tot één procedure met extra parameter op. Het resultaat is helaas minder fraai dan wanneer ze voluit en apart geschreven zouden zijn. Wat betreft excepties hebben we er ons ook maar met een Jantje van Leiden afgemaakt: in plaats van verschillende excepties voor elke bijzondere toestand, hebben we slechts twee excepties: error voor een overtreding van een semantische integriteitsregel en access_violation voor een protectiefout.

```
type dept = (ST,TENT,COMP,PERS,DOC,TAS,
             STop,TENTop,COMPop,PERSop,DOCop,TASop,STUD);
     ops = (add,delete,change,inspect);
```

```
type STADT = module
exports
  procedure proc(op:ops;u:user->s:STADT;n,f:string);
  exception error, access_violation
external
  type dept, ops, user,PERSADT;
  var STuserid:user; PERSDB: PERSADT;
end external
type mrkt= relation [ key Name] of
      Name, Mentor:string
end relation ;
rep record STREL: relation [ key Name] of
        Name:string;
        Mentor:string;
        Faculteit:string;
        P_nr:integer   { Persoonsnummer }
      end relation
    end record
ops
  function Cycle_in_graph(m:mrkt):boolean;
  { controleert relaties op cycles }
  procedure proc(op:ops;u:user->s:STADT;n,f:string);
  imports user.identification,PERSADT.proc,
    Cycle_in_graph: invoked ;
    STuserid, PERSDB: readonly ;
  exception error, access_violation ;
  var ui:dept {user identification};
    pn:integer {persoonsnummer};
    om,a: string {oude mentor, adres};
    mrk: mrkt;
  begin ui:= user.identification(u);
    if ~ (op = inspect & ui in [ST,TENTop,PERSop] | ui = ST) then
      signal access_violation
    end if;
    s.STREL%:= s.STREL[n];
    case op of
    when inspect:
      if s.STREL% = nil then signal error
```

```
        else
          if ui = ST then m:= s.STREL%.Mentor else m:= '' end if ;
          f:= s.STREL%.Faculteit
        end if
      when add:
        if s.STREL% ~= nil ¦ s.STREL[m] = nil then
          signal error
        else write 'address please'; read a;
          {vraag een persoonsnummer pn aan}
          PERSADT.proc(add,STuserid,PERSDB,pn,n,STUD,a);
          s.STREL:+ [<n,m,f,pn>];
        end if
      when delete:
        if s.STREL% = nil then signal error end if;
        {Pas op dat deze student geen mentor is}
        mrk:= STREL where Mentor = n =>(Name, Mentor);
        if mrk ~= [] then signal error
        else PERSADT.proc(delete,STuserid->PERSDB,s.STREL%.P_nr,n,ui,a);
          s.STREL:- s.STREL%
        end if
      when change:
        if s.STREL% = nil ¦ s.STREL[m] = nil then signal error
        else
          om:=s.STREL%.Mentor;
          s.STREL%.Mentor:= m;
          mrk:= s.STREL => (Name,Mentor);
          if Cycle_in_graph(mrk) then
            s.STREL%.Mentor:= om; signal error
          else s.STREL%.Faculteit:= f
          end if
        end if
      else : signal error
    end case
    end proc;
end module ;

type TENTADT = module
exports
```

```
  procedure proc(op:ops;u:user->t:TENTADT;n,v:string;c:integer);
  exception error,access_violation;
external
  type dept, ops, user,STADT;
  var TENTuserid:user; STDB: STADT;
end external
rep record TENTREL: relation [ key N,V] of
          N:string;  {naam}
          V:string;  {vak}
          C:integer  {cijfer}
        end relation ;
      TVFCREL: relation [ key F,V] of
          F:string;  {faculteit}
          V:string   {vak}
        end relation
      end record
ops
  procedure proc(op:ops;u:user->t:TENTADT;n,v:string;c:integer);
  imports user.identification,STADT.proc: invoked ;
    TENTuserid, STDB: readonly ;
  exception error,access_violation;
  handler strange_love;
  begin clear ; write 'deze procedure mag niet worden aangeroepen'
  end strange_love;
  var m,fac:string;
  begin
    if user.identification(u) ~= TENT then
      signal access_violation
    end if ;
    t.TENTREL%:= t.TENTREL[n,v];
    case op of
    when inspect:
      if t.TENTREL% = nil then signal error
      else c:= t.TENTREL%.C
      end if
    when add:
      if t.TENTREL% ~= nil then signal error
      else STADT.proc(inspect,TENTuserid->STDB,n,m,fac)
```

```
            ![access_violation,error:strange_love];
          if t.TVFCREL[fac,v] = nil then signal error
          else t.TENTREL:+ [<n,v,c>]
          end if
        end if
      when delete:
        if t.TENTREL% = nil then signal error
        else t.TENTREL:- t.TENTREL%
        end if
      when change:
        if t.TENTREL% = nil then signal error
        else STADT.proc(inspect,TENTuserid->STDB,n,m,fac)
              ![access_violation,error:strange_love];
          if t.TVFCREL[fac,v] = nil then signal error
          else t.TENTREL%.C:=c
          end if
        end if
      else : signal error
      end case
  end proc;
begin
  {hier moet de relatie TVFCREL een initiële waarde krijgen}
end module ;

type COMPADT = module
exports
  procedure proc(op:ops;u:user->c:COMPADT;
    pn:integer;pw:string;pv:integer);
  procedure change_password(c:COMPADT;pn:integer);
  exception access_violation, error;
external
  type dept, ops, user,PERSADT;
  var COMPuserid:user; PERSDB: PERSADT;
end external
rep
  record COMPREL: relation [ key P_nr] of
          P_nr:integer;  {Persoonsnummer}
          PW:string;     {Paswoord}
```

```
          PV:integer;     {Aantal Paswoord Veranderingen}
        end relation
  end record
ops
  procedure proc(op:ops;u:user->c:COMPADt;
    pn:integer;pw:string;pv:integer);
  imports user.identification,PERSADT.proc: invoked ;
  exception error, access_violation;
  begin c.COMPREL%:= c.COMPREL[p];
    if user.identification(u) ~= COMP then
      signal access_violation
    end if;
    case op of
    when inspect:
      if c.COMPREL% = nil then signal error
      else pw:= c.COMPREL%.PW; pv:= c.COMPREL%.PV
      end if
    when add:
      if c.COMPREL% ~= nil then signal error
      else c.COMPREL:+ [<pn,pw,0>]
      end if
    when delete:
      if c.COMPREL% = nil then signal error
      else c.COMPREL:- c.COMPREL%
      end if
    when change:
      if c.COMPREL% = nil then signal error
      else c.COMPREL%.PW:= pw; c.COMPREL%.PV:=pv
      end if
    else : signal error
    end case
  end proc;
  procedure change_password(c:COMPADT;pn:integer);
  imports PERSADT.proc: invoked ;
    PERSDB, COMPuserid: readonly ;
  var n,nme,op,np,adr:string;
    afd:dept;
  begin {geen user-controle}
```

```
    write 'please type in name, old password and new password';
    read n,op,np;
    PERSADT.proc(inspect,COMPuserid->PERSDB,pn,nme,afd,adr);
    if n ~= nme then signal access_violation
    else c.COMPREL%:=c.COMPREL[pn];
      if c.COMPREL% = nil then signal access_violation
      else if c.COMPREL%.PW ~= op then signal access_violation
        else c.COMPREL%.PW:= np;
             c.COMPREL%.PV:= c.COMPREL%.PV + 1
        end if
      end if
    end if
  end change_password;
end module ;

type PERSADT = module
exports
  procedure proc(op:ops;u:user->p:PERSADT;pn:integer;
    n:string;afd:dept;adr:string);
  exception access_violation, error
external
  type dept, ops, user,STADT,DOCADT,TASADT;
  var PERSuserid:user;STDB:STADT;DOCDB:DOCADT;TASDB:TASADT
end external
rep PERSREL : relation [ key P_nr] of
        P_nr:integer;  {Persoonsnummer}
        N:string;      {Naam}
        AFD:string;    {Afdeling}
        ADR:string     {Adres}
      end relation ;
      last_pers_nr: integer
    end record
ops
  procedure proc(op:ops;u:user->p:PERSADT;pn:integer;
    n:string;afd:dept;adr:string);
  imports last_pers_nr: modified ;
    PERSuserid: readonly ;
    user.identification,STADT.proc,DOCADT.proc,TASADT.proc: invoked ;
```

```
var i:integer;ui:dept;
    s,t:string;
  handler increase_i;
  imports i: modified ;
  begin clear ; i:=i+1; {continue met volgend statement} end ;
begin ui:= user.identification(u);
  if ~(ui in [PERS,STop,COMPop,DOCop,TASop] ¦ (op=inspekt & ui=COMP))
      then signal access_violation
  end if ;
  p.PERSREL%:= p.PERSREL[pn]
  case op of
  when inspect:
    if p.PERSREL% = nil then signal errror
    else n:=p.PERSREL%.N; afd:=p.PERSREL%.AFD;
      adr:= p.PERSREL%.ADR
    end if
  when add:
    if p.PERSREL% = nil then
      p.last_pers_nr:=p.last_pers_nr + 1;
      pn:= p.last_pers_nr;
      p.PERSREL:+ [<p.last_pers_nr,n,afd,adr>]
    else {de toevoeging betreft een reeds bestaande persoon;
          bijvoorbeeld een TAS medewerker die al als student
          was ingeschreven}
      if p.PERSREL%.AFD = STUD then
         p.PERSREL%.AFD:= afd
      end if ;
      pn:= p.PERSREL%.P_nr
    end if
  when delete:
    if p.PERSREL% = nil then signal error
    else i:= 0;
    {we gebruiken een trucje met excepties om uit te zoeken
     in hoeveel van de andere databanken deze persoon zit;
     op deze manier zijn we er zeker van dat alle studenten,
     docenten en TAS medewerkers slechts één keer in
     PERSREL voorkomen}
    if ui ~= STop then STADT.proc(inspect,PERSuserid->
```

```
        STDB,n,s,t) ![error:increase_i];
      if ui ~= DOCop then DOCADT.proc(inspect,PERSuserid->
        DOCDB,n,s,t) ![error:increase_i];
      if ui ~= TASop then TASADT.proc(inspect,PERSuserid->
        TASDB,n,s,t) ![error:increase_i];
      if i >= 2 then p.PERSREL:- p.PERSREL%
    when change:
      if p.PERSREL% = nil then signal error
      else p.PERSREL%.AFD:= afd;
        {de naam is onveranderbaar}
        p.PERSREL%.ADR:= adr
      end if
    else : signal error
    end case
  end proc;
begin last_pers_nr:= 0 end module ;
```

De ADT user kan vrijwel letterlijk van paragraaf 4.8. overgenomen worden, waarbij overigens een voorziening getroffen moet worden voor het creëren van de speciale userid's, zoals STuserid. Dit is in principe een eenvoudige zaak: ofwel deze userid's worden geacht altijd bestaan te hebben zoals de userid uu van de vorige paragraaf ofwel het (operating) systeem (ALPHA) creëert ze door het aanroepen van user met een extra parameter.

De bovenstaande definities kunnen nu in toepassingsprogramma's gebruikt worden. In deze toepassingsprogramma's zijn controles op semantische integriteitseisen en protectieregels niet meer nodig. Wel moeten ze adequaat reageren op excepties.

Aan het slot van dit hoofdstuk gekomen, menen we te kunnen concluderen dat het inderdaad mogelijk is vanuit één gezichtspunt en met één taal (PLAIN) databankmanipulaties, semantische integriteitseisen en protectie- en privacyregels te beschrijven.

VERANTWOORDING

Deze voordracht kwam tot stand gedurende de zomer van 1980 toen schrijver dezes te gast was bij de Medical Information Science groep van de universiteit van Californië te San Francisco, daartoe mede in staat gesteld door een Fulbright-Hayes fellowship.

Ook Z.W.O. medewerker drs. M.L. Kersten was deze zomer te gast bij deze groep, zodat er intensief met professor Wasserman, onze gastheer, kon worden samengewerkt hetgeen o.a. resulteerde in dit verhaal.

LITERATUUR

[1] ABRIAL, J.R., Data Semantics, in Data base management, J.W. Klimbie and K.I. Koffeman, North Holland, 1974.

[2] ANGELUCCI, W.A.W., The Optimal Placement of Dynamic Recovery Checkpoints in Recoverable Computer Systems, Techn. Report DSL TR-126, Stanford University, December 1976.

[3] APERS, P.M.G., Distributed Query Processing: Minimum Response Time Schedules for Relations, Informatica Report, IR 50, Vrije Universiteit, Amsterdam, March 1979.

[4] APERS, P.M.G., Data Allocation and Distributed Query Processing, Proceedings ACM Pacific "80, San Francisco, November 12-14, 1980.

[5] ARMSTRONG, W.W., Dependency Structures and Database Relationships, Proc. IFIP 74, North Holland, pp.580-1083.

[6] ASTRAHAN, M.M. et al, System R: Relational Approach to Database Management, ACM Transactions on Database Systems Vol. 1, No 2, June 1976, pp. 97-137.

[7] BACHMAN, C.W., The Programmer as Navigator, Communications of the ACM Vol. 16, No 11, November 1973.

[8] BACHMAN, C.W. & M. DAYA, The Role Concept in Data Models, Proc. 3rd Intn'l. Conf. on VLDB, October 1977, pp.464-476.

[9] BERNSTEIN, P.A., J.B. ROTHNIE & D.W. SHIPMAN, Tutorial: Distributed Data Base Management, IEEE Computer Society 1978.

[10] BERNSTEIN, P.A., Synthesizing Third Normal Form Relations from Functional Dependencies, ACM Transactions on Database Systems, December 1976, pp.277-293.

[11] BLASGEN, M.W. & K.P. ESWARAN, Storage and Access in Relational Data Bases, IBM Systems Journal Vol. 6, No 4, 1977.

[12] BRODIE, M.R., Specification and verification of database semantic integrity, Technical Report Computer Systems Research Group, University of Toronto, Ph.D. Dissertation, 1978.

[13] BRODIE, M.R., personal communication.

[14] BUBENKO, J.A., Data Models and their Semantics, Data Design, Infotech State of the Art Report 8.4, 1980.

[15] BUBENKO, J.A., IAM: Inferential Abstract Modeling - An Approach to Design of Information Models for Large Shared Data Bases, IBM Technical Report, RC6343 1977.

[16] CHEN, P.P.S., The entity-relationship model: toward a unified view of data, ACM Transactions on Database Systems, March 1976.

[17] CODD, E.F., A relational model for large shared data banks, Communications of the ACM Vol. 13, No 6, June 1970, pp.377-387.

[18] CODD, E.F., R.S. ARNOLD, J-M. CADIOU, C.L. CHANG & N. ROUSSOPOULOS, RENDEZVOUS version 1: an experimental English-language query formulation system for casual users of relational data bases, Research report RJ2144, IBM Research Laboratory, San Jose, California, January 1978.

[19] CODD, E.F., Extending the database relational model to capture more meaning, ACM Transactions on Database Systems Vol. 4, No 4, December 1979.

[20] DATE, C.L., An Introduction to Data Base Systems, 2nd edition, Addison-Wesley 1977.

[21] ESWARAN, K.P. et al., On the Notion of Consistency and Predicate Locks in Data Base Systems, Communications of the ACM, November 1976.

[22] FAGIN, R., A normal form for relational databases that is based on domains and keys, Research Report RJ2520, IBM Research Laboratory, San Jose, California, May 1979.

[23] FAGIN, R., Horn clauses and database dependencies, Research Report RJ2741, IBM Research Laboratory, San Jose, California, March 1980.

[24] FAGIN, R., Multivalued dependencies and a new normal form for relational database, ACM Transactions on Database Systems Vol. 2, No 3, September 1977, pp.262-278.

[25] FRY, J.P. et al., Stored Data Description and Translation: A Model and

Language, Information Systems Vol. 2, No 3, 1976.

[26] GERRITSEN, R. & H.L. MORGAN, Dynamic Restructuring of Data Bases with Generation Data Structures, Proc. ACM Annual Conf., 1976.

[27] GERRITSEN, R., The Relational and Network Models of Data Bases: Bridging the Gap, Proc. 2nd USA-Japan Computer Conf., August 1975.

[28] GRAY, J.N., Notes on Data Base Operating Systems, in Operating Systems an Advanced Course Bayer, Grahm and Seegmueller (eds.), Lecture Notes in Computer Science Vol. 60, Springer-Verlag, 1978.

[29] GRIFFITHS-SELINGER, P., M.M. ASTRAHAN, D.D. HAMBERLIN, R.A. LORIE & T.G. PRICE, Access Path Selection in a Relational Database Management System, Research Report RJ2429, IBM Research Laboratory, San Jose, California, August 1979.

[30] HOARE, C.A.R., Monitors: An Operating System Structuring Concept, Communicatons of the ACM Vol 17, No 10, October 1974.

[31] HSIAO, D.K., D.S. KERR & S.E. MADNICK, Computer Security, ACM Monograph Series, Academic Press, New York, 1979.

[32] HSIAO, D.K. & S.E. MADNICK, Data Base Machine Architecture in the Context of Information Technology Evolution, Proc. 3rd Intn'l. Conf. on VLDB, October 1977.

[33] JONES, A.K. & W.A. WULF, Towards the Design of Secure Systems, Software-Practice and Experience, Vol. 5, 1975, pp.321-336.

[34] KERSCHBERG, L. et al., A Taxonomy of Data Models, Proc. 2nd Intn'l. Conf. on Software Engin., October 1976.

[35] KLEEFSTRA, W., The concept of data model in database systems, Ph.D. Dissertation, Technical University Twente, 1980.

[36] KOBAYASHI, I., DBTG Model, Relational Model and Information Space Model of the Information Structure, Proc. 2nd USA-Japan Computer Conf., August 1975.

[37] LAMPSON, B.W., J.J. HORNING, R.L. LONDON, J.G. MITCHELL & G.J. POPEK, Report on the programming Language Euclid, Xerox Report,Palo Alto, August 1976.

[38] LANGEFORS, B. & B. SUNDGREN, Information Systems Architecture, Petrocelli Charter 1975.

[39] LE BIHAN, J., C. ESCULIER, G. LE LANN & L. TREILLE, SIRIUS-DELTA: un Prototype de Systeme de Gestion de Bases de Donnees Reparties, Proceedings of the Intn'l. Symp. on Distributed Databases, C. Delobel and W. Litwin (eds.), North Holland, 1980.

[40] LIN, C.S. et al., The Design of a Rotating Associative Memory for Relational Data Base Applications, ACM Transactions on Database Systems, March 1976.

[41] LINDENCRONA-OHLIN, E., A Study on Conceptual Data Modeling, Ph.D. Dissertation, Chalmers University of Technology and University of Gothenburg, 1979.

[42] LISKOV, B. et al., Abstraction Mechanisms in CLU, Communications of the ACM Vol. 20, No 8, 1977, pp.564-1076.

[43] MEYER, B. & H.J. SCHNEIDER, Predicate Logic and Database Technology, Advanced Course on Database Languages and Natural Language Processing, 1975.

[44] MOHAN, C., An overview of recent database research, db, quarterly newsletter of SIGBDP of the ACM Vol. 10, No 2, 1978.

[45] MORRIS, R. & K. THOMPSON, Password Security: A Case History, Communications of the ACM Vol. 22, No 11, November 1979.

[46] OZKARAHAN, E.A., S.A. SCHUSTER & K.C. SEVCIK, Performance Evaluation of a Relational Associative Processor, ACM Transactions on Database Systems Vol. 2, No 2, 1977, pp.175-195.

[47] PALERMO, E.P., A Database Search Problem, Proceedings 4th Intn'l. Symp. on Computers and Information Science, Miami Beach, Fla, December 1972.

[48] PIROTTE, A., The Entity-Association Model: An Information Oriented Data Base Model, Proc. ACM Intn'l. Computing Symp., April 1977.

[49] ROTHNIE, J.B., P.A. BERNSTEIN, S.FOX, N. GOODMAN, M. HAMMER, T.A.LANDERS, C. REEVE, D.W. SHIPMAN & E. WONG, Introduction to a System for Distributed Databases (SDD-1), ACM Transactions on

Database Systems Vol. 5, No 1, March 1980.

[50] ROUSSOPOULOS, N. & J. MYLOPOULOS, Using Semantic Networks for Data Base Management, Proc. 1st Intn'l. Conf. on VLDB, September 1975.

[51] SCHMIDT, J.W., Some High Level Language Constructs for Data of Type Relation, ACM Transactions on Database Systems Vol. 2, No 3, September 1977, pp.247-261.

[52] SENKO, M.E., Conceptual Schemas, Abstract Data Structures, Enterprise Descriptions, Proc. Intn'l. Computing Symp., 1977.

[53] SENKO, M.E., Data Structures and Data Accessing in Data Base Systems - Past, Present, Future, IBM Systems Journal Vol. 16, No 3, 1977.

[54] SICHERMAN, G.L., W. DE JONGE & R.P. VAN DE RIET, Answering Queries Without Revealing Secrets, draft report, vakgroep informatica, Vrije Universiteit, 1980.

[55] SHU, N.C., B.C. HOUSEL, R.W. TAYLOR, S.P. GHOSH & V.Y. LUM, EXPRESS: A Data Extraction, Processing and Restructuring System, ACM Transactions on Database Systems Vol. 2, No 2, 1977, pp.134-174.

[56] SMITH, D.C.P. & J.M. SMITH, Conceptual Database Design, Data Design, Infotech State of the Art Report 8.4, 1980.

[57] SMITH, J.M. & D.C.P. SMITH, Database Abstractions: Aggregation and Generalization, ACM Transactions on Database Systems Vol. 2, No 2, June 1977, pp.105-133.

[58] STONEBRAKER, M., E. WONG, P. KREPS & G. HELD, The Design and Implementation of INGRES, ACM Transactions on Database Systems Vol. 1, No 3, September 1976, pp.189-222.

[59] SU, S.Y.W. & G.J. LIPOWSKI, CASSM: A Cellular System for Very Large Data Bases, Proc. 1st Intn'l. Conf. on VLDB, September 1975.

[60] SUNDGREN, B. Database design in theory and practice - towards an integrated methodology, Issues in Data Base Management H. Weber and A.I. Wasserman (eds.), North Holland, 1978.

[61] THOMAS, R.A.C, Process Structure Alternatives towards a Distributed INGRES, Proceedings Intn'l. Symp. on Distributed Data Bases, Paris, March 12-14, 1980, pp.215-227.

[62] TODD, S.J.P, Hardware Design for High Level Data Bases, Technical Report TN49, IBM Peterlee Scientific Centre, England, 1977.

[63] TSICHRITZIS, D. & A. KLUG (eds.), The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Database Management Systems, Information Systems Vol. 3, 1978; also in A.I. Wasserman and P. Freeman (eds.): Tutorial on Software Design Techniques, 3rd edition, IEEE Computer Society, 1980.

[64] ULLMAN, J.D, Principles of Database Systems, Computer Science Press Potomac Maryland, 1980.

[65] VAN DE RIET, R.P., M.L. KERSTEN & A.I. WASSERMAN, A Module Definition Facility for Access Control in Distributed Data Base Systems, Proceedings of the 1980 Security and Privacy conference, Oakland, 1980.

[66] VAN DE RIET, R.P., Normalisation - A Tool in Designing a Database, Data Design, Infotech State of the Art Report 8.4, 1980.

[67] VAN DE RIET, R.P, On multivalued dependencies and independencies, IBM Research Report RJ2380, IBM Research Laboratory, San Jose, California 1978.

[68] WASSERMAN, A.I., A software engineering view of data base management, in Issues in Data Base Management H. Weber and A.I.WASSERMAN (eds.), North Holland, 1979.

[69] WASSERMAN, A.I., D.D. SHERERTZ, M.L. KERSTEN, R.P. VAN DE RIET & M.D. DIPPÉ, Revised Report on the Programming Language PLAIN, Informatica Rapport vakgroep informatica, Wiskundig Seminarium, Vrije Universiteit, Amsterdam, 1980.

[70] WIEDERHOLD, G., Database Design, McGraw Hill, 1977.

[71] WONG, E. & K. YOUSSEFI, Decomposition: A strategy for Query Processing, ACM Transactions on Database Systems September 1976.

[72] WULF, W., E. COHEN, W. CORWIN, A. JONES, R. LEVIN, C. PIERSON & F. POLLACK, HYDRA:The Kernel of a Multiprocessor Operating System, Communications of the ACM Vol. 17, No 6, June 1974, pp.337-345.

[73] YOUNG, J.W. & H.K. KENT, Abstract formulation of data processing

problems, in Evolution of business systems analysis techniques Couger and Knapp (eds.), Wiley, 1974.

LOCKING AND RECOVERY IN A SHARED DATABASE SYSTEM: AN APPLICATION PROGRAMMING TUTORIAL*

C.J. DATE
IBM, San Jose, USA

1. INTRODUCTION

We are concerned in this paper with the concepts of locking and recovery as they apply in the context of a shared database system. We define such a system, informally, as one that supports simultaneous access to one or more data bases from multiple concurrently executing programs. ("Simultaneous access" and "concurrently executing" here are to be understood in the loose sense in which such terms are usually employed in discussions of multiprogramming and the like.) The system operates under the control of a database management system or DBMS. It is well known that a shared database system faces special problems in the areas of recovery and data integrity, and that solutions to such problems are based on a variety of techniques, most of them involving some form of *locking*. Over the past few years, in fact, a considerable body of practical experience has built up in the general area of locking and recovery; in addition, a number of papers have been published treating the subject from a more formal and theoretical standpoint (see, e.g., references [1-6, 15, 16, 18-23]). Tutorial papers have also been published (for example, see [7]). But little has appeared concerning the impact of locking and recovery considerations on the application programmer. This paper is an attempt to rectify this omission.

Of course, a major reason for the omission is that locking and recovery are generally supposed to be "transparent to the user" (user here meaning the application programmer): such aspects are considered to be the responsibility of the system, not the user. In other words, the user should be able to code the application without concern for the fact that the database is shared. We

* Republished, with permission of IEEE, from Proceedings, Fifth International Conference on Very Large Data Bases, Oct. 3-5, 1979, Rio de Janeiro, Brazil.

sympathize with this objective, and agree that it is quite often achievable. But it is still necessary to consider locking and recovery from the application programming standpoint, for the following reason.

In general it is undesirable that the totality of data that is used (or, worse, that might be used) by a given application be locked for the whole time that the application executes, because concurrency would be reduced. Therefore, the system must have rules or protocols to establish what *subset* of the data has to be locked and for what *subset* of the total time it has to remain locked. The existence of these rules, and in particular their subsetting nature, together imply that:

- the semantic definition of the user's programming language must take into account the fact that the contents of the database are subject to concurrent change by some external agent;
- it must be possible to override the rules (via explicit request from the application), because there will be situations in which the default locking dictated by the rules is not what is wanted, for logical reasons or performance reasons or both. This statement applied both to *scope* (amount of data that is locked) and to *duration* (length of time the locks are retained).

Hence we find that there are already languages in existence - DL/I [8] and the DBTG COBOL Data Manipulation Language [9] are examples - that include various locking and recovery features for use by application programmers. The programming manuals for these languages, however, rarely give much by way of explanation as to why the features are provided or how they are expected to be used. The present paper may be seen as an attempt to provide some motivation for such features - also as an attempt to pave the way for understanding of some of the more rigorous published material. It consists of a tutorial on basic locking and recovery concepts; our primary viewpoint will be that of the application programmer, though from time to time we shall find it necessary to take other viewpoints as well. Readers interested in the *implementor's* viewpoint will find excellent discussions in [18] and [21].

Three final introductory remarks. First, the reader is assumed to understand the need for integrity and consistency in a database; a good discussion of these concepts can be found in [17]. Second, our discussion of recovery is limited to *transaction* recovery, or "in-process" recovery as it is sometimes known; we do not address either post-process recovery (necessitated by discovery of an error some time after the transaction has finished) or system recovery (necessitated by a catastrophic failure on the part of, e.g.,

the DBMS or operating system or hardware). Third, we do not treat the special problems of distributed processing at all; locking and recovery in such systems, though considerably more complex than in a centralized system, should ideally not require any additional action on the part of the application programmer. Reference [20] gives a good introduction to the distributed case from a system viewpoint. (It also suggests that locking per se is not necessarily the best technique for *inter-computer* synchronization [because of the line traffic incurred], and describes an alternative approach that is being implemented for the system SDD-1.)

2. THE APPLICATION PROGRAMMING LANGUAGE

For definiteness we assume that application programs are written using the language UDL [10-12]. UDL (Unified Database Language) is a set of proposed extensions to existing high-level languages such as COBOL and PL/I that supports relational, hierarchical, and network structures in a uniform and consistent manner. We choose UDL as a vehicle for our discussions because it embodies most of the concepts we need to consider. But of course the concepts discussed are quite general and apply equally well to other languages, such as those of [8] and [9].

A database as viewed from UDL consists of a collection of records, partitioned into disjoint *basesets* (one baseset per record-type; the baseset for a given record-type is simply the set of all records of that type). In addition there may be one or more *fansets* defined over these basesets. A fanset is the UDL analogue of the parent-child structure in a hierarchy [8] or the "set type" of DBTG [9]. A hackneyed example is shown in Fig. 1 (irrelevant syntactic details omitted).

```
BASESETS (DEPTSET RECTYPE(1 DEPT, 2 DEPT#, 2 BUDGET),
          EMPSET  RECTYPE(1 EMP,  2 EMP#,  2 SALARY))
FANSETS  (DEPTEMP RECORD (EMP) UNDER (DEPT))
```

Fig. 1: A department-employee hierarchy in UDL

The most fundamental executable statement, and the only one we need consider here, is FIND, whose function is to locate a particular record in the database and set a *cursor* to point to it. The FIND statement below (Fig. 2) sets cursor E to point to ("select") a specific EMP record.

```
FIND UNIQUE (EMP WHERE EMP.EMP# = GIVEN.EMP#)
                                        SET (E);
```

Fig. 2: An example of the UDL FIND statement.

A cursor is a program variable that behaves somewhat like a pointer (but a pointer that points into a databse): its value typically is the address of some particular database record. A program may have any number of cursors, and thus may be addressing (i.e., selecting) any number of records simultaneously.

We are now ready to embark on the tutorial per se. Purely for reference, we summarize in Fig. 3 those features of UDL that are provided explicitly for locking and recovery purposes.

Declarative features

```
                    [ REFERENCE           ]
PROCESSINGLEVEL ( < >                     )
                    [ CHANGE [(mode)]     ]

                    [ NONE                ]
SHARELEVEL      ( < REFERENCE             > )
                    [ CHANGE [(isolation)] ]
```

Manipulative features

Executable statements:		Statement options:
COMMIT	ROLLBACK	KEEP (on FIND)
KEEP	RELEASE	NOWAIT (on FIND)
Exceptions:		RETAIN (on COMMIT)
ROLLBACK	DENIAL	HOLD (on FIND, KEEP)

Fig. 3: UDL locking and recovery feature summary

3. TRANSACTION PROCESSING

We begin by introducing two fundamental concepts, viz. "transaction" and "recoverable unit". We use "transaction" to mean what is variously known as a *process*, a (major) *task*, or a *run-unit*: i.e., it represents a single execution of some application program. As such, it possesses a duration. This duration is subdivided, in a way to be explained, into a sequence of one or more time intervals, and that portion of the transaction corresponding to any one of these intervals is called a *recoverable unit*. A transaction thus

consists of a sequence of recoverable units. Transactions cannot be nested.

(Note: In practice many transactions consist of a single recoverable unit. Some systems even enforce this restriction and consider the two terms to be synonymous. In fact, we should point out right at the outset that there is no commonly agreed nomenclature in the field of locking and recovery; the term "transaction", in particular, has different meanings in different systems, and the "recoverable unit" concept frequently seems to have no name at all. Also "application program" must sometimes be understood to include the case of physically distinct (but communicating) modules, possibly even located at distinct sites in a distributed processing environment.)

We give an example, taken from a banking system, to make these ideas a little more concrete. A typical transaction in such a system might be "transfer amount A from account X to account Y" (A, X and Y being parameters). Notice that the user of this system - the bank clerk - would view this as a single operation; typically, to cause the operation to take place, all such a user would have to do would be to enter a command such as

```
TRANSFER $100 FROM 4732166 TO 9940103
```

at the terminal. But the application program that is invoked to perform this transaction, like most others, will have to perform more than one update on the underlying database (two distinct account records have to be changed). Although the database is consistent before and after the transaction, it is not so between the two updates (a "transfer" transaction should have no overall effect on the total dollar balance, but between the two updates this total is temporarily altered).

This transaction consists of a single recoverable unit. For the sake of the example, let us suppose that the user is able to request two or more transfers at once from the terminal, for instance:

```
TRANSFER $100 FROM 4732166 TO 9940103
         $400 FROM 1573861 TO 8005483
```

Then it is likely that the transaction would involve two recoverable units, one for each individual transfer. Intuitively, each recoverable unit is a unit of work that is perceived as *atomic* by the enterprise
the database system is serving. We shall expand on this point later; for the moment, we content ourselves with noting that the recoverable units in the

example are atomic in that each is an all-or-nothing proposition (either the transfer is successful, or nothing happens at all; the database should not be left in an intermediate, inconsistent state).

All examples in this paper show transactions involving a single recoverable unit, unless otherwise stated.

4. LOCKING

In a shared database system, then, there are M users (transactions) dynamically competing for N resources (units of information in the database; typically individual records). We assume that the transactions are all independent of one another - that is, they could be "serialized", or run one at a time in some arbitrary sequence, without logically impairing the overall operation of the total system. In such an environment, some form of coordinating *control mechanism* is obviously needed, and locking (explained below) is such a mechanism - not the only one possible, but the one most commonly used. Ideally this control mechanism should both maximize the degree of concurrency obtainable and have a minimal impact on the logical design of transactions. (As ususal the objectives conflict; in practice a system may provide several different forms of locking, each representing a different tradeoff between the two. For example, locking individual records rather than entire basesets may allow more concurrency but may also mean more work for the programmer.)

The system, then, must include a component, the *lock manager*, that will grant a lock on a specified object on behalf of a specified transaction. The request for the lock may be issued explicitly by the transaction or implicitly on the transaction's behalf by the DBMS. A lock is a device for controlling access to the object with which it is associated (the object that is locked). To be more precise, a lock is a construct with certain defined properties and behaviour that make it suitable for such access control purposes. For example, if transaction A is granted an "exclusive lock" on record R, then transaction B cannot access R (more accurately, B cannot acquire a lock on R) until A releases its lock.

There is no intrinsic limit on the number of locks that may be granted.

Locking is necessitated by three major problems: (1) lost updates; (2) transaction rollback; and (3) inconsistent analysis. We consider each of these in turn.

5. LOST UPDATES

Figure 4 illustrates the problem, using a simplified form of UDL syntax. "FIND R" is shortened for a FIND that sets some cursor to select some specific record occurrence R; "change R" is shorthand for any of a variety of statements that may update that record R in some way. In the example, A's update is "lost" in the sense that it is never seen; B overwrites it without even looking at it. To make the example more specific, suppose that R contains a field F with initial value 4, and suppose that both A and B copy F into some program variable immediately after the FIND, add one to this variable, and then "change R" by replacing F by the incremented value. After time t4 the value of F will be 5, whereas it should be 6.

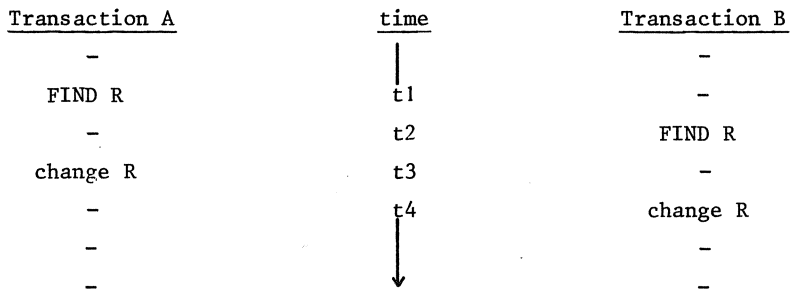


Fig. 4: A's update is lost at time t4

The particular interleaved execution of A and B shown in Fig. 4 is not "serializable": that is, it is not equivalent to running the transactions one at a time, in either possible sequence. A given interleaved execution of some given set of transactions is said to be serialized if it produces the same result as some serial execution of those same transactions [4]. Any serializable execution of a given set of transactions can be considered *correct*, in the sense that it produces the same result as running those transactions one at a time in some arbitrary sequence. (Remember that we are assuming that transactions are all independent of one another, so no particular sequence can be considered any "more correct" than any other.) For this reason serializability is generally accepted as a *formal criterion for correctness* (and a very useful criterion too, in that it greatly aids clear thinking in this potentially confusing area). In other words, if the system is such that the only execution sequences it generates are serializable sequences, then that

system is guaranteed to be correct. In practice, however (as will be discussed later, under "Levels of Isolation" and elsewhere), systems do *not* usually restrict themselves to serializable sequences only, since a given sequence may be nonserializable and yet still be consider correct. Serializability is a sufficient condition for correctness but not a necessary one. (At the risk of laboring the obvious, we stress once more that "serializable" does *not* necessarily mean "serialized" - transactions are still executing concurrently, but locking, or some other synchronization mechanism, is being used to guarantee correctness.)

We illustrate the concept of serializability by introducing a locking protocol that will solve the lost update problem, as follows: (1) no transaction is allowed to change a record without first acquiring an exclusive lock (abbrev. X lock) on that record; (2) if a transaction cannot be granted such a lock at the time it requests it, then that transaction must *wait* until the lock can be so granted. The effect of such a rule on the example is illustrated in Fig. 5 ("FINDX R" is shorthand for "FIND R and acquire an X lock on it"; "unlock R" means release the lock on R - for reasons to be discussed, UDL does *not* actually provide a "unlock" statement as such, but of course some form of unlocking function is provided in the language).

Transaction A	time	Transaction B
-	↓	-
FIND R	t1	-
-	t2	FINDX R (wait)
change R	t3	wait
-	↓	wait
unlock R	t4	resume (FINDX R)
-	t5	change R
-	t6	unlock R
-	↓	-
-	↓	-

Fig. 5: B is forced to wait for A's update

It can be seen that the effect is to delay execution of B's "FINDX" until after A's "unlock". Thus B is forced to see A's update before proceeding with its own. The interleaved execution of A and B is now equivalent to the serial execution A-then-B; the locking protocol has enforced serializability,

and hence correctness. In this sense we have solved the lost update problem. However, the solution itself can lead to another problem, namely the problem of deadlock.

6. DEADLOCK

Deadlock is a situation in which two or more transactions are in a simultaneous wait state, each one waiting for one of the others to release a lock before it can proceed. Consider Fig. 6.

Transaction A	time	Transaction B
-	↓	-
FINDX R1	t1	-
-	t2	FINDX R2
FINDX R2	t3	-
wait	t4	FINDX R1
wait	↓	wait
wait		wait

Fig. 6: Deadlock occurs at time t4

It is possible to define protocols that will preclude the possibility of deadlock. However, such protocols are generally inapplicable in the database context. For example, one such protocol is: Permit a transaction to request any number of locks simultaneously; grant the request only if all requested locks can be granted; do not grant the request if the transaction already holds any locks. Simplifying matters somewhat, this rule means that transactions must lock all the data they need before they update any of it. It can easily be seen that observing this protocol will solve the problem of Fig. 6. However, it is not always possible to observe the protocol; for example, referring again to Fig. 6, transaction A may be unable to identify record R2, and hence unable to lock it, until it has locked and examined R1 (the identity of R2 may depend on values stored in R1). What distinguishes a database system from other environments (as far as locking is concerned) is the following: (1) lockable resources are not usually identified by name but by value, which may mean that it cannot be determined until runtime that two distinct requests are for the same object; (2) the precise locking scope for a given transaction is usually determined dynamically. It follows

that deadlock cannot be totally avoided, in general. Hence the system must be prepared to detect deadlocks and to resolve them when they oocur.

Aside: It is of course untrue to say that deadlock is totally unavoidable. One way to avoid it is simply not to allow two transactions to run concurrently if their locking scopes can possibly conflict. We have already suggested in the introduction that this approach is generally unsatisfactory: the DBMS must necessarily be pessimistic in its assumptions - e.g., it must assume that two transactions conflict if they both update the same record *type*, regardless of whether they will actually attempt to update the same record *occurrence* - with the result that concurrency will tend to be unacceptably low. A second way to prevent deadlock is simply to reject any lock request that, if granted, would [immediately] cause it, returning a "request denied" indication to the transaction. This solution is also unattractive, however, since there is probably nothing constructive the transaction can do in such a situation. UDL does not explicitly support this approach. A third avoidance technique, due to Rosenkrantz et al. [19], has the advantage of requiring no special action on the part of the user. We defer discussion of this technique to the end of the next section, since it relies on the concept of rollback (introduced in that section) for its implementation. *End of aside.*

7. TRANSACTION ROLLBACK

Detecting a deadlock is basically a matter of detecting a cycle in the graph of "who is waiting for whom" [1]. Checking for deadlock can be done whenever a lock request causes a wait, or on some periodic basis. (Checking on every wait allows deadlocks to be detected as soon as they occur, but may impose too much overhead [especially as most checks will result in "no deadlock found"]; checking less frequently reduces the overhead but may mean that some deadlocks are detected late.) Alternatively the system may use a timeout mechanism, and simply assume that a transaction is deadlocked if it has done no work in a given time period.

To break a deadlock, the system must choose one of the deadlocked transactions - say the one most recently started, or the one holding the fewest locks - and "roll it back": that is, cancel the recoverable unit concurrently in progress, thus releasing its locks so that its resources can be allocated to some other transaction. (Note that the victim may have locked other

system resources in addition to objects in the database; deadlock is a system-wide problem, not just a database problem. The lock manager must serve the entire system, not just the DBMS.)

Incidently, deadlock is not the only reason for rolling back a transaction. Other reasons include: abnormal termination of the application program; system crash; integrity violation (failure of some consistency check at recoverable unit completion). Space precludes detailed discussion of these additional causes here.

In some systems rollback causes abnormal program termination. The system may then automatically schedule another attempt at execution, by reloading the program. (Such automatic retry is particularly desirable if the transaction is serving an end-user at a terminal, since that user should ideally not even be aware that rollback has occurred.) In other systems rollback may simply cause control to be returned to the program with an appropriate error indication. We shall discuss this case later.

The process of rollback involves *undoing all database changes made by the current recoverable unit*. The effect is as if that recoverable unit had never started. The DBMS must therefore keep a dynamic *audit trail* or *log* of all database changes, and be prepared to use this log to undo changes as just indicated. (It is important to understand that "change" here, and throughout this paper, refers to *any* kind of update, including insertions and deletions.)

Now consider the examples in Figs 7 and 8 below.

Transaction A	time	Transaction B
-	↓	-
change R	t1	-
unlock R	t2	-
-	t3	FIND R
** ROLLBACK **	t4	-
-	↓	-
-		-

Fig. 7: B is dependent on A's uncommitted change (at time t3)

Transaction A	time	Transaction B
-	↓	-
change R	t1	-
unlock R	t2	-
-	t3	FINDX R
-	t4	change R
** ROLLBACK **	t5	-
-	↓	-
-		-

Fig. 8: B's update is lost at time t5

In Fig. 7, B has seen data (at time t3) that "never existed", since when the rollback occurs at time t4 record R will be reset to the value it had prior to the execution of A (strictly, to the value it had prior to the execution of A's current recoverable unit, but for the simplicity we are assuming that each transaction consists of a single recoverable unit.) We say that B has become "dependent on an uncommitted change" - an uncommitted change being one that is subject to possible rollback. (Notice that the sequence of events in Fig. 7 is not serializable.) In Fig. 8 the situation is even worse: when the rollback occurs, B's update to R is lost, because it is overwritten by R's original (pre-A) value. (Again we do not have serializability.)

While the situation in Fig. 7 may be tolerable in some circumstances, that in Fig. 8 is definitely not. Hence, in order to avoid the possiblity of losing updates through rollback, no transaction must *EVER* be allowed to update a change that may subsequently be backed out. This leads us to the following *cardinal principle*:

UNCOMMITTED CHANGES MUST REMAIN LOCKED UNTIL
RECOVERABLE-UNIT TERMINATION

(Note: The only type of lock we have discussed so far is the exclusive or X lock, and the principle should be understood to refer to such locks.) This principle, in conjunction with the rule introduced earlier that a transaction must acquire a lock on the record when it updates it, is sufficient to prevent loss of updates because of rollback. What about the situation in

Fig. 7? As already suggested, there are cases where it may be acceptable to permit a transaction to see (but not update) an uncommitted change. (The "see" operation is basically a special form of FIND that does not request a lock on the found record.) An example of such a situation might be a transaction that is performing some sort of statistical analysis, is not interested in 100% accuracy, and would prefer not to be held up waiting for the data to be committed and locks to be released. But in general such a transaction should operate with considerable circumspection.

We can now see why, as already indicated, UDL does not provide an "unlock" operator for releasing locks on individual updates. Instead, it provides a COMMIT operator, which is used to signal end-of-recoverable-unit (a "commit point"). Normal program termination causes an implicit COMMIT. Issuing COMMIT is an undertaking on the part of the transaction that it has successfully completed some logical unit of work and has not left the database in an inconsistent state. COMMIT releases all locks held by the transaction (with certain exceptions, to be explained), and "commits" all changes made by the transaction since the previous COMMIT (if any). These changes are now visible to other transactions. A committed change can never be rolled back.

Aside: In principle it would be possible to release locks (and to commit changes) one by one instead of all at once. If a rollback occurred then only uncommitted changes would need to be undone. However, such a protocol would imply that special and possibly rather complex application logic would be needed for trying again after a rollback, since in general some updates would have been done and some not. Moreover, the recoverable unit would no longer be an all-or-nothing proposition. In practice, therefore, it is reasonable to commit all changes simultaneously, and the principle as stated above is generally accurate. (But see the discussion of IMMEDIATE mode at the end of the paper.) *End of aside.*

Consider now a transaction that is sending messages to a user at a terminal. Just as database changes made by the transaction must remain locked until COMMIT, so messages sent to the user should generally be kept pending and not transmitted by the system until COMMIT, for precisely analogous reasons. If rollback occurs these messages should not be sent at all. A striking example of the need for such a rule is provided by the case of a message that triggers some irrevocable external action, such as a payment from a cash-issuing terminal or the firing of a missile ... . (However, there are also situations in which messages must be transmitted immediately. Detailed

discussion of message handling is beyond the scope of this paper.)

As already mentioned, rollback in some systems causes abnormal program termination, and the application is removed from the system (and possibly then reloaded). In other systems the effect is limited to the removal of changes and the release of locks (and perhaps the purging of messages): the application program is not terminated but remains in operation. An indication is returned to it that rollback has occurred, so that it can request locks again and perform any other special processing that is necessary. In UDL such a situation causes an interrupt (e.g., it raises an ON-condition in the PL/I version), and control is passed to a special ROLLBACK procedure. Returning control to the application imposes an additional burden on the programmer but is desirable in some cases. A good example is a longrunning batch program, which should generally be subdivided into a sequence of several recoverable units, to reduce the amount that has to be redone in the event of rollback.

There are also situations in which the transaction itself can determine that rollback is necessary. For example, it may receive an error return from a database operation (e.g., after an insert operation that violates a uniqueness constraint), and decide that there is no point in continuing. In UDL the ROLLBACK statement is provided to allow a transaction to request its own rollback in such a situation. Abnormal program transaction causes an implicit ROLLBACK, as already mentioned.

We conclude this section with a brief discussion of the deadlock avoidance technique proposed in [19]. The technique comes in two versions, "Wait-Die" and "Wound-Wait". The basic idea in both is that transactions have a unique start-time, and thus a unique "age" at any given instant; and protocols ensure that the system cannot possibly include both an older transaction waiting for a younger and a younger waiting for an older. In other words, *either* all waits are of older transactions waiting for younger ones (Wait-Die), *or* all waits are of younger transactions waiting for older ones (Wound-Wait). Locking conflicts such as those at times t3 and t4 in Fig. 6 are resolved as follows: (a) in Wait-Die the requestor waits if it is older, otherwise it is rolled back and retried; (b) in Wound-Wait the requestor waits if it is younger, otherwise the *other* transaction is rolled back and retried. Whichever version is in effect, it is clear that it is impossible to find a sequence of transactions T1,T2,T3, ..., Tn such that T1 is waiting for T2, T2 is waiting for T3,..., and Tn is waiting for T1, because of the time ordering of transactions and the protocol; thus deadlock cannot occur.

8. INCONSISTENT ANALYSIS

A transaction may need to keep data locked even if it is not updating. Consider Fig. 9, which shows two transactions operating on account (ACC) records: transaction A is summing account balances, transaction B is transferring an amount 10 from account 3 to account 1.

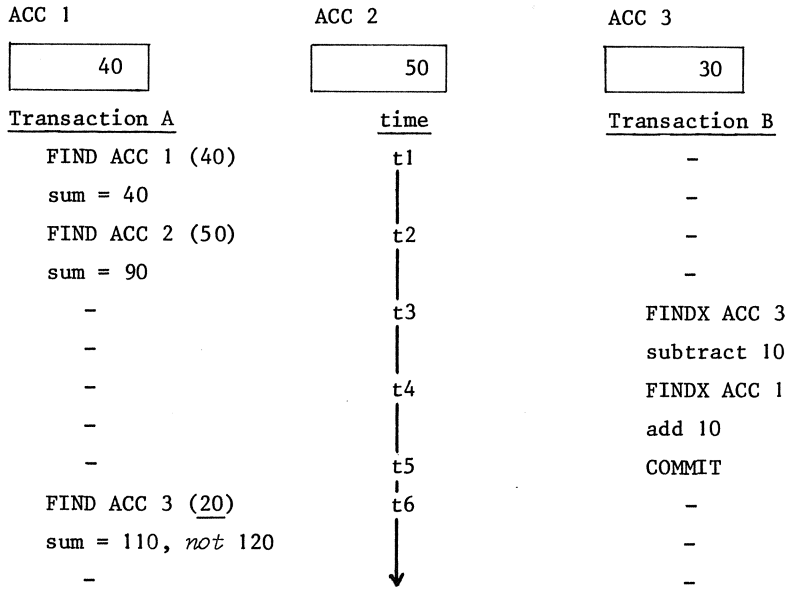


Fig. 9: A sees inconsistent data

The result produced by A (110) is obviously incorrect. If A goes on to write this result back into the database, it will actually leave the database in an inconsistent state. It is clear that A should lock each account record as it reaches it, and should retain the locks at least until the summation is complete; this will prevent B from updating an account record after A has seen it. In the example, in fact, it will lead to deadlock! A moment's reflexion will show that this effect, undesirable as it may appear at first sight, is exactly what is wanted.

The locks set by A need not be exclusive (X locks), however. X locks would serve the purpose, of course, but they would also reduce concurrency unnecessarily. All that is requires to prevent concurrent transactions from *changing* the data - there is no harm in allowing them just to see it.

Accordingly we introduce a second type of lock, the shared or S lock, with the property that if transaction A holds an S lock on record R (say), then transaction B can also be granted an S lock on R, but *cannot* be granted an X lock on R. (Remember that a transaction is required to obtain an X lock on a record when it updates it.)

UDL provides two special operators to help with some problems like that of Fig. 9: KEEP, which acquires and retains an S lock on a specified record, and RELEASE, which releases such locks. KEEP may appear as a statement in its own right or as an option on FIND. Thus transaction A of Fig. 9 should use FIND...KEEP for each account record, and should RELEASE the locks on these records after the summation is complete.

Aside: A transaction A that releases any lock and then goes on to acquire another lock always runs the risk of seeing inconsistent data. That is, it is always theoretically possible to define another transaction B that could execute concurrently with A in such a way that the interleaved execution of the two transactions is not serializable (i.e., not equivalent to the sequence A-then-B, nor to the sequence B-then-A). If all transactions obey the following rules:

(a) before operating on an object the transaction first acquires a lock on that object; and

(b) after releasing a lock on the transaction never acquires any more locks;

then all interleaved executions of those transactions are serializable [4]. This statement is a little misleading, however: the rules are accurate as stated, but it must be clearly understood that sometimes it is the *nonexistence* of some data that is the "object" that must be locked! See the discussion of phantoms, later. For a more complete discussion of this phenomenon the reader is referred to [4].

The rules as states are also pessimistic: that is, they assume the worst case. In practice the application designer for transaction A may *know* that no interfering transaction B will actually exist in the system, and may therefore choose to release locks early in order to gain more concurrency. But it should be clearly understood that such a decision represents a performance tradeoff (increased concurrency vs. the possibility of seeing inconsistent data). *End of aside.*

We have said that a transaction that wishes to update a record must obtain a lock on that record before being allowed to do the update. Some systems will allow this to be an S lock rather than an X lock, and will "promote"

the lock to X level if the update actually occurs. Deadlock can occur in such a system as illustrated in Fig. 10.

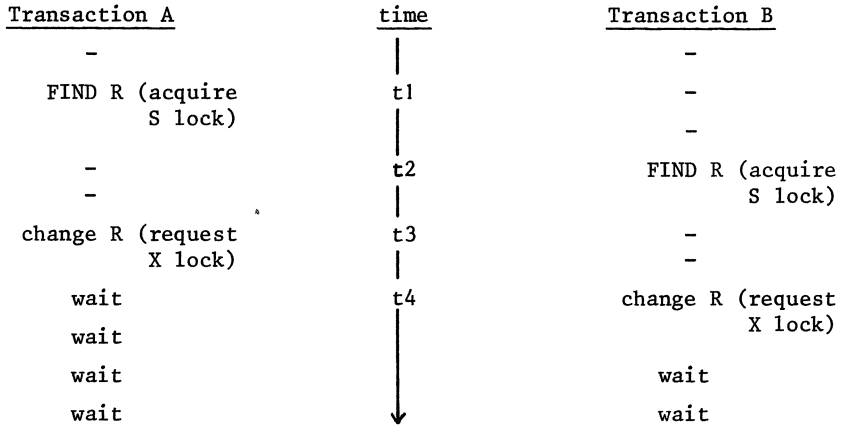


Fig. 10: Deadlock occurs at time t4

9. LOCK GRANULARITY AND LOCK TYPES

So far we have generally assumed that the unit of locking is the individual record. However, locks can also be applied to larger or smaller units of data: for example, to an entire database, or to a baseset (set of all records of a given type), or, going to the opposite extreme, to a field within an individual record. We speak of *locking granularity* [5,6]. (As usual there is a tradeoff: the finer the granularity, the greater the concurrency; the coarser, the simpler the transaction logic.) Indeed, the *logical* concept of locking individual records is frequently implemented by *physically* locking pages or blocks (containing multiple records), for reasons that need not concern us here. This fact does not affect the model of locking we are presenting. But it does illustrate a general principle, which we shall call Implementation Principle #1: *The system can safely lock more than the minimum requested.* "Safely" here means that integrity is not compromised, though of course concurrency may suffer and there is an increased chance of deadlock. (Also there is some danger that programs may come to rely on the fact that the implementation is doing more than is strictly necessary, and this could lead to problems in moving programs between systems.)

In passing we note another fairly obvious general principle, viz. Implementation Principle #2: *The system can safely retain locks longer than required.*

We also introduce the concept of lock *types*. Two types, X and S, have already been discussed, but other types are possible. Reference [5] defines five: S, X, IS (intent shared), IX (intent exclusive), and SIX (shared, intent exclusive). A general rule is that a Transaction cannot acquire an S or X lock on an object without first acquiring an appropriate intent lock (IS, IX or SIX) - or stronger lock, see Fig. 11 - at the next coarser granularity. For example, to acquire an X lock on a record, the transaction must already hold at least an IX or SIX lock on the baseset containing that record. This rule is needed to simplify the detection of locking conflicts; see the discussion of Fig. 11, later. (The rule as stated here is somewhat simplied, in that it assumes that the lockable resources - fields, records, basesets, etc. - form a strict hierarchy. In practice the presense of indexes and other implementation structures may complicate the picture considerably. See [5] for more details.)

The various types of lock may be explained intuitively as follows. For simplicity, we restrict consideration to two granularities only, baseset and record; in such an environment, intent locks (IS, IX, SIX) apply only at the baseset level, S and X locks apply at both levels. The explanations below refer to a transaction T that has requested the indicated type of lock on a baseset B. We do not repeat the explanation of S and X at the record level (already discussed earlier in the paper).

S (*for a baseset*): T can tolerate concurrent readers, but not concurrent updaters, in baseset B. T will not update any records in B.

X (*for a baseset*): T cannot tolerate any concurrent access to B at all; T itself may update individual records in B.

IS: T can tolerate concurrent readers and concurrent updaters in B; T will set S locks on individual records in B when necessary to guarantee stability of those records.

IX: Same as IS, *plus* T will update individual records in B and therefore will set X locks on those records.

SIX: Same as IX, *except* that T cannot tolerate concurrent updaters in B.

To put it another way, an S, X or SIX lock on B means that no concurrent transaction can make any changes in B at all. X and SIX both allow T to make changes in B, S does not. The advantage of an X lock over an SIX lock at the baseset level is that there is no need to set individual record locks, which may be significant if T will update many records in B. (Setting many locks

implies overhead in both space and time, and could lead to abnormal termination if the lock manager runs out of space for its lock lists.) Reorganization and similar utilities, in particular, commonly use X locking at a very coarse granularity - sometimes even at the level of an entire database.

A rigorous definition of lock types is provided by the following *compatibility matrix* (Fig. 11). A "C" in this matrix at a particular row-and-column position indicates that the lock types corresponding to that row and column conflict, in the following sense: If transaction B requests a lock of the type corresponding to the row (on some object), then that request is in conflict if some distinct transaction A already holds a lock of the type corresponding to the column (on that same object). Note that the matrix is symmetric. Conflict usually means that the requesting transaction has to wait. In some systems, however, control is given straight back to the requestor with an indication that the data is locked, on the principle that there may be other useful work that can be done. UDL provides a NOWAIT option on FIND to request immediate return of control if the FIND encounters a lock. In such a system, if a lock is encountered (and NOWAIT has been specified), an interrupt occurs and control is passed to a special DENIAL procedure. Note that FIND...NOWAIT cannot possibly cause a deadlock.

Figure 11 also shows a *precedence graph* for the five lock types (explained below).

	S	X	IS	IX	SIX
S	-	C	-	C	C
X	C	C	C	C	C
IS	-	C	-	-	-
IX	C	C	-	-	C
SIX	C	C	-	C	C

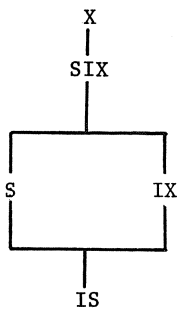


Fig. 11: Lock type compatibility matrix and precedence graph

The precedence graph of Fig. 11 is interpreted as follows: Lock type L1 is *weaker* (lower in the graph) than lock type L2 if and only if, whenever there is a C in L1's column in the matrix at a given position, there is also a C in L 's column at the same position. This definition of relative strength implies that a lock request that fails for a given lock type will certainly

fail for a stronger lock type. This leads to Implementation Principle #3: *The system can safely treat a request for a given lock type as a request for any stronger type*. For example, a system that does not support S locks can interpret all requests for such locks as requests for X locks instead.

10. UDL INTENT SPECIFICATIONS

Locking at the baseset level is specified in UDL by an INTENT declaration for the baseset concerned. (A possible extension would allow INTENT to be specified at runtime. Note that INTENT in UDL does not refer to "intent locking", as in IS, IX, etc., but to the transaction's intended use of a baseset.) An INTENT consists of a PROCESSINGLEVEL and a SHARELEVEL. Intuitively, PROCESSINGLEVEL indicates the type of processing this program intends to perform, and SHARELEVEL indicates the type of processing this program can tolerate on the part of concurrent programs. For PROCESSINGLEVEL the possibilities are REFERENCE (default), meaning reference-type operations only, and CHANGE, meaning update-type operations (CHANGE subsumes REFERENCE). For SHARELEVEL the possibilities are CHANGE (default), REFERENCE, and NONE. The implementation of these specifications in terms of the lock types previously discussed is indicated in Fig. 12. (The figure shows the locks that must be set at the baseset level. For IS, IX, and SIX, of course, record-level locks must also be set as needed.)

SHARE / PROC	NONE	REFERENCE	CHANGE
REFERENCE	X	S	IS
CHANGE	X	SIX	IX

Fig. 12: SHARELEVEL (SHARE) and PROCESSINGLEVEL (PROC)
- implementation in terms of baseset locks

Note 1: It would be possible to extend both PROCESSINGLEVEL and SHARELEVEL to allow explicit specification of, e.g., "delete" intent, "update" intent against particular fields, and so on. But such extensions do not seem to offer any possibilities of increased concurrency (though they could be useful for authorization purposes).

Note 2: The DBMS can implement a SHARELEVEL of NONE as one of REFERENCE if PROCESSINGLEVEL is also specified as REFERENCE. Such an implementation will permit an increase in concurrency without any accompanying loss of integrity.

Note 3: A database that includes fansets has two fundamentally distinct ways of representing information, via records and via parent-child links [13]. The operators CONNECT, DISCONNECT, and RECONNECT provide for the creation, destruction, and modification of such links. For completeness it might be desirable to consider fansets and links, as well as basesets and records, as lockable resources. However, a conscious decision not to take this approach was made in the design of UDL, to reduce complexity. Instead, CONNECT, DISCONNECT, and RECONNECT are considered as update operations against the corresponding parent and child records, and a program using them must acquire the necessary locks on those records (or their containing basesets).

Note 4: Baseset locks are normally released only when the program completes or when rollback occurs, not at COMMIT. A system that does release baseset locks at COMMIT (or on rollback, if control is returned to the transaction) must provide a means - possibly implicit - for the transaction to reacquire those locks before continuing. UDL currently assumes that such reacquisition, if required, is totally implicit. Incidentally, the transaction may have to wait a very long time to reacquire a baseset lock if some other transaction has seized that baseset in the interim, which is a good argument against releasing such locks.

11. LEVELS OF ISOLATION

Suppose that SHARELEVEL (CHANGE) has been specified for a baseset B. To what extent does the transaction have to be aware of the possibility of concurrent change in B? The answer for this question depends on the *level of isolation* specified for B (for the transaction). The concept of isolation level (under the name "degree of consistency") was introduced in [6], though there it was applied to an entire transaction rather than to the transaction's use of an individual baseset. (System R [14] supports the concept at the transaction level.) In UDL isolation level is specified as a suboption of SHARELEVEL (CHANGE) [since it is irrelevant for other SHARELEVELS]:

```
SHARELEVEL (CHANGE[(isolation)])
```

"Isolation" defines the degree of interference this transaction can tolerate with respect to this baseset; the higher the level, the lower the interference (and the lower the concurrency). Many levels can be defined; some possibilities, expressed rather anthropomorphically, are as follows:

Level 1: Don't let me update anyone else's uncommitted changes.

Level 2: Don't let me see anyone else's uncommitted changes.

Level 3: Don't let anyone else change any record I am currently selecting.

Level 4: Don't let anyone else change any record I have seen (within the current recoverable unit).

Level 5: Don't let me be aware of anyone else's existence at all (within the current recoverable unit).

Each level subsumes all preceding levels. This rule is slightly arbitrary - e.g., level 3 does not actually imply level 2 - but seems desirable for reasons of intellectual manageability. It leads to Implementation Principle #4: *The system can safely treat a request for a given level as a request for any higher level.* The default level in UDL is 3 (the final version of UDL will use keywords rather than integers, for readability and also because it simplifies later interpolation of intermediate levels).

The various levels merit some further discussion. Note first that *no* level allows the update of uncommitted changes (or "dirty data", as it is called in [6]). Level 1 does allow the transaction to see "dirty data", however. Not until we reach level 3 do we have "currency stability" - that is, a guarantee that once a record has been selected (via FIND), that record will not be changed by a concurrent transaction so long as a cursor remains positioned on it. At levels 1 and 2 an operation against a record may fail in various ways even though that record is current; for example, see Fig. 13. (These two levels may seem undesirable from the programmer's point of view, but in fact are exactly the levels supported, for various pragmatic reasons, in certain implemented systems.)

We have been tacitly assuming level 3 isolation in all our discussions prior to this point. Note that this level does not guarantee that the transaction will see the same value for a given record every time it accesses it: it is perfectly possible for a concurrent transaction to change the record (and commit the change) between two such accesses.

Transaction A	time	Transaction B
-		-
FIND R	t1	-
-	t2	FIND R
-	t3	delete R
-	t4	COMMIT
change R	t5	-
(NOT FOUND error)		-
-		-

Fig. 13: Transaction A has isolation level less than 3 for R baseset.

In level 4 the effect is as if every FIND included a KEEP option; if transaction A of Fig. 9 has been using level 4 the "inconsistent analysis" problem illustrated there would not have arisen. Level 4 thus does guarantee that the transaction (more accurately, recoverable unit) will see the same value every time it accesses a given record. But it does not handle the "phantom" problem [4], which we may illustrate as follows. Suppose that transaction T, operating at level 4 on the employee baseset, selects all employees with a salary $20000. None of these records can now be changed by a concurrent transaction (until T reaches a commit point). However, there is nothing to stop a concurrent transaction from creating a new employee with a salary of $20000, or updating an existing one so that the salary becomes $20000. If T now selects "employees with a salary of $20000" again it will discover a *phantom* - i.e., a record it did not see on the previous occasion.

Level 5 does handle the "phantom" problem. Level 5 means, loosely, that the transaction is not aware of the existence of any concurrent transactions, at least as far as this baseset is concerned (it differs from an X lock at the baseset level in that other transactions are not precluded from accessing portions of the baseset that this transaction has not touched). The intent of level 5 is that the transaction (more accurately, recoverable unit) should see no changes at all, except of course for changes made by the transaction itself (but see Note 4 below). Thus a request for employees with a salary of $20000 will select the same records every time it is executed (and this statement is true even in the special case where there are no such employees). The implication is that the "access path" used by the DBMS in selecting those records must be locked: for example, if an index on salaries is

used, the index entry for $20000 must be locked. This lock will prevent the creation of phantoms, because such creation would require the access path (the index entry, in our example) to be updated. In effect, then, avoiding phantoms requires the ability to lock the *nonexistence* of certain data (as suggested earlier in the "Inconsistent Analysis" section).

Aside: This discussion of level 5 touches on a point which is worth spelling out explicitly, viz. that all locking protocols described in this paper apply not only to records explicitly requested by the DBMS in responding to such explicit requests. For example, a search for the next employee record with salary = $20000 may have to wait on some prior employee record in sequence, even if that record does not have salary = $20000, simply because that prior record is locked by some other transaction. *End of aside.*

The record locking implications for the five isolation levels (in outline) are as follows:

Level 1: "Change R" implies: if R is locked, conflict; else acquire X lock on R. Release lock at COMMIT.

Level 2: (Level 1, plus) "FIND R" implies: if R is locked exclusive, conflict.

Level 3: (Level 2, plus) "FIND R" implies: acquire S lock on R; release when cursor set to new value.

Level 4: (Level 3, but) retain the S locks until COMMIT.

Level 5: (Level 4, plus) "FIND R" implies: acquire S lock on access path to R; retain until COMMIT.

A transaction operating at level 3 or less on some baseset may need to use explicit operators to acquire and release locks over and above those guaranteed by that level, in order to obtain an additional degree of integrity. In UDL these operators are KEEP and RELEASE (already mentioned under "Inconsistent Analysis", earlier). Their semantics are as follows:

- KEEP: acquire an S lock on the specified record.

For the moment let us agree to call an S lock acquired via KEEP (instead of implicitly) a K lock.

- RELEASE: release K lock on specified record (or all K locks on all records, if operand specified as ALL).

Note that RELEASE cannot be used to release lock acquired on behalf of the transaction by implicit action of the DBMS; that is, it cannot be used to undermine the guaranteed isolation level. COMMIT forces an implicit RELEASE of all K locks.

Note 1: Some systems provide a refinement on KEEP/RELEASE that we may call "keep classes". The basic idea is that KEEP can specify a "class identifier": the keep record is then logically appended to the indicated class (it may even be possible to keep the same record simultaneously in multiple classes and/or multiple times in the same class). An alternative form of RELEASE can then be used to release all records in a specified class.

Note 2: In some systems, the automatic locks on "current" records for level 3 and above are retained across COMMIT (other systems force all cursors to go null). Retention guarantees that cursor positioning is not lost at a commit point. Such locks can then be released when the cursor is set to a new value, as usual (isolation level 3 only); or the system may guarantee that the locks are retained even if a rollback occurs, in which case they cannot be released until the next commit point. (The locks on "kept" records could also be treated in this fashion.) In UDL a RETAIN option on COMMIT allows the transaction to indicate that it relies on the retention of such locks.

Note 3: In general a transaction may hold multiple locks, possibly of different types, on the same resource at the same time (again, not all systems permit this - but a general rule is that a transaction should not be able to lock *itself* out, so that the system must be prepared to deal *somehow* with the situation where a transaction requests a lock on something it already holds locked). Internally, therefore, a lock may be a fairly complex object, containing, for example, a count of the number of times the transaction has explicitly locked the resources (via KEEP) and another count of the number of times the DBMS has implicitly locked it on the transaction's behalf. A resource can then be considered locked if either count is nonzero. The formal semantics of KEEP, RELEASE, COMMIT, and other relevant operators can be defined appropriately in terms of these counts.

Note 4: Level 5 isolation is discussed in this paper because it is implemented (as "level 3") in at least one system, viz. System R. It may be considered a partial implementation of the general concept of "predicate locking" introduced in [4]. But it is a little difficult to give an informal characterization of this level that is both readily comprehensible and totally accurate (*formal* explanations are given in [6]). "The user sees the logical equivalent of a single-user system" [14]. But consider, for example, a system in which every change is timestamped in the database. Any given user can then certainly detect the presence of others by simply examining the timestamps.

Note 5: A transaction using isolation level less than 5 on some baseset always runs the risk of seeing inconsistent data: that is, serializability is not guaranteed (see the aside under "Inconsistent Analysis", earlier). Although there is no doubt that lower levels of isolation are valuable, it is not easy to say exactly what the result of using a given lower level will be in any given situation; nor is it easy to provide the application designer with guidance as to what level to use (if the system permits any choice).

12. NONRECOVERABLE DATA

So far we have assumed that all changes are logged, and hence that all data is "recoverable". For some data, however, the benefits of automatic recoverability are outweighed by the cost of such logging. The choice as to whether or not to log changes for a given baseset (assumed to be the unit for purposes of this discussion) must be made by the database administrator. Data for which logging is not done is said to be nonrecoverable. A nonrecoverable baseset can be accessed in either DEFERRED or IMMEDIATE mode. The mode is specified as a suboption of PROCESSINGLEVEL (CHANGE); DEFERRED is the default. (DEFERRED is the only possibility for a recoverable baseset.) IMMEDIATE means that changes by this transaction to the baseset are unlocked ("committed") as soon as they are made (possible since such changes can never be rolled back); DEFERRED means that changes remain locked exclusive until COMMIT. In other respects locking protocols are not affected. The DEFERRED/IMMEDIATE mode concept leads to Implementation Principle #5: *The system can safely implement a request for IMMEDIATE mode as a request for DEFERRED mode.*

A program using IMMEDIATE mode may suffer from the drawback (mentioned earlier) that its logic for retry after a rollback has to be quite complex.

13. UPDATE LOCKS

Some systems (IMS [8] is an exmaple) support an additional lock type at the record level, the update or U lock, with definition as given in Fig. 14.

	S	U	X
S	-	-	C
U	-	C	C
X	C	C	C

Fig. 14: Definition of U lock type

Intuitively, a request for a U lock (expressed in UDL via a HOLD option on FIND and KEEP) is a statement that the transaction may be going to update the record. A system supporting U locks will normally insist that a transaction acquire such a lock before being allowed to update the record. Such a rule permits slightly more concurrency than one that requires an X lock before updating. The U lock will be promoted to X level if the update actually occurs. Also, the holder of the U lock is guaranteed that it will be possible to perform the update (unless deadlock occurs), because no concurrent transaction will be able to acquire either a U or an X lock on the record.

14. SUMMARY

We have discussed the topic of locking and recovery from the standpoint of the application programmer. In doing so we have introduced a large number of concepts, some of the most important being:

- transactions and recoverable units;
- serializability;
- deadlock and rollback;
- shared and exclusive locks;
- lock granularity and intent locking;
- isolation levels.

We have established one "cardinal principle", viz. that no transaction may ever update an uncommitted change, and five implementation principles that can be used to ensure "safe" execution of any given transaction. The language UDL has been used as a vehicle for these discussions, since it

supports most of the concepts discussed. In conclusion, we list below the design objectives for the locking and recovery features of UDL:

- comprehensiveness;
- simplicity, especially for "normal" case;
- safe defaults (integrity before concurrency);
- system independence;
- extensibility.

ACKNOWLEDGEMENTS

This paper is the outcome of many hours of discussion with numerous colleagues in IBM, especially David Beech, Bob Engles, Jim Gray, Bill Kent, John Nauman, Phil Shaw, Irv Traiger, and Vern Watts. Each of these people also commented most constructively on early drafts of the paper. Further helpful comments were received from Phil Bernstein and Dave Shipman of Computer Corporation of America. Finally, thanks are due to the Future Database Language Project of GUIDE International, who were party to the design of the locking and recovery features of UDL and with whom I have had many fruitful discussions.

REFERENCES

[1] KING, P.F. & A.J. COLLMEYER, *Database Sharing - An Efficient Mechanism for Supporting Concurrent Processes*, Proc. NCC 1973.

[2] CHAMBERLIN, D.D., R.F. BOYCE & I.L. TRAIGER, *A Deadlock-Free Scheme for Resource Locking in a Data Base Environment*, Proc. IFIP Congress 1974.

[3] HAWLEY, D.A., J.S. KNOWLES & E.E. TOZER, *Database Consistency of the CODASYL DBTG Proposals*, Comp. J. 18, 3 (August 1975).

[4] ESWARAN, K.P., J.N. GRAY, R.A. LORIE & I.L. TRAIGER, *The Notions of Consistency and Predicate Locks in a Data Base System*, CACM 19, 11 (November 1976).

[5] GRAY, J.N., R.A. LORIE & G.R. PUTZOLU, *Granularity of Locks in a Large Shared Data Base*, Proc. Int. Conf. on VLDB (September 1975).

[6] GRAY, J.N., R.A. LORIE, G.R. PUTZOLU & I.L. TRAIGER, *Granularity of Locks and Degrees of Consistency in a Shared Data Base*, Proc. IFIP TC-2 Working Conference on "Modelling in Data Base Management Systems", ed. G.M. Nijssen (January 1976), North-Holland 1976.

[7] VERHOFSTAD, J.S.M., *Recovery Techniques for Database Systems*, Comp. Surv. 10, 2 (June 1978).

[8] McGEE, W.C., *The IMS/VS System*, IBM Sys. J. 16, 2 (June 1977).

[9] ENGLES, R.W., *A Description of the COBOL Data Base Facility*, Proc. GUIDE 47 (November 1978).

[10] DATE, C.J., *An Architecture for High-Level Language Database Extensions*, Proc. ACM SIGMOD Int. Conf. on Management of Data (June 1976).

[11] DATE, C.J., *An Architecture for High-Level Language Database Extensions: PL/I Version, Part I: Record-at-a-time Operations*, Proc. SEAS Anniversary Meeting (September 1977).

[12] DATE, C.J., *An Architecture for High-Level Language Database Extensions (Unified Database Language - UDL): PL/I Version*, (September 1978; a major revision of [11].) Available from the author.

[13] CODD, E.F. & C.J. DATE, *Interactive Support for Non-Programmers: The Relational and Network Approaches*, Proc. ACM SIGFIDET Workshop on Data Description, Access and Control, Vol. II (May 1974).

[14] ASTRAHAN, M.M. et al., *System R: A Relational Approach to Data Base Management*, TODS 1, 2 (June 1976).

[15] ENGLES, R.W., *Currency and Concurrency in the COBOL Data Base Facility*, Proc. IFIP TC-2 Working Conference on "Modelling in Data Base Management Systems", ed. G.M. Nijssen (January 1976), North-Holland 1976.

[16] BAYER, R., *Integrity, Concurrency and Recovery in Databases*, Proc. ECI Conference 1976: Lect. Notes in Comp. Sc., Vol. 44, Springer-Verlag (August 1976).

[17] ESWARAN, K.P. & D.D. CHAMBERLIN, *Functional Specifications of a Subsystem for Data Base Integrity*, Proc. Int. Conf. on VLDB (September 1975).

[18] NAUMAN, J.S., *Observations on Sharing in Data Base Systems*, IBM Tech. Report TR 03.047 (May 1978).

[19] ROSENKRANTZ, D.J., R.E. STEARNS & P.M. LEWIS II, *System Level Concurrency Control for Distributed Database Systems*, ACM TODS 3, 2 (June 1978).

[20] BERNSTEIN, P.A. & N. GOODMAN, *Approaches to Concurrency Control in Distributed Database Systems*, Tech. Report TR-26-78, Aiken Computation Laboratory, Harvard University (1978).

[21] GRAY, J.N., *Notes on Data Base Operating Systems*, IBM Research Report RJ2188, IBM San Jose Research Laboratory (February 1978).

[22] DAVIES, C.T. Jr., *Recovery Semantics for a DB/DC System*, Proc. ACM Nat. Conf. 1973.

[23] BJORK, L.A., *Recovery Scenario for a DB/DC System*, Proc. ACM Nat. Conf. 1973.

VERZAMELINGEN EN DATABASES

E.O. de BROCK
Technische Hogeschool, Eindhoven

0. INLEIDING

Er is de laatste jaren een steeds sterkere behoefte aan een stevige, zuiver wiskundige basis voor de theorie van databases, met name op het zogenaamde "conceptuele" niveau. Met het onderhavige verhaal doen we een poging gedeeltelijk in die behoefte te voorzien. De nadruk ligt hier niet op logica, zoals bij vele andere artikelen over de theorie van databases, maar op verzamelingenleer! We zullen dan ook beginnen met een hoofdstuk waarin we een aantal basisbegrippen uit de verzamelingenleer de revue nog eens laten passeren. De lezer, die zijn kennis van de verzamelingenleer nog verder wil opfrissen, verwijzen we naar VAN DALEN, DOETS en DE SWART [4].

Het "conceptuele" niveau, ons gebied van onderzoek, behoort implementatie-onafhankelijk te zijn. Efficiëntie, opslagstructuren etc., hoe interessant deze begrippen op zich ook zijn, horen dus niet op het "conceptuele" niveau thuis. Deze onderwerpen komen in dit verhaal dan ook niet aan de orde. We zullen nu een kort overzicht geven van de onderwerpen die wel worden behandeld.

Gegevens worden vaak gerepresenteerd door middel van tabellen. In hoofdstuk 2 wordt een formele definitie van "tabel" afgeleid. Op grond van die definitie zijn we in staat om ook de "heading" en de graad van een tabel formeel te definiëren, alsmede het begrip "attribuut van een tabel".

In hoofdstuk 3 worden enige operaties op tabellen behandeld. Naast de vereniging en de "orthogonale" operaties selectie en projectie van een tabel introduceren we de operatie die een tabel van een andere heading voorziet. De laatstgenoemde operatie heeft in de relationele aanpak (CODD [1]) geen analogon.

Een formele theorie van databases moet een duidelijk onderscheid maken tussen tabellen en zogenaamde "time-varying tables". Een "time-varying table" blijkt geen tabel te zijn! Een mogelijke formalisatie van de intuïtieve notie "time-varying table" is het in hoofdstuk 4 geïntroduceerde begrip *tabelvariabele* van een database-systeem. Ook voor een tabelvariabele kunnen we op zinvolle wijze de heading etc. definiëren.

In hoofdstuk 5 geven we eerst de definities van de begrippen "functionele afhankelijkheid" en "unieke identificatie", beide zowel voor tabellen als voor tabelvariabelen. In beide gevallen is unieke identificatie een belangrijk speciaal geval van functionele afhankelijkheid. In vragen (queries) en eisen (constraints) die we stellen aan een database-systeem, wordt vaak gesproken over "het element van de tabel T met ···", waarbij op de plaats van de stippen een "sleutel" en de bijbehorende attribuutwaarde(n) worden aangegeven. We kunnen "het element van T met ···" *formeel* beschrijven met behulp van *de identificatie-functie van* T. Die functie definiëren we met behulp van het eerder genoemde begrip "unieke identificatie" .

Om eisen *tussen* tabelvariabelen formeel te kunnen vastleggen, zullen we alle tabelvariabelen moeten "samennieten" tot één "grote" variabele, waarvan het universum dan een (door de zojuist genoemde eisen bepaalde) deelverzameling is van het "product" van de universa van elk van de oorspronkelijke tabelvariabelen. Een variabele waarvan het universum van de voornoemde vorm is, noemen we (onzes inziens terecht) een *database*. Een database is dus niet een verzameling tabelvariabelen, maar één variabele waarvan elke "component" een tabelvariabele "simuleert"!

De formele definities worden vaak voorafgegaan door een omschrijving in "alledaags" Nederlands. Deze omschrijvingen zijn meestal beter te "begrijpen", maar minder scherp dan de formele definities. De in te voeren begrippen zullen we meestal voor een zo groot mogelijke klasse definiëren (dat wil zeggen: we houden de vóóronderstellingen bij de definities zo zwak als redelijkerwijze mogelijk is), terwijl de lemma's die gevallen behandelen waar het ons uiteindelijk om gaat.

Het symbool □ wordt gebruikt om het einde van een voorbeeld of van een bewijs aan te geven.

Het onderzoek waaraan deze voordracht is gewijd wordt gedaan in samenwerking met drs. F. Remmen, prof.dr. W. Peremans en drs. M.L. Potters, allen van de TH Eindhoven.

1. ENIGE DEFINITIES UIT DE VERZAMELINGENLEER

Het begrip *verzameling* wordt bekend verondersteld. Als een verzameling wordt gedefinieerd door een eigenschap, dan kunnen we de verzameling als volgt aangeven:

(1) $\{x|\phi'(x)\}$,

d.w.z. de verzameling van alle x waarvoor de eigenschap $\phi(x)$ geldt. Als A een reeds eerder gegeven verzameling is, dan schrijven we in plaats van $\{x|x \in A \text{ en } \phi'(x)\}$ ook wel:

(2) $\{x \in A|\phi'(x)\}$.

De verzameling in (2) is verkregen door *selectie* uit de verzameling A. Een eindige verzameling kunnen we ook door *enumeratie* aangeven, bijvoorbeeld: [a,b,2,3] is de verzameling waarvan de elementen a, b, 2 en 3 zijn. Enumeratie en selectie zijn twee zeer verschillende manieren van vastleggen van verzamelingen! Daarom willen we de bijbehorende notaties ook duidelijk van elkaar onderscheiden en gebruiken we bij enumeratie rechte haakjes in plaats van de gebuikelijke accolades.

Volgens één van de axioma's van de verzamelingenleer zijn de verzamelingen A en B gelijk desda[1] ze dezelfde elementen hebben. Daaruit volgt bijvoorbeeld dat [a,b,a] = [a,b] en [a,b,c] = [c,b,a]: een verzameling kent geen duplicaten en geen ordening!

De definitie van *de vereniging van twee verzamelingen* wordt bekend verondersteld[2]. We zullen ook gebruik maken van de definitie van *de vereniging van een willekeurig aantal verzamelingen*, D1. Deze definitie is volgens LE1, punt 2), in zekere zin een generalisatie van de definitie van de vereniging van twee verzamelingen. Wij zijn hoofdzakelijk geïnteresseerd in de vereniging van minder dan twee verzamelingen; zie daarvoor LE1, punten 0) en 1).

D1: Als W een verzameling van verzamelingen is, dan:
$\underline{\cup W} = \{x \mid \exists A \in W: x \in A\}$.

LE1: Als A en B verzamelingen zijn, dan:
0) $\cup \emptyset = \emptyset$;
1) $\cup[A] = A$;
2) $\cup[A,B] = A \cup B$.

Evenals het begrip *verzameling* is in onze opzet ook het begrip *geordend paar* een primitief begrip. Geordende paren voldoen aan het volgende axioma, dat vertelt wanneer geordende paren gelijk zijn:

(3) $\langle x;y\rangle = \langle u;v\rangle$ desda $x = u$ en $y = v$.

We noemen x *de eerste coördinaat van* $\langle x;y\rangle$. *De tweede coördinaat van* $\langle x;y\rangle$ is y. Als p een willekeurig geordend paar is, dan duiden we de eerste coördinaat van p aan met $\pi_1(p)$ en de tweede coördinaat met $\pi_2(p)$.

Een *functie* is een verzameling geordende paren waarvan de eerste coördinaten onderling verschillend zijn.

D2: f is een <u>functie</u> desda f is een verzameling geordende paren en
$\forall p \in f: \forall p' \in f:$ als $\pi_1(p) = \pi_1(p')$, dan $\pi_2(p) = \pi_2(p')$.

De verzameling van alle eerste coördinaten in een functie f heet *het domein van* f. En *het bereik van* f, ook wel *de range van* f genoemd, is de verzameling van alle tweede coördinaten in f:

D3: Als f een functie is, dan:
a) $\underline{dom(f)} = \{\pi_1(p) \mid p \in f\}$;
b) $\underline{rng(f)} = \{\pi_2(p) \mid p \in f\}$.

Vaak wordt alleen maar het begrip "functie van A naar B" gedefinieerd (D4). Maar dankzij D2 zijn we gelukkig niet verplicht telkens twee verzamelingen expliciet te maken wanneer we over een functie willen spreken.

D4: Als A en B verzamelingen zijn, dan:
f is een <u>functie van A naar B</u> desda f is een functie en $dom(f) = A$ en $rng(f) \subseteq B$.

Als f een functie is en $x \in dom(f)$, dan is $\underline{f(x)}$ per definitie de (unieke) y waarvoor geldt $\langle x;y \rangle \in f$. Dus als $x \in dom(f)$, dan geldt: $y = f(x)$ desda $\langle x;y \rangle \in f$.

LE2 geeft een nuttig criterium voor gelijkheid van functies:

LE2: Als f en g functies zijn, dan:
$f = g$ desda $dom(f) = dom(g)$ en $\forall x \in dom(f)$: $f(x) = g(x)$.

Een *één-één-duidige functie*, ook wel "injectieve functie" of "injectie" genoemd, is een verzameling geordende paren waarvan de eerste coördinaten onderling verschillend zijn en bovendien de tweede coördinaten onderling verschillend zijn:

D5: f is een <u>één-één-duidige functie</u> desda f is een functie en
$\forall p \in f$: $\forall p' \in f$: als $\pi_2(p) = \pi_2(p')$, dan $\pi_1(p) = \pi_1(p')$.

D6: Als B een verzameling is, dan:
f is een <u>één-één-duidige functie op B</u> desda f is een één-één-duidige functie en $rng(f) = B$.

Een *set-functie* is een functie waarvoor geldt, dat elk element van het bereik van die functie een verzameling is:

D7: F is een <u>set-functie</u> desda F is een functie en $\forall a \in dom(F)$: F(a) is een verzameling.

Als f en g functies zijn en $x \in dom(f)$, dan kunnen we de functie f toepassen op x en we krijgen dan f(x) als resultaat. Als nu $f(x) \in dom(g)$, dan kunnen we vervolgens g toepassen op f(x), resulterend in g(f(x)). Op deze manier verkrijgen we $g \circ f$, *de samenstelling van* g *met* f:

D8: Als f en g functies zijn, dan:
$\underline{g \circ f} = \{\langle x;\ g(f(x))\rangle \mid x \in dom(f) \text{ en } f(x) \in dom(g)\}$.

Vaak wordt de samenstelling van g met f alleen maar gedefinieerd indien $rng(f) \subseteq dom(g)$, maar wij stellen ons iets liberaler op.

Figuur 5 in hoofdstuk 3 geeft een concreet voorbeeld van de samenstelling van twee functies, terwijl LE3 enige algemene eigenschappen van de in D8 gedefinieerde samenstellingen opsomt.

LE3: Als f en g functies zijn, dan:

a) $g \circ f$ is een functie;

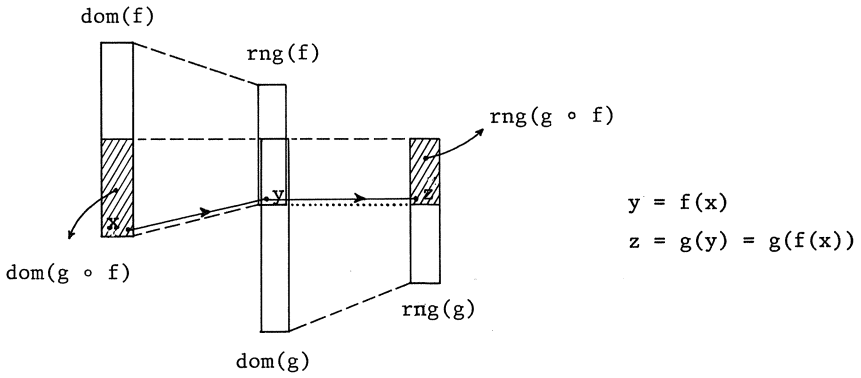
b) $dom(g \circ f) \subseteq dom(f)$ en $rng(g \circ f) \subseteq rng(g)$;

c) als $rng(f) \subseteq dom(g)$, dan $dom(g \circ f) = dom(f)$;

d) als $dom(g) \subseteq rng(f)$, dan $rng(g \circ f) = rng(g)$;

e) als h een functie is, dan $h \circ (g \circ f) = (h \circ g) \circ f$.

Figuur 1 illustreert D8 en een gedeelte van LE3:



Figuur 1.

Een deelverzameling van een functie is weer een functie:

LE4: Als f een functie is en $A \subseteq f$, dan:

a) A is een functie;

b) $dom(A) \subseteq dom(f)$ en $rng(A) \subseteq rng(f)$.

Uit dit eenvoudige maar belangrijke hulpresultaat volgt, dat voor alle functies f de verzamelingen van de vorm $\{p \in f | \phi(p)\}$ ook weer functies zijn. Dit betekent, informeel geformuleerd, dat "elke" eigenschap (selectie-criterium) ϕ een operatie op functies[3] bepaalt, namelijk de operatie die een functie f overvoert in de functie $\{p \in f | \phi(p)\}$! Een voorbeeld van zo'n operatie op functies is "..|B", de restrictie (van functies) tot een vaste verzameling B: *de restrictie van een functie* f *tot een verzameling* B is de verzameling van alle geordende paren in f waarvoor geldt dat de eerste coördinaat in B ligt:

D9: Als f een functie is en B een verzameling, dan:
$\underline{f|B} = \{p \in f | \pi_1(p) \in B\}$.

LE5 somt enige triviale eigenschappen van restricties op:

LE5: Als f een functie is en B een verzameling dan:
a) f|B is een functie;
b) f|B = f|(B ∩ dom(f));
c) dom(f|B) = B ∩ dom(f);
d) als B ⊆ dom(f), dan dom(f|B) = B;
e) f|∅ = ∅.

LE5 a) en e) zijn niet met elkaar in tegenspraak, want uit D2 volgt, dat ∅, de lege verzameling, ook een functie is. Het is in feite zelfs een één-één-duidige functie. Het is bovendien een set-functie.

2. TABELLEN

Gegevens worden vaak gerepresenteerd door middel van tabellen. Aan de hand van een nauwkeurige analyse van een voorbeeld zullen we een algemene wiskundige definitie van "tabel" vinden.

V1. Figuur 2 geeft een tabel weer van alle werknemers van een of ander bedrijf. Onder NRASS wordt geregistreerd het nummer van de assistent van de werknemer in kwestie. (Elke werknemer heeft, althans op papier, een assistent.)

NAME	NR	DPT	NRASS	SAL
Smith	52020202	D1	56112602	1700
Jones	40100504	D2	52020201	1915
Smith	52020201	D2	57112004	1700
Brown	29122201	D2	52020201	1960
Young	57112004	D2	52020201	1676
Cohen	56112602	D1	52020202	1655

Figuur 2.

In figuur 3 staat één van de rijen van de bovenstaande tabel, de rij die de werknemer Smith van afdeling 2 representeert.

NRASS	NR	DPT	NAME	SAL
57112004	52020201	D2	Smith	1700

Figuur 3.

De "kop" van de tabel is ook onderdeel van elke rij, opdat we bijvoorbeeld kunnen zien welk nummer van Smith zelf is en welk nummer van zijn assistent. Een rij bestaat dus niet alleen uit diverse waarden, maar ook uit de overeenkomstige attributen! We merken tenslotte op dat de in figuur 2 en in figuur 3 gekozen volgordes van de "kolommen" niet ter zake doen. □

Hoewel we voor representatie op papier een bepaalde rij-volgorde moeten kiezen is er conceptueel geen ordening van de rijen van een tabel. Duplicaten van rijen doen conceptueel evenmin ter zake. We kunnen vanwege deze twee eigenschappen een tabel beschouwen als een verzameling, een verzameling van "rijen". Zo is de tabel in figuur 2 een verzameling met 6 elementen.

Hoe moeten we nu het intuïtieve begrip "rij" in de betekenis zoals die hier gebruikt wordt, formaliseren? We zullen door abstractie van het voorbeeld in figuur 3 tot een antwoord komen.

De rij in figuur 3 bestaat uit geordende paren, bijvoorbeeld de geordende paren <DPT;D2>, <NAME;Smith> en <SAL;1700>. De eerste coördinaat van zo'n geordend paar is een attribuut en de tweede coördinaat is de bijbehorende waarde. Uit de laatste opmerking in V1 volgt, dat er conceptueel geen ordening tussen de paren in een rij is. En omdat in een tabel alle attributen verschillend zijn, zijn binnen een rij ook alle geordende paren verschillend. Dus een "rij" is een verzameling van geordende paren waarvan de eerste coördinaten onderling verschillend zijn. Dus formeel gezien is een "rij" een functie. Tenslotte constateren we, dat alle rijen in een tabel uit dezelfde verzameling attributen bestaan, dat wil zeggen, dat ze, als functies beschouwd, het zelfde domein hebben.

De voorafgaande analyse levert de volgende definitie van "tabel": een *tabel* is een verzameling functies, die allemaal het zelfde domein hebben. Formeel:

D10: T is een tabel desda T is een verzameling functies en
$\forall t \in T: \forall t' \in T: dom(t) = dom(t')$.

De idee om een "rij" te beschouwen als een functie, die aan elk attribuut de bijbehorende waarde toevoegt, is niet nieuw. Maar in tegenstelling tot vele andere schrijvers zullen wij deze zienswijze consequent aanhouden. Wij zijn van mening dat zowel de genoemde zienswijze als het consequent aanhouden ervan noodzakelijke voorwaarden zijn voor een vruchtbare ontwikkeling van de theorie van databases! De zienswijze, waarin een "rij" wordt beschouwd als een n-tupel (of als een functie met het "standaarddomein" [1,2,...,n]), schiet in twee belangrijke opzichten tekort. Enerzijds kan bij die zienswijze één tabel (in de intuïtieve betekenis) door verschillende wiskundige relaties worden gerepresenteerd, afhankelijk van de gekozen volgorde van de attributen[4]. Anderzijds representeert één wiskundige relatie een zeer grote klasse van tabellen: tabellen met verschillende "koppen", maar met dezelfde waarden, worden allemaal gerepresenteerd door dezelfde wiskundige relatie[5] en zijn dus ononderscheidbaar bij de "relationele zienswijze"![6]

Uit D10 blijkt dat we $\emptyset$, de lege verzameling, ook als een tabel willen beschouwen.

Dankzij D10 kunnen we nu vele intuïtieve begrippen formaliseren, zoals bijvoorbeeld de heading ("kop") en de graad van een tabel T. Ook de eigenschap 'attribuut van T' kan worden geformaliseerd.

Als T een niet-lege tabel is, dan hebben alle elementen van T het zelfde domein (D10). Dat gemeenschappelijke domein is in feite de "kop" van de tabel. De formalisatie van "de kop van T" noemen we *de heading van* T. Als heading van de lege tabel kiezen we de lege verzameling. Om beide gevallen in één keer te definiëren, maken we in D11 gebruik van de vereniging, gedefinieerd in D1. Uit LE6 volgt direct, dat de formele definitie overeenkomt met onze voorafgaande, informele omschrijving.

D11: Als T een tabel is, dan:
$\underline{\text{heading(T)}} = \cup\{\text{dom}(t) \mid t \in T\}$.

LE6: 0) $\text{heading}(\emptyset) = \emptyset$;
1) als T een tabel is en $t_0 \in T$, dan $\text{heading}(T) = \text{dom}(t_0)$.

LE6,0) volgt uit LE1,0), en LE6,1) volgt uit LE1,1) en de definitie van "tabel".

Een *attribuut van een tabel* is een element van de heading van de tabel, en *de graad van een tabel* is het aantal attributen van de tabel:

D12: Als T een tabel is, dan:
a is een attribuut van T desda a $\in$ heading(T).

D13: Als T een tabel is, dan:
graad (T) = # heading(T).

V2. De heading van de tabel in figuur 2 is de verzameling [DPT,NAME,NR, NRASS,SAL]. De tabel heeft dus graad 5. De tabel heeft 6 elementen ("rijen"); we zeggen ook wel: de cardinaliteit[7] van de tabel is 6. □

3. ENIGE OPERATIES OP TABELLEN

Een operatie die elke tabel weer overvoert in een tabel, noemen we een *operatie op tabellen*[8].

Een tabel is een verzameling, dus alle operaties die toepasbaar zijn op verzamelingen, zijn ook toepasbaar op tabellen. In §3.1 en §3.2 zullen we van een aantal operaties op verzamelingen nagaan in hoeverre het tevens operaties op tabellen zijn (in de zin zoals hiervoor omschreven).

In §3.3 en §3.4 behandelen we een aantal operaties die meer specifiek voor tabellen bestemd zijn.

3.1. Selectie

Een deelverzameling van een tabel is weer een tabel:

LE7: Als T een tabel is en A $\subseteq$ T, dan:
a) A is een tabel;
b) als A $\neq \emptyset$, dan heading(A) = heading(T).

Uit dit eenvoudige maar belangrijke hulpresultaat volgt, dat voor alle tabellen T de verzamelingen van de vorm $\{t \in T | \phi(t)\}$ ook weer tabellen zijn. Dit betekent, informeel geformuleerd, dat "elke" eigenschap (selectie-criterium) ϕ een operatie op tabellen bepaalt, namelijk de operatie die een tabel T overvoert in de tabel $\{t \in T | \phi(t)\}$! Een zeer speciaal voorbeeld van zo'n operatie op tabellen is de doorsnijding met een (vaste) verzameling B[9]; dit is namelijk het geval waarin $\phi(t) = (t \in B)$. Ook $\phi(t) = (t \notin B)$ levert een bekende operatie:

LE8: Als T een tabel is en B een verzameling, dan:
a) T $\cap$ B is een tabel;
b) T - B is een tabel.

3.2. Vereniging

Voor de vereniging van twee verzamelingen geldt het volgende algemene resultaat:

LE9: Als A en B verzamelingen zijn, dan:
A ∪ B is een tabel (desda A is een tabel en B is een tabel en, als $A \neq \emptyset$ en $B \neq \emptyset$, dan heading(A) = heading(B).

BEWIJS: Stel A ∪ B is een tabel; $A \subseteq A \cup B$ en $B \subseteq A \cup B$, dus zijn A en B tabellen volgens LE7a), en als A en B bovendien niet-leeg zijn, dan geldt heading(A) = heading(A ∪ B) = heading(B), dankzij LE7b). Omgekeerd, stel dat A en B tabellen zijn en dat als $A \neq \emptyset$ en $B \neq \emptyset$, dan heading(A) = heading (B); we controleren D10 voor A ∪ B: uiteraard is de vereniging van twee verzamelingen functies weer een verzameling functies; als $t \in A \cup B$ en $t' \in A \cup B$, dan zijn er 4 mogelijkheden: 1) $t \in A$ en $t' \in A$; 2) $t \in B$ en $t' \in B$; 3) $t \in A$ en $t' \in B$; 4) $t \in B$ en $t' \in A$.

1) dom(t) = heading(A) = dom(t') volgens LE6,1);
2) gaat analoog;
3) nu geldt $A \neq \emptyset$ en $B \neq \emptyset$, dus dan geldt ook heading(A) = heading(B);
nu weer LE6,1) toepassen: dom(t) = heading(A) = heading(B) = dom(t');
4) gaat analoog. □

LE10 volgt uit LE9 en spreekt waarschijnlijk meer aan:

LE10: Als T en T' niet-lege tabellen zijn dan:
a) T ∪ T' is een tabel desda heading (T) = heading (T');
b) als T ∪ T' een tabel is, dan heading (T ∪ T') = heading (T).

3.3. Projectie

Zoals het wegstrepen van een aantal "rijen" in een tabel - tabel in de intuïtieve zin - formeel resulteert in een deelverzameling van de formele tabel (selectie), zo resulteert het wegstrepen van een aantal "kolommen" van een tabel formeel in een zogenaamde "projectie" van de formele tabel. Selectie en projectie zijn dus in zekere zin "orthogonale" operaties. *De projectie van een tabel* T *op een verzameling* B is de verzameling van de restricties van alle elementen van T, tot B. We kunnen (en zullen) de projectie voor elke verzameling B en elke verzameling T van functies definiëren, niet

alleen voor B $\subseteq$ heading(T) en zelfs niet alleen voor een "homogene" verzameling functies:

D14: Als T een verzameling functies is en B een verzameling, dan:
$\Pi_B(T) = \{t|B|t \in T\}$.

V3. In figuur 4 staat de projectie van de tabel van figuur 2 op de verzameling [NAME,SAL].

SAL	NAME
1960	Brown
1915	Jones
1655	Cohen
1700	Smith
1676	Young

Figuur 4.

De projectie van de tabel van figuur 2 op [NAME,SAL] is weer een tabel; de graad van die tabel is 2. Er staan in figuur 4 maar 5 elementen, omdat de tabel in figuur 2 twee rijen met dezelfde NAME- en SAL-waarden heeft. En daar een projectie een verzameling is, vervallen eventuele duplicaten. □

De operatie $\Pi_B(..)$ is een operatie op tabellen (LE11a)) en ook een operatie op niet-lege tabellen[10] (volgt uit LE11b) "dan"). LE11b) "slechts dan" en LE11c) behandelen de speciale gevallen[11]. De heading van $\Pi_B(T)$ is gelijk aan B $\cap$ heading(T). Dus de enige essentiële gevallen zijn die waarvoor geldt B $\subseteq$ heading(T). In V3 zagen we, dat $\Pi_B(T)$ een kleiner aantal elementen kan hebben dan T. $\Pi_B(T)$ kan echter niet méér elementen hebben dan T. Resumerend:

LE11: Als T een tabel is en B een verzameling, dan:
a) $\Pi_B(T)$ is een tabel;
b) $\Pi_B(T) = \emptyset$ desda $T = \emptyset$;
c) $\Pi_\emptyset(T) = [\emptyset]$ desda $T \neq \emptyset$;
d) heading($\Pi_B(T)$) = B $\cap$ heading(T);
e) $\#\Pi_B(T) \leq \#T$.

3.4. Heading-transformatie

Wanneer wordt overgegaan van een "losse" verzameling bestanden naar een geïntegreerde database, dan zijn "corresponderende" attributen vaak ongelijk. Daarom willen we een tabel wel eens van een andere heading voorzien. Stel dat we de tabel T willen voorzien van de heading A. Er zal dan moeten gelden dat elk element van A één attribuut van T vervangt, dat verschillende elementen van A verschillende attributen van T vervangen en tenslotte dat élk attribuut van T wordt vervangen (eventueel door zichzelf). Dus we kunnen zo'n vervanging aangeven door een één-één-duidige functie op heading(T). Zo'n functie noemen we een *heading-transformator voor* T.

D15: Als T een tabel, dan:
h is een heading-transformator voor T desda h is een één-één-duidige functie op heading(T).

V4. Figuur 5 toont respectievelijk een heading-transformator H voor de tabel van figuur 2, een element t_1 van die tabel, en het overeenkomstige element in de nieuwe tabel.

H:

SAL	NUMMER	AFD	ASSNR	NAAM
SAL	NR	DPT	NRASS	NAME

t_1:

SAL	NR	DPT	NRASS	NAME
1655	56112602	D1	52020202	Cohen

SAL	NUMMER	AFD	ASSNR	NAAM
1655	56112602	D1	52020202	Cohen

Figuur 5.

Hoe kunnen we het nieuwe element uitdrukken in t_1 en H? H(AFD) = DPT en t_1(DPT) = D1, dus D1 = t_1(DPT) = t_1(H(AFD)) = $t_1 \circ$ H(AFD). Het zal nu duidelijk zijn, dat het nieuwe element de functie $t_1 \circ$ H is.

De nieuwe tabel is dus de verzameling van alle functies $t \circ H$, $t \in T_0$, waarbij T_0 de "oude" tabel van figuur 2 is. □

We geven de door de heading-transformator h uit T verkregen tabel aan met $T * h$. (Evenals D14 is D16 iets algemener dan strikt noodzakelijk is voor ons huidige interessegebied.)

D16: Als h een functie is en T een verzameling functies, dan:
$\underline{T * h} = \{t \circ h \mid t \in T\}$.

LE12: Als T een tabel is en h een heading-transformator voor T, dan:
a) als $t \in T$, dan $rng(h) = dom(t)$ en $dom(t \circ h) = dom(h)$ en $rng(t \circ h) = rng(t)$;
b) $T * h$ is een tabel;
c) $T * h \neq \emptyset$ desda $T \neq \emptyset$;
d) $heading(T * h) = dom(h)$.

BEWIJS: a) $Rng(h) = heading(T) = dom(t)$, volgens D6 en D15, resp. LE6,1); de rest volgt nu uit LE3,c) en d).
b) $T * h$ is een verzameling functies (LE3,a); als $t \in T$ en $t' \in T$, dan $dom(t \circ h) = dom(h) = dom(t' \circ h)$ volgens a); dus $T * h$ is een tabel.
c) Triviaal.
d) Stel $T = \emptyset$; dan $rng(h) = heading(T) = \emptyset$; maar als $rng(h) = \emptyset$, dan $h = \emptyset$, dus ook $dom(h) = \emptyset$; en als $T = \emptyset$, dan $T * h = \emptyset$, dus dan heading $(T * h) = \emptyset = dom(h)$; stel nu $T \neq \emptyset$; neem $t_1 \in T$, dan heading $(T * h) = dom(t_1 \circ h) = dom(h)$, volgens à). □

Volgens LE12 b) en d) is de operatie ".. * h" voor elke één-één-duidige functie h een operatie die tabellen met heading rng(h) overvoert in tabellen met heading dom(h).

4. TABELVARIABELEN

V5. In figuur 2 stonden alle werknemers per 1 juli 1979; in figuur 6 staan alle werknemers van het zelfde bedrijf per 1 januari 1980.

NAME	NR	DPT	NRASS	SAL
Smith	52020202	D1	60061701	1740
Jones	40100504	D2	57112004	1975
Brown	29122201	D2	57112004	2016
Young	57112004	D2	60061701	1720
Stone	60061701	D1	52020202	1680

Figuur 6.

□

In figuur 6 staat een andere tabel dan in figuur 2. We mogen niet zeggen, dat de tabel van figuur 2 "veranderd" is of een "andere waarde" gekregen heeft: tabellen veranderen niet en krijgen evenmin een andere "waarde" (dat geldt trouwens voor alle verzamelingstheoretische objecten). Toch is er een "samenhang" tussen de twee tabellen: elk representeerde de "actuele" verzameling werknemers. We kunnen die "samenhang" belichamen door een variabele (die we maar EMP zullen noemen): de tabel in figuur 2 is dan de waarde van de variabele EMP op 1 juli 1979 en de tabel in figuur 6 is dan de andere waarde van de zelfde variabele op 1 januari 1980. We moeten dus een duidelijk onderscheid maken tussen een tabel enerzijds en zo'n variabele anderzijds!

Om een en ander te kunnen formaliseren, gaan we eerst nader in op variabelen en op het begrip "database-systeem", waarvan we een gedeelte van de definitie (in wording[12]) zullen geven.

Een variabele is altijd een variabele vân iets, bijvoorbeeld van een zeker programma P of van een zeker database-systeem P. Eén van de componenten van P is de "variable declarations part" van P, waarin elke variabele van P opgenoemd wordt en een verzameling toegewezen krijgt, *de verzameling van de in* P *toegestane waarden voor die variabele*. Geen enkele variabele wordt meer dan eens gedeclareerd en de volgorde van de afzonderlijke declaraties doet niet ter zake. Dus formeel gezien is de "variable declarations part" van P een set-functie (D7), de functie die aan elke variabele de verzameling toegestane waarden voor die variabele toevoegt. Daarom zal de formele definitie van *database-systeem* de volgende vorm hebben:

P is een database-systeem desda P is een ..-tuple <F;G;...> waarvoor geldt:
F is een set-functie en
G is een ... en ... .

De eerste component van een database-systeem P zullen we aangeven met $P_{\text{1}\!\text{1}}$. Dus $P_{\text{1}\!\text{1}}$ representeert de "variable declarations part" van P.

Met behulp van het bovenstaande gedeelte van de definitie van 'database-systeem' zijn we in staat om de definities te geven die ons voor ogen staan. De definities, die gebruik maken van het begrip 'database-systeem', zijn van een sterretje voorzien. Deze "sterdefinities" zijn *relatief exact*, d.i. wanneer er een exacte definitie van 'database-systeem' voorhanden is, dan zijn ook de "sterdefinities" exact.

D^*17: Als P een database-systeem is, dan:
X is een variabele van P desda $X \in \text{dom}(P_{\text{1}\!\text{1}})$.

D^*18: Als P een database-systeem is en X een variabele van P, dan:
v is een in P toegestane waarde voor X desda $v \in P_{\text{1}\!\text{1}}(X)$.

We zullen $P_{\text{1}\!\text{1}}(X)$ wel eens *het universum van* X *in* P noemen.

Van een variabele zoals EMP eisen we, dat het universum, met daaruit weggelaten de lege tabel, een *homogene tabellenverzameling* is, d.w.z. een verzameling tabellen waarvan alle elementen dezelfde heading hebben. Een variabele die aan de genoemde eis voldoet, noemen we een *tabelvariabele van een database-systeem.*

D19: V is een homogene tabellenverzameling desda V is een verzameling tabellen en $\forall T' \in V$: heading(T) = heading(T').

D^*20: Als P een database-systeem is, dan:
E is een tabelvariabele van P desda E is een variabele van P en $P_{\text{1}\!\text{1}}(E) - [\emptyset]$ is een homogene tabellenverzameling.

Omdat alle niet-lege tabellen in het universum van een tabelvariable van een database-systeem dezelfde heading hebben, is het zinvol om ook *de heading van een tabelvariabele in* P te definiëren. Als het universum $P_{\text{1}\!\text{1}}(E)$ geen niet-lege tabellen heeft (d.i. $P_{\text{1}\!\text{1}}(E) \subseteq [\emptyset]$), dan $P_{\text{1}\!\text{1}}(E) = [\emptyset]$ of $P_{\text{1}\!\text{1}}(E) = \emptyset$. Als $P_{\text{1}\!\text{1}}(E) = [\emptyset]$, dan kiezen we als heading van E in P de heading van $\emptyset$, de enige tabel in $P_{\text{1}\!\text{1}}(E)$. Ook als $P_{\text{1}\!\text{1}}(E) = \emptyset$, dan kiezen we als heading van E in P de lege verzameling. We kunnen de drie gevallen in één keer afhandelen door in D^*21, evenals in D11, de vereniging uit D1 te gebruiken. LE13 toont nog eens aan, dat dat inderdaad goed gaat.

D^*21: Als P een database-systeem is en E een tabelvaria

$\underline{head_P(E)} = U\{heading(T) | T \in P_{\mathbb{1}}(E)\}$.

LE13: Als P een database-systeem is en E een tabelvariabele van P, dan:

0) als $P_{\mathbb{1}}(E) = \emptyset$, dan $head_P(E) = \emptyset$;

1) als $P_{\mathbb{1}}(E) = [\emptyset]$, dan $head_P(E) = \emptyset$;

2) als $T_0 \in P_{\mathbb{1}}(E) - [\emptyset]$, dan $head_P(E) = heading(T_0)$.

BEWIJS: LE13,i) volgt uit LE1,i) voor elke $i \in [0,1,2]$; aan i = 2 voegen we toe dat $head_P(E) = U[heading(T_0), heading(\emptyset)]$ als $\emptyset \in P_{\mathbb{1}}(E)$, en $head_P(E) = U[heading(T_0)]$ als $\emptyset \notin P_{\mathbb{1}}(E)$. □

D^*22: Als P een database-systeem is en E een tabelvariabele van P, dan:

a is een <u>attribuut van E in P</u> desda $a \in head_P(E)$.

D^*23: Als P een database-systeem is en E een tabelvariabele van P, dan:

$\underline{graad_P(E)} = \#\, head_P(E)$.

V6. We zullen een beschrijving geven van $P_{\mathbb{1}}(EMP)$, het universum van de variabele EMP in het (nog op te zetten) database-systeem P van het in V1 en V5 genoemde bedrijf.

$P_{\mathbb{1}}(EMP)$ zal onder meer de tabellen van figuur 2 en figuur 6 als element moeten hebben. Uit LE13,2) concluderen we dan ook dat $head_P(EMP)$ de verzameling [NAME,NR,DPT,SAL, NRASS] zal moeten zijn.

We zullen bij elk van de vijf attributen van EMP een verzameling aangeven, de verzameling van alle waarden die we toestaan voor het betreffende attribuut. Voor SAL staan we alle gehele waarden tussen 1200 en 6500 toe. Aan DPT voegen we de verzameling [D1,D2,D3,D4] toe. Voor NAME staan we alle strings (van characters) met lengte kleiner dan 21 toe. Aan NR en NRASS voegen we $[10^7,\ldots,10^8-1]$ toe. In feite hebben we een functie gespecificeerd, een set-functie met $head_P(EMP)$ als domein. We zullen deze functie F_0 noemen. Elk element van $P_{\mathbb{1}}(EMP)$ moet nu een tabel zijn waarbij elk element t $dom(F_0)$ als domein heeft en t bovendien voldoet aan $t(a) \in F_0(a)$ voor elk attribuut $a \in dom(t)$; zie (4).

Anderzijds zal niet elke tabel die aan bovenstaande voorwaarden voldoet een element van $P_{\mathbb{1}}(EMP)$ zijn; een tabel T moet aan een aantal extra "constraints" voldoen opdat ze een toegestane waarde voor EMP is: Het attribuut NR moet "uniek identificerend" zijn voor T, d.i. twee verschillende elementen van T moeten verschillende NR-waarden hebben; zie (5). Verder moet

uiteraard elk assistentnummer ook in de "NR-kolom" voorkomen, zie (6). De assistent van een employee moet een andere employee zijn; deze eis wordt uitgedrukt door (7). Hiermee hebben we alle eisen opgesomd die wij in dit voorbeeld aan een tabel T stellen opdat $T \in P_{\mathbb{1}}(EMP)$. Dus:

$P_{\mathbb{1}}(EMP) = \{T \mid T$ is een verzameling functies en

(4) $\forall t \in T: dom(t) = dom(F_0)$ en $(\forall a \in dom(t): t(a) \in F_0(a))$ en

(5) $(\forall t' \in T:$ als $t'(NR) = t(NR)$ dan $t' = t)$ en

(6) $(\exists u_t \in T: t(NRASS) = u_t(NR))$ en

(7) $t(NR) \neq t(NRASS)\}$.

Het is eenvoudig te controleren dat de tabellen van figuur 2 en figuur 6 inderdaad elementen zijn van de hierboven gedefinieerde $P_{\mathbb{1}}(EMP)$. □

Voor de formele beschrijving van sommige constraints is het nodig om "het element van T met NR-waarde w" formeel te kunnen uitdrukken. Dat kunnen we doen met behulp van *de identificatie-functie van een tabel*, die in het volgende hoofdstuk wordt gedefinieerd. In de definitie maken we gebruik van het begrip "unieke identificatie", een generalisatie van (5) en een speciaal geval van "functionele afhankelijkheid".

5. DE IDENTIFICATIE-FUNCTIE VAN EEN TABEL

We beginnen dit hoofdstuk met de definitie van functionele afhankelijkheid, eerst binnen een tabel en dan met betrekking tot een tabelvariabele.

Als T een tabel is en A en B verzamelingen (van attributen van T) zijn, dan is B *functioneel afhankelijk van* A *binnen* T desda twee elementen van T met verschillende restricties tot B ook verschillende restricties tot A hebben. Als E een tabelvariabele van een database-systeem P is, dan is B *functioneel afhankelijk van* A *met betrekking tot* E *in* P desda B is f.a. (functioneel afhankelijk) van A binnen elke $T \in P_{\mathbb{1}}(E)$.

D24: Als T een tabel is en A en B zijn verzamelingen, dan:
B is functioneel afhankelijk van A binnen T desda
$\forall t \in T: \forall t' \in T:$ als $t|A = t'|A$ dan $t|B = t'|B$.

D*25: Als P een database-systeem is en E een tabelvariabele van P, en A en B zijn verzamelingen, dan:
B is functioneel afhankelijk van A met betrekking tot E in P desda
$\forall T \in P_{\Pi}(E)$: B is functioneel afhankelijk van A binnen T.

Wanneer er sprake is van één vast database-systeem, dan kan de toevoeging 'in P' vaak worden weggelaten.

We zijn uiteindelijk geïnteresseerd in functionele afhankelijkheden met betrekking tot tabelvariabelen (D*25), maar om die afhankelijkheden te verwezenlijken zullen we bij de beschrijving van de verzameling toegestane waarden voor een tabelvariabele, d.i. in de declaratie van die variabele, gebruik moeten maken van D24 (functionele afhankelijkheden binnen tabellen).

V7. Het bedrijf uit V6 wil ook gegevens over klanten bijhouden. Daartoe wordt er een variabele KL geïntroduceerd. KL heeft onder meer de attributen ADDR, CITY en PC. Nu is het in de "werkelijkheid" zo, dat adres plus woonplaats de postcode eenduidig bepaalt. En de bedrijfsleiding wil, dat dat niet alleen in de werkelijkheid geldt, maar ook in hun database-systeem[13]. We moeten dus zorgen dat [PC] functioneel afhankelijk is van [ADDR,CITY] met betrekking tot KL in het database-systeem P van het bedrijf. Dat realiseren we door van elke tabel te eisen dat [PC] functioneel afhankelijk is van [ADDR,CITY] binnen die tabel. Dus $P_{\Pi}(KL)$ heeft de vorm

$\{T \mid T$ is een tabel en ([PC] is f.a. van [ADDR,CITY] binnen T) en $\phi\}$,

waarbij ϕ de resterende eisen representeert. □

Wanneer we het over functionele afhankelijkheden hebben, dan moeten we altijd duidelijk aangeven of we bedoelen *binnen een tabel* of *met betrekking tot een tabelvariabele*; het weglaten van die toevoeging geeft snel aanleiding tot misverstanden.

We sommen nu een aantal eenvoudige eigenschappen op:

LE14: Als P een database-systeem is en E een tabelvariabele van P en A, B en C zijn verzamelingen, dan:
a) B is f.a. van A m.b.t. E in P desda $\forall b \in B$: [b] is f.a. van A m.b.t. E in P;
b) als C is f.a. van A m.b.t. E in P en $A \subseteq B$, dan C is f.a. van B m.b.t. E in P;

c) als $A \subseteq B$, dan A is f.a. van B m.b.t. E in P;

d) als C is f.a. van B m.b.t. E in P en B is f.a. van A m.b.t. E in P, dan C is f.a. van A m.b.t. E in P;

e) B is f.a. van $\emptyset$ m.b.t. E in P desda $\forall T \in P_{1\!\!1}(E) : \# \Pi_B(T) \leq 1$;

f) $\emptyset$ is f.a. van A m.b.t. E in P.

Als we in LE14 "m.b.t. E in P" vervangen door "binnen T" en in e) "$\forall t \in P_{1\!\!1}$ (E):" weglaten, dan verkrijgen we ook een geldig (maar minder belangrijk) lemma.

LE15 geeft het verband tussen *functionele* afhankelijkheid (binnen een tabel) en functies. Het lemma volgt rechtstreeks uit D2 en D24.

LE15: Als T een tabel is en A en B zijn verzamelingen, dan:
B is f.a. van A binnen T desda $\{<t|A;t|B>|t \in T\}$ is een functie.

Een belangrijk speciaal geval van functionele afhankelijkheid (zowel van D24 als van D*25) is "unieke identificatie".

Als T een tabel is en A een verzameling (van attributen van T), dan is A *uniek identificerend voor* T desda twee verschillende elementen van T ook verschillende restricties tot A hebben. A *heeft de unieke identificatie eigenschap voor een tabelvariabele* E *in een database-systeem* P desda A is u.i. (uniek identificerend) voor elke $T \in P_{1\!\!1}(E)$.

D26: Als T een tabel is en A een verzameling, dan:
A is uniek identificerend voor T desda $\forall t \in T: \forall t' \in T:$ als $t|A = t'|A$ dan $t = t'$.

D*27: Als P een database-systeem is en E een tabelvariabele van P en A een verzameling, dan:
A heeft de unieke identificatie eigenschap voor E in P desda
$\forall T \in P_{1\!\!1}(E)$: A is uniek identificerend voor T.

Voorwaarde (5) in V6 is dus dat [NR] uniek identificerend is voor T. Deze voorwaarde blijkt te gelden voor élke $T \in P_{1\!\!1}(EMP)$, dus heeft [NR] de u.i.e. (unieke identificatie eigenschap) voor EMP in P.

LE16: Als P een database-systeem is en E een tabelvariabele van P en A en B zijn verzamelingen, dan:

a) A heeft de u.i.e. voor E in P desda $head_P(E)$ is f.a. van A m.b.t. E in P;

b) als A de u.i.e. voor E in P heeft en $A \subseteq B$, dan heeft B de u.i.e. voor E in P;

c) $head_P(E)$ heeft de u.i.e. voor E in P;

d) als B de u.i.e. voor E in P heeft en B is f.a. van A m.b.t. E in P, dan heeft A de u.i.e. voor E in P;

e) $\emptyset$ heeft de u.i.e. voor E in P desda $\forall T \in P_{\Pi}(E)$: $\#T \leq 1$.

LE16a) volgt uit LE13,2) en LE6,1) en het feit dat $t|dom(t) = t$ voor elke functie t. LE16 b) tot en met LE16 e) volgen uit de overeenkomstige onderdelen van LE14, met behulp van de "substitutie-regel" LE16 a).

Evenals LE14 heeft ook LE16 een geldig analogon voor tabellen, zie LE17, a) tot en met e). In LE17 f) behandelen we nog een paar speciale gevallen.

LE17: Als T een tabel is en A en B zijn verzamelingen, dan:

a) A is u.i. voor T desda heading(T) is f.a. van A binnen T;

b) als A u.i. is voor T en $A \subseteq B$, dan is B u.i. voor T;

c) heading(T) is u.i. voor T;

d) als B u.i. is voor T en B is f.a. van A binnen T, dan is A u.i. voor T;

e) $\emptyset$ is u.i. voor T desda $\#T \leq 1$;

f) A is u.i. voor $\emptyset$ en A is u.i. voor $[\emptyset]$.

LE17 a) volgt uit LE6,1) en het feit dat $t|dom(t) = t$ voor elke functie t. LE17 b) tot en met LE17 e) volgen uit de overeenkomstige onderdelen van het "tabelanalogon" van LE14, met behulp van de "substitutieregel" LE17 a).

Dat "unieke identificatie" een speciaal geval is van functionele afhankelijkheid, zoals van tevoren al is aangekondigd, blijkt misschien niet direct uit de definities, maar wel uit LE16 a) en LE17 a).

Ook unieke identificatie (voor een tabel) kan worden geformuleerd in termen van functies:

LE18: Als T een tabel is en A een verzameling, dan:

A is u.i. voor T desda $\{<t|A;t> \mid t \in T\}$ is een functie.

LE18 volgt rechtstreeks uit D2 en D26. Als A u.i. is voor T, dan is de in LE18 genoemde functie een één-één-duidige functie op T (zie D6), met $\Pi_A(T)$ als domein.

Laten we $\{<t|A;t>|t \in T\}$ even aangeven met $F_{T;A}$. Als we voor een (vaste) tabel T alle functies $F_{T;A}$, met $A \subseteq$ heading(T) en A is u.i. voor T, met elkaar verenigen, dan resulteert dat in een "grote" verzameling van geordende paren. Die "grote" verzameling blijkt ook een functie te zijn! We zullen $\uparrow_T$, die met de tabel T geassocieerde functie, *de identificatie-functie van* T noemen.

D28: Als T een tabel is, dan:

$\uparrow_T = \{<x;t>|t \in T$ en $\exists A \subseteq$ heading(T): (A is u.i. voor T en $x = t|A)\}$.

ST1: Als T een tabel is, dan is $\uparrow_T$ een functie.

BEWIJS: We zullen controleren of $\uparrow_T$ aan D2 voldoet:
Het is duidelijk dat $\uparrow_T$ een verzameling geordende paren is. Laat nu $<x;t> \in \uparrow_T$ en $<x';t'> \in \uparrow_T$; dan zijn er deelverzamelingen A en A' van heading(T), die u.i. voor T zijn en waarvoor geldt $x = t|A$ en $x' = t'|A'$; $dom(x) = dom(t|A) = A \cap dom(t) = A \cap$ heading(T) = A; evenzo $dom(x') = A'$; als nu $x = x'$, dan $A = dom(x) = dom(x') = A'$, dus dan $t'|A = t'|A' = x' = x = t|A$, dus $t' = t$ volgens D26. Dus $\uparrow_T$ voldoet aan D2. □

Het belang van ST1 ligt in het feit dat we met elke tabel T maar <u>één functie</u> ($\uparrow_T$) hoeven te associëren in plaats van een hele <u>verzameling functies</u> ($\{F_{T;A}|A \subseteq$ heading(T) en A is u.i. voor T$\}$), zoals misschien door LE18 wordt gesuggereerd.

Volledigheidshalve geven we van $\uparrow_T$ ook een omschrijving met behulp van "alledaags" Nederlands: $\uparrow_T$ is de functie die aan elk element van elke projectie van T op een (voor T) uniek identificerende deelverzameling van heading(T), het bijbehorende element van T toevoegt. Als $x \in dom(\uparrow_T)$, dan noemen we dat bij x behorende element van T *de completering van* x *binnen* T; dus $\uparrow_T(x)$ is het (unieke) element van T waarvan x een "gedeelte" - om precies te zijn: restrictie (of ook: deelverzameling) - is. De voorwaarde '$x \in dom(\uparrow_T)$' hebben we in LE19 nader uitgeschreven:

LE19: Als T een tabel is, dan:

$x \in dom(\uparrow_T)$ desda $\exists t \in T$: $\exists A \subseteq$ heading(T): A is u.i. voor T en $x = t|A$.

Bij LE19 kunnen we nog opmerken dat de daar genoemde t en A uniek zijn voor $x \in dom(\uparrow_T)$.

Met behulp van identificatie-functies kunnen we constraints formuleren die we zonder die functies niet of nauwelijks formeel zouden kunnen

beschrijven (zie V8). Behalve bij het formeel beschrijven van constraints zijn identificatie-functies ook nodig bij het formeel beschrijven van "queries". In beide gevallen betreft het situaties waarin wordt *gesproken* over "het element van T met ...", waarbij op de plaats van de stippen een "sleutel" en de bijbehorende attribuutwaarde(n) worden aangegeven. Bijvoorbeeld, de formele beschrijving van "het element van T met NR-waarde w" is: $\uparrow_T([<NR;w>])$.

V8. Als er in het bedrijf van V6 ook nog wordt geëist dat de assistent van een employee op de zelfde afdeling moet werken, dan moeten we in de daar gegeven beschrijving van $P_{\mathbb{1}}(EMP)$ nog voorwaarde (8) toevoegen:

(8) $\quad \forall t \in T\colon t(DPT) = \uparrow_T([<NR;t(NRASS)>])(DPT).$

Deze constraint kunnen we overigens wél zonder gebruikmaking van de identificatie-functie $\uparrow_T$ formuleren, en wel door (6) in de beschrijving van $P_{\mathbb{1}}(EMP)$ te vervangen door de eis

$$\exists u_t \in T\colon (t(NRASS) = u_t(NR) \text{ en } t(DPT) = u_t(DPT)).$$

Deze oplossing is mogelijk omdat $\uparrow_T([<NR;t(NRASS)>])$ juist de in (6) genoemde u_t is. □

6. DATABASES

V9. Het in onze voorbeelden genoemde bedrijf wil niet alleen gegevens van werknemers en klanten bijhouden, maar ook gegevens van de afdelingen en de orders. De "variable declarations part" van het database-systeem van dat bedrijf zou er dus als volgt uit kunnen zien:

```
VAR  EMP : W1,
     KL  : W2,
     DEP : W3,
     ORD : W4
END.
```

Figuur 7.

(We hebben geordende paren <X;Y> geschreven als 'X:Y'.) Elk van de tabellenverzamelingen W1, W2, W3 en W4 is zó gekozen, dat aan alle op te leggen constraints *per attribuut* (bijvoorbeeld: salaris tussen 1200 en 6500, zie V6), *tussen attributen* (bijvoorbeeld (7) in V6) en *per tabelvariabele* (bijvoorbeeld (5) en (6)) is voldaan. Eisen die we met deze opzet niet kunnen vastleggen zijn bijvoorbeeld dat "op elk moment" elke in EMP genoemde afdeling ook daadwerkelijk in DEP voorkomt en dat elke in DEP genoemde manager volgens EMP (d.i. volgens het attribuut DPT aldaar) ook bij die afdeling hoort. Dit zijn voorbeelden van eisen *tussen* tabelvariabelen. (Voor een nauwkeuriger omschrijving van de genoemde classificatie verwijzen we de lezer naar DE BROCK [3].) □

Voordat we nader ingaan op de vraag hoe we constraints tussen tabelvariabelen moeten vastleggen geven we eerst een paar definities die we nodig hebben bij onze verdere uitleg.

Het product van een set-functie F is de verzameling van alle functies f die hetzelfde domein hebben als F en waarvoor bovendien voor elke $a \in dom(f)$ geldt dat $f(a) \in F(a)$:

D29: Als F een set-functie is dan:
$\underline{X(F)} = \{f \mid f \text{ is een functie en } dom(f) = dom(F) \text{ en } \forall a \in dom(f): f(a) \in F(a)\}$.

Voorwaarde (4) in V6 is dus, dat $T \subseteq X(F_0)$. Een deelverzameling van het product van een set-functie F noemwen we wel een *tabel over* F:

D30: Als F een set-functie is, dan:
T is een <u>tabel over F</u> desda $T \subseteq X(F)$.

Hoewel het niet expliciet in de definities wordt geëist, is een tabel over F inderdaad een tabel (zoals de naamgeving al deed vermoeden). Verder is de heading van een niet-lege tabel over F gelijk aan dom(F):

LE20: Als F een set-functie is en T een tabel over F, dan:
a) T is een tabel;
b) als $T \neq \emptyset$, dan heading(T) = dom(F).

Eisen tussen tabelvariabelen impliceren dat de waarden van de tabelvariabelen niet onafhankelijk van elkaar kunnen variëren. Met andere woorden: bepaalde combinaties van elementen van elk van de Wi's (zie figuur 7) zijn

niet toegestaan. Dus, als G_0 de in figuur 7 uitgebeelde set-functie is, dan wordt elke "toestand" van de database gerepresenteerd door een element van $X(G_0)$, maar niet elk element van $X(G_0)$ representeert ook een toe te stane toestand van de database.

Ten einde eisen tussen tabelvariabelen formeel te kunnen opnemen in een database-systeem, zullen we een database moeten beschouwen als één "grote" variabele waarvan het universum U een deelverzameling is van X(G); hierbij is G de functie die aan elke "vroegere" tabelvariabele E het "vroegere" universum van E toevoegt. We hebben het adjectief "vroegere" toegevoegd, omdat de *status* van E is veranderd (E zelf niet): E is nu geen variabele van het database-systeem meer, maar een "onderdeel" van een andere variabele D. We geven een voorbeeld.

V10. Figuur 8 illustreert de nieuwe opzet van het database-systeem voor ons bedrijf. We laten de "variable declarations part" vooraf gaan door een "set definitions part" waarin we (complexe) verzamelingen een naam kunnen geven. Met behulp van "set definitions" kunnen we verzamelingen stap voor stap "opbouwen", zodat alles enigszins begrijpelijk kan blijven.

```
SET  .
     .
     .
     PRDG = PRODUCT  EMP : W1,
                     KL  : W2,
                     DEP : W3,
                     ORD : W4
            END;
     U = {v ∈ PRDG | φ(v)};
VAR  D : U  END.
```

Figuur 8.

De variabele D is een voorbeeld van een *database* zoals in D*31 gedefinieerd. Constraints tussen "vroegere" tabelvariabelen worden in figuur 8 gerepresenteerd door '$\phi(v)$'. We zullen ter illustratie twee fragmenten van '$\phi(v)$' bepalen door het formaliseren van de beide in V9 genoemde eisen.

DEP heeft het attribuut DNR voor het afdelingsnummer en het attribuut MAN voor het emplo@yeenummer van de manager van de afdeling. De eerstgenoemde

eis in V9 levert het volgende fragment van '$\phi(v)$' op:

$$\forall t \in v(EMP):[<DNR;t(DPT)>] \in \Pi_{[DNR]}(v(DEP)).$$

De tweede eis levert op:

$$\forall t \in v(DEP): t(DNR) = \uparrow_{v(EMP)} ([< NR;t(MAN)>])(DPT).$$

Voor een volledig uitgewerkt voorbeeld verwijzen we de geïnteresseerde lezer naar (de appendix van) DE BROCK [3]. □

We zullen een formele definitie geven van het begrip "database" zoals het in het voorafgaande al informeel gebruikt werd. Zo'n "database" is een variabele waarvan het universum van een bepaalde vorm is. Wat is nu de algemene vorm van een universum van een willekeurige "database"? Het antwoord staat verscholen in de alinea voorafgaande aan V10. Een database-universum U is een deelverzameling van het product van een set-functie, dus een tabel. (Elke tabel is altijd een tabel over een of andere set-functie, en omgekeerd.) De heading van die tabel U is juist de verzameling "vroegere" tabelvariabelen. Uit D^*20 volgt dan ook de eis dat voor elke $E \in$ heading(U) de verzameling $\{v(E) | v \in U\} - [\phi]$ een homogene tabellenverzameling moet zijn. Andere eisen aan U zijn er niet. Dus:

D^*31: Als P een database-systeem is, dan:
D is een <u>database van P</u> desda D is een variabele van P en $P_{\text{ll}}(D)$ is een tabel en $\forall E \in$ heading($P_{\text{ll}}(D)$) : $\{v(E) | v \in P_{\text{ll}}(D)\} - [\emptyset]$ is een homogene tabellenverzameling.

Uiteraard zullen in de praktijk de meeste database-systemen slechts één database hebben, maar er bestaan database-systemen (van dienstverlenende computercentra) die meer dan één database bevatten.

NOTEN

1. Dan en slechts dan als.
2. De definitie van de *doorsnede* van twee verzamelingen wordt ook bekend verondersteld. Met A-B geven we het *verschil* van A en B aan. Dus: $A-B = \{x \in A | x \notin B\}$.

3. Een operatie die elke functie over voert in een functie, noemen we een *operatie op functies*. In het algemeen: Als 'Q' een zelfstandige-naamwoordsgroep is en 'Qs' (lees "kuus", niet "kuu-es") de bijbehorende meervoudsvorm, dan is een *operatie op* Qs een operatie die elke Q overvoert in een Q. In het onderhavige geval: 'Q' = 'functie' en 'Qs' = 'functies'. Zie ook de noten 8 en 10.
   Andere voorbeelden van operaties op functies zijn de operaties "h∘.." en "..∘h", voor elke functie h; zie LE3 a).
4. Het is op zich al "onnatuurlijk" dat er een volgorde moet worden *gecreëerd*.
5. Eigenlijk: door dezelfde *verzameling* wiskundige relaties; zie het eerstgenoemde bezwaar.
6. Het eerstgenoemde bezwaar is op zijn minst "onaangenaam", maar het tweede is vanuit theoretisch oogpunt echt een fundamentele tekortkoming: Bij een wiskundig object moet diens intuïtieve tegenhanger "éénduidig" vast liggen; wiskundig(er) gezegd: de "representatie-relatie" tussen wiskundige, resp. intuïtieve objecten moet een functie zijn. En het is op zijn minst "aangenaam" dat bij een intuïtief object diens formele tegenhanger ook een eenduidig vast ligt; wiskundig(er) gezegd: de zojuist genoemde functie moet bij voorkeur één-één-duidig zijn. (Ten slotte moeten we ook eisen dat elk intuïtief object daadwerkelijk een formele tegenhanger hééft, dat wil zeggen, dat de genoemde functie een (één-één-duidige) functie op de verzameling (eigenlijk: "klasse") intuïtieve objecten is.)
7. Het aantal elementen van een verzameling A (*de cardinaliteit van* A) geven we aan met #A.
8. In feite een speciaal geval van de algemene "definitie" in noot 3: neem 'Qs' = 'tabellen' en 'Q' = 'tabel'.
9. Ook de THETA-SELECT (zie CODD [2]) is een zeer speciaal voorbeeld van een operatie op tabellen, geïnduceerd door één of ander selectie-criterium.
10. Neem in noot 3 'Q' = 'niet-lege tabel' en 'Qs' = 'niet-lege tabellen'.
11. Merk op dat er een verschil is tussen ∅, de lege tabel, en [∅], de tabel met de functie ∅ als enige element.
12. Omdat op het gebied van databases het begrip *database-systeem* "alles" omvat, kan dat begrip pas als laatste van alle begrippen volledig worden geformaliseerd. En zover zijn we nog lang niet...

13. Niet elke "regel" die in de werkelijkheid geldt hoeft als constraint in het database-systeem te worden opgenomen! Die "regels" waarvan we persé willen dat ze ook in ons database-systeem gelden, moeten we uiteraard wel opnemen als constraint, maar de andere "regels" zouden we kunnen negeren. Want bedenk, dat elke opgenomen constraint ook telkens moet worden gecontroleerd! Van elke "regel" moeten de voor- en nadelen van het opnemen als constraint worden afgewogen.

LITERATUUR

[1] CODD, E.F., *A Relational Model of Data for Large Shared Data Banks*, Comm. A.C.M. 13 (1970), 377-387.

[2] CODD, E.F., *Extending the Database Relational Model to Capture More Meaning*, ACM Trans Database Syst. 4 (1979), 397-434.

[3] BROCK, E.O. DE, *Tables, Database Tables, and Static Integrity Constraints*, Memorandum 80-12, onderafdeling der wiskunde, TH Eindhoven, 1980

[4] DALEN, D. VAN, H.C. DOETS & H.C.M. DE SWART, *Verzamelingen: naïef, axiomatisch en toegepast*, Oosthoek, Scheltema en Holkema, 1975.

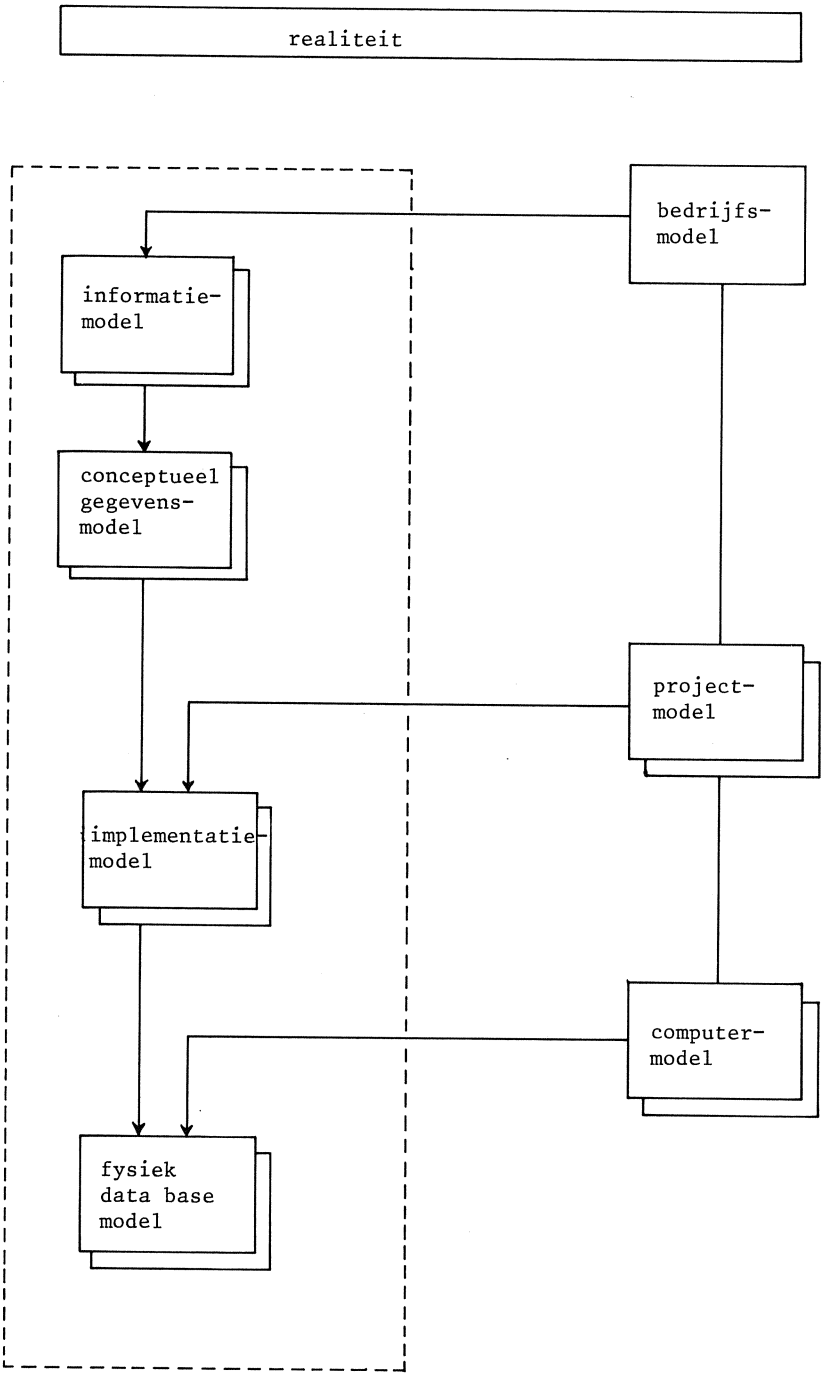
DATA BASE ONTWERP

J.A. VANDENBULCKE
Katholieke Universiteit, Leuven

1. INLEIDING

De oorsprong van een informatiesysteem heeft te maken met een informatiebehoefte omtrent de *realiteit*. Men dient er zich nochtans van bewust te zijn dat men slechts die aspecten van de werkelijke realiteit kan opvangen die door onze perceptoren kunnen worden waargenomen. Overigens is het zo dat onze perceptoren fragmentarische en sterk gepersonaliseerde informaties naar ons zenuwstelsel doorsturen, zodat wij kunnen verwachten dat er een belangrijk onderscheid bestaat tussen de werkelijke en de waargenomen realiteit. In een menselijke organisatie zoals een *onderneming* of *bedrijf* werken een aantal personen samen in het kader van het realiseren van de doelen waartoe de organisatie is opgericht. Hun informatiebehoeften houden verband met de functie die zij uitoefenen en hebben betrekking op de organisatie en haar omgeving. Deze informatiebehoeften kunnen door middel van informatie-elementen en informatie-structuur elementen worden samengevat in een aantal *informatiesubsysteemmodellen*. Vanaf deze modellen kunnen *conceptuele gegevensmodellen* worden gebouwd die de gegevens en de gegevensstructuren incorporeren dewelke nodig zijn om de onderkende informatie af te leveren. De conceptuele gegevensmodellen vormen de operands waarop een aantal *projectmodellen* zich ten behoeve van het verkrijgen van informaties gaan beroepen. Conceptuele gegevensmodellen die karakteristieken inhouden omtrent de wijze waarop projectmodellen de gegevens-resource wensen te gebruiken zijn *implementatie-modellen*. Deze moeten, rekening houdend met het beschikbare *computermodel*, worden omgezet in performante *fysieke modellen* die aan computergeheugenmedia worden toegewezen (figuur 1).

Met deze context als achtergrond gaan wij vervolgens een methodologie voorstellen voor het ontwerpen van conceptuele gegevensmodellen en voor het omzetten van deze modellen naar fysieke data base modellen.



FIGUUR 1. Situering van de problematiek in verband met het ontwerpen en het omzetten van conceptuele gegevensmodellen.

2. KENMERKEN VAN DE VOORGESTELDE METHODE VOOR HET ONTWERPEN EN OMZETTEN VAN EEN CONCEPTUEEL GEGEVENSMODEL

Het traditionele data base ontwerp was alleen maar gericht op een efficiënte DBMS-implementatie van een gebruikers-applicatie. Data base ontwerpers hadden weinig of geen aandacht voor het opsporen van relevante informatiebehoeften in verband met het onderkennen van bepaalde gebruikerszienswijzen en gebruikerskarakteristieken bij het behandelen van informatie, etc. Het onderzoeken van al deze 'noden' werd beschouwd als overbodig en erg tijdrovend, etc.

Organisaties die deze weg van data base ontwerp zijn ingeslagen zien zich geconfronteerd met steeds opnieuw opduikende problemen in verband met het uitbreiden, herstructureren, reorganiseren en onderhouden van data bases. Deze organisaties pleiten nu zelf voor een totaal 'alternatief' data base ontwerp, dat veel dichter aansluit bij de objectieven, in verband met effectief en efficiënt gegevensbeheer.
De ontwerpmethode die wij presenteren kent vier fasen:

- Informatie-analyse.
  Het opsporen en identificeren van relevante informatiebehoeften bij geautoriseerde gebruikers van een te ontwikkelen informatiesysteem.

- Data-analyse.
  Het representeren van de informatiebehoeften door middel van gegevens en gegevensstructuren die zijn samengevat in een stabiel, effectief en automatiserings-onafhankelijk conceptueel gegevensmodel.

- Implementatie-analyse.
  Het documenteren van het conceptueel gegevensmodel met karakteristieken in verband met aard en gebruik van gegevens.

- Fysieke analyse.
  Het opzetten van het gedocumenteerd conceptueel gegevensmodel in een performant fysiek model.

De aanwezigheid van deze vier fasen is eigenlijk logisch wanneer men bedenkt dat binnen een informatiesysteem een aantal operatoren optreden die, gebruik makend van een bepaald middel, een operand wensen te exploiteren.
De operatoren zijn processen van gegevensverwerking en gegevenscommunicatie.

De operand is een gegevensverzameling. Het middel is een combinatie van computerapparatuur en programmatuur op grond waarvan de gegevensverzameling kan worden behandeld. De fasen informatie-analyse en data-analyse staan in verband met het onderkennen van de operand. De fase implementatie-analyse onderkent het noodzakelijk onderzoek naar de wijze waarop een aantal operatoren deze operand willen gebruiken. De fase fysieke analyse onderkent het onderzoek naar de mogelijkheden en beperkingen van apparatuur en programmatuur die inherent zijn aan het gebruikte middel en waarmee eveneens rekening zal moeten worden gehouden bij het omzetten van een conceptueel gegevensmodel.

Belangrijk is ook het terrein dat door de diverse fasen wordt bestreken. Hiermee bedoelen wij het deelgebied van de organisatie waarover de activiteiten binnen een bepaalde fase zich uitstrekken. Het 'bereik' van informatie-analyse is een belangrijk deelgebied van de organisatie. Verderop gaan wij in op de criteria die bij het bepalen van dit deelgebied aan bod komen.
Het 'bereik' van data-analyse is een deelterrein van het terrein waarop de informatie-analyse werd uitgevoerd. Wij gaan straks eveneens in op de criteria die bij het bepalen van dit deelgebied spelen.
Het 'bereik' van implementatie-analyse is opnieuw een deelgebied van het terrein waarop de data-analyse fase is uitgevoerd. Hieromtrent zullen wij eveneens in een later stadium criteria aanbrengen om dit deelgebied te bepalen.
Het 'bereik' van fysieke-analyse is een deelmodel van een gedocumenteerd conceptueel gegevensmodel, te weten een deelgebied van het terrein waarop de implementatie-analyse reeds is uitgevoerd. Door het bepalen van deelgebieden waarop een verdere gedetailleerde studie zal worden uitgevoerd voorkomen wij een te lange wachttijd op een operationeel resultaat. Wij zorgen ervoor dat het data base ontwerp gefaseerd in de tijd operationeel wordt, weliswaar binnen een visie die de volledige compatibiliteit van de oplossingen garandeert en die rekening houdt met terzake toepasselijke prioriteitsbepalingen van het management.

Van hieruit de mening opperen dat de voorgestelde data base ontwerpmethodologie een absolute en volledige top-down benadering inhoudt is nochtans iets te voorbarig. Uiteraard is deze top-down benadering wél mogelijk, maar wij zien het liever als een top-down benadering waarin op bottom-up wijze bepaalde goed functionerende en effectieve delen worden ingepast

(hetgeen vooral belangrijk kan zijn in verband met een mogelijk sneller doorlopen van de fasen 'data-analyse' en 'implementatie-analyse'). Wij denken hier ondermeer aan een mogelijk erg behulpzame inventarisatie in aard en gebruik van gegevens binnen bepaalde projecten die goed zijn gedocumenteerd, waarvan het gegevensgebruik en het gegevensstructuurgebruik zorgvuldig werden bestudeerd, waarvan de programma-logica gestructureerd is opgebouwd, waarvan de naamgeving van gegevens degelijk is overlegd, etc.

3. INFORMATIE-ANALYSE

Het objectief van deze fase kan als volgt worden samengevat: het opsporen en identificeren van relevante informatiebehoeften in verband met een te ontwikkelen informatiesysteem waarop gebruikers zich kunnen beroepen voor het sturen van processen en het beheren van activiteiten waarvoor zij, binnen een bepaalde functie, verantwoordelijkheden dragen. De oplossing van het probleem hoe men reële informatiebehoeften kan achterhalen is moeilijker dan men denkt. Anderzijds is de wijze waarop een antwoord op dit probleem kan worden gegeven bepalend voor een succesvol ontwikkelen van informatiesystemen en data base systemen, daar in dit stadium het contact wordt gelegd tussen gebruiker en ontwerper van het systeem, en men logischerwijze verwacht dat dit systeem zo getrouw mogelijk aan de behoeften van de gebruiker kan voldoen.
In de informatie-analyse fase onderscheiden wij vier stadia:

Subsysteem-ontwerp
Het bepalen van de aard en van de omvang van de deelsystemen waarop informatie-analyse zal worden uitgevoerd, en het aangeven van de volgtijdelijkheid van behandeling van de deelsystemen.

Functie-analyse
Het onderkennen van de belangrijkste functies binnen een deelsysteem en het via functie-analyse opsporen van de processen en/of activiteiten waarover personen binnen een functie verantwoordelijkheid dragen.

Proces/Activiteiten Informatie-analyse
Het onderzoeken van processen en/of activiteiten naar de informatiebehoeften waarop zij zijn gebaseerd, en het ontleden van de informatiebehoeften in informatie-elementen en informatie-structuurelementen.

Informatiemodel-ontwerp

Het synthetiseren van resultaten van deze fase en het grafisch voorstellen van informatie-elementen en informatie-structuurelementen in een subsysteem informatiemodel.

4. DATA-ANALYSE

Het objectief van deze fase kan als volgt worden samengevat: het representeren van de informatiebehoeften door middel van gegevens en gegevensstructuren die worden samengevat in een stabiel, effectief en automatiserings-onafhankelijk conceptueel gegevensmodel. Deze fase wordt uitgevoerd op een gebied waarop eveneens de informatie-analyse is gebeurd, of op een deelgebied hiervan. In het laatste geval omvat dit deelgebied de informatienoden voor één of meer *projecten* die binnen het subsysteem met prioriteit zullen worden verwezenlijkt.
In de data-analyse fase onderscheiden wij opnieuw vier stadia:

Multiproject-ontwerp

Het bepalen van de aard en van de omvang van het deelgebied waarop data-analyse zal worden uitgevoerd, het aangeven van de eventuele volgtijdelijke planning van toepasselijke projecten, en het vooropstellen van een aantal principes die bij het concipiëren van een (multi-)project conceptueel gegevensmodel moeten worden gerespecteerd.

Data-specificatie onderzoek

Het opstellen van standaard gegevensnamen voor een aantal entiteiten en kenmerken van entiteiten en het vastleggen van hun overeengekomen betekenis.

Datastructuur-manipulatie onderzoek

Het vastleggen van een effectief gegevensstructuur-concept op grond waarvan gegevens eenvoudig kunnen worden gemanipuleerd.

Conceptueel gegevensmodel-ontwerp

Het synthetiseren van de resultaten uit deze fase en het grafisch voorstellen van gegevens en gegevensstructuren in een (multi-)project conceptueel gegevensmodel.

De wijze waarop de gegevens binnen het model zijn samengebracht waarborgt een effectieve behandeling van deze gegevens. Het model is onafhankelijk van logische structuurdefinities die binnen een bepaald DBMS kunnen worden gehanteerd, alsook van fysieke beperkingen die bij realisatie kunnen optreden. Deze karakteristieken bezorgen het model een ideale vertrekbasis voor de ontwikkeling van een performant fysiek model. Vooraleer dit gebeurt zullen wij nochtans het conceptueel gegevensmodel dokumenteren met allerlei indicaties omtrent de wijze waarop het door diverse operatoren zal worden aangewend.

5. IMPLEMENTATIE-ANALYSE

Het objectief van deze fase kan als volgt worden samengevat: het documenteren van het conceptueel gegevensmodel met karakteristieken in verband met aard en gebruik van gegevens. Uitgaande van de functies die binnen een project moeten worden ondersteund, alsook van de verzameling van gegevens waarop zij zich hiertoe willen beroepen, onderzoeken wij nu de *wijze* waarop de functies bepaalde gegevens en gegevensstructuren zullen behandelen conform aan de verantwoordelijkheden die zij incorporeren.
In de implementatie-analyse onderscheiden wij drie stadia:

Basis implementatie-analyse
Inventarisatie in aard en gebruik van gegevens en gegevensstructuren door een reeks van processen/activiteiten die samen in een functie voorkomen.

Statische implementatie-analyse
Het bepalen van verwachte data volumes en het onderzoeken van de verdeling van gegevens langsheen bepaalde structuurverbanden (op grond van een samenvatting en een analyse van de verkregen karakteristieken uit de 'Basis implementatie-analyse').

Dynamische implementatie-analyse
Het bepalen van de verwachte frequentie van gebruik van gegevens en het bepalen van de verwachte frequentie van gebruik van access-paden (eveneens op grond van een samenvatting en een analyse van de verkregen karakteristieken uit de 'Basis implementatie-analyse').

6. FYSIEKE ANALYSE

Het objectief van deze fase kan als volgt worden samengevat: het omzetten van (delen van) een gedocumenteerd conceptueel gegevensmodel in een performant data base model. De stap van een gedocumenteerd gegevensmodel naar een data base model is niet zo rechtlijnig als gedacht zou kunnen worden. Meestal zijn er heel wat beperkingen op grond waarvan het data base model in sterke mate kan verschillen van hetgeen conceptueel werd bedacht. Hieruit de conclusie trekken dat men beter onmiddellijk start met het ontwerpen van een data base model is nochtans fout. In deze omstandigheid worden inherente beperkingen niet langer als beperkingen beschouwd en komt men in de bekende situatie waarin de automatisering zich niet aanpast aan het bedrijf maar integendeel het bedrijf wordt aangepast aan de automatisering.
In de fysieke-analyse fase onderscheiden wij drie stadia:

Hardware-onderzoek
Een onderzoek naar de karakteristieken van de apparatuur waarop het gegevensmodel moet worden gerealiseerd.

Software-onderzoek
Een onderzoek naar de karakteristieken van de systeemsoftware en van de data base software door middel waarvan het gegevensmodel kan worden gerealiseerd.

Data base ontwerp
Het omzetten van een gedocumenteerd conceptueel gegevensmodel in een performant data base model waarbij rekening wordt gehouden met de hardware en software mogelijkheden en met de hardware en software beperkingen die terzake toepasselijk zijn.

7. SAMENVATTENDE CONCLUSIES (FIGUUR 2)

- In de 'Informatie-analyse' fase pogen wij relevante informatiebehoeften op te sporen en te identificeren bij geautoriseerde gebruikers van een informatiesysteem. Wij richten ons daartoe naar de '*real world*'-*processen/activiteiten*. Na het uittekenen van de subsystemen en het bepalen van de functies binnen deze subsystemen passen wij een functie-analyse toe op grond waarvan wij inzicht verkrijgen in de processen/activiteiten die in het kader van een functie moeten worden uitgevoerd.

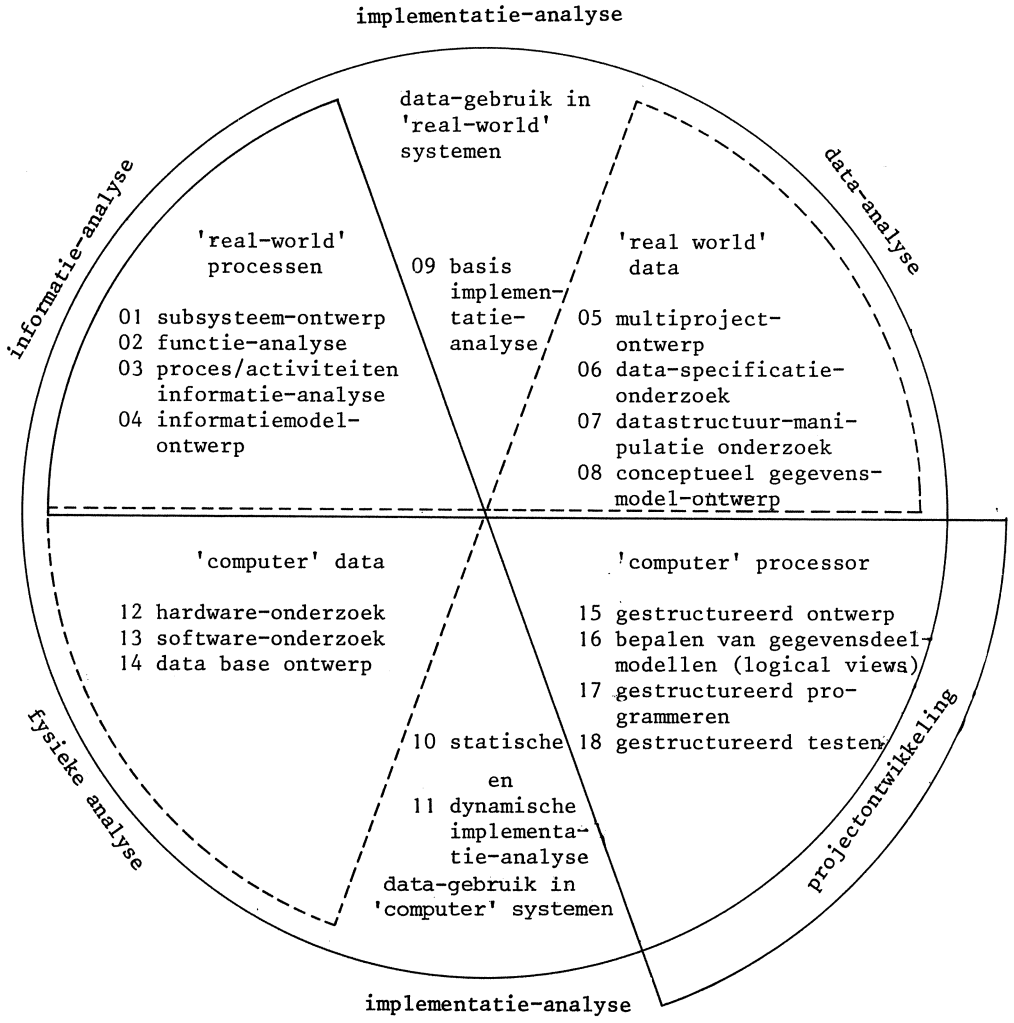
Vervolgens bepalen wij - per activiteit - de essentiële informatie-noden die wij nadien grafisch uitzetten in een informatiemodel dat een weergave vormt van zowel de benodigde informatie-elementen (kenmerken van entiteiten) als van de informatie-structuurelementen (associaties tussen kenmerken van entiteiten).

- In de 'Data-analyse' fase bouwen wij een stabiel, effectief en computer-onafhankelijk conceptueel gegevensmodel. Met deze fase treden wij op het terrein van de *'real world'-data*. Wij bepalen vooraf het deelterrein waarop de data-analyse zal worden uitgevoerd en wij preciseren een aantal vereisten waaraan het te ontwikkelen conceptueel gegevensmodel moet voldoen. Vervolgens stellen wij standaard-gegevensnamen op voor een aantal gegevenseenheden en registreren wij de toepasselijke betekenis van deze eenheden. Hierop ontwikkelen wij een structurele gegevensmodel-vorm op grond van een methodologie die toelaat de bekende objectieven te realiseren. Het resultaat vatten wij samen in een grafisch conceptueel gegevensmodel.

- In de 'Implementatie-analyse' fase willen wij het verkregen conceptueel gegevensmodel documenteren met karakteristieken die in verband staan met aard en gebruik van gegevens en van gegevensstructuren. In een eerste stadium worden per functie een aantal karakteristieken rond gegevensgebruik verzameld. Wij maken hier de *relatie tussen 'real world' processen/activiteiten en 'real world' data*. In de opvolgende stadia worden deze karakteristieken geanalyseerd en worden indicaties verzameld in verband met een effectief en efficiënt *interacteren van 'computer' processen op 'computer' data*.

- In de 'Fysieke analyse' fase wordt een gedocumenteerd conceptueel gegevensmodel (een implementatiemodel) omgezet in een performant data base model. Hier treden wij op het terrein van de *'computer' data*. Bij de omzetting houden wij rekening met kenmerken van apparatuur en van algemene systeemprogrammatuur en data base programmatuur. De eigenlijke omzetting naar een data base model gebeurt volgens een (DBMS-afhankelijke) ontwerp-methodologie. Vanaf het data base model zullen deelmodellen worden ontwikkeld die (conform aan de informatiebehoeften voor een bepaald 'real world' proces), via een *'computer'* proces zullen worden gebruikt ter voortbrenging van de vereiste informatie.

- Het is essentiëel dat de vastgestelde methode van het concipiëren en omzetten van conceptuele gegevensmodellen wordt ingekaderd in de systeemontwikkelingsmethodiek die men in verband met de uitvoering van automatiseringsprojecten binnen een organisatie zal volgen. Dit impliceert vanzelf dat toekomstige generaties van systeemontwikkelingsmethodieken de hiertoe benodigde 'infrastructuur' moeten bieden.



FIGUUR 2. Het ontwerpen, ontwikkelen en omzetten van een conceptueel gegevensmodel. Samenvatting van voorgestelde methodologie.

VRAGEN BEANTWOORDEN ZONDER GEHEIMEN TE ONTHULLEN

W. de JONGE, G.L. SICHERMAN, R.P. van de RIET
Vrije Universiteit, Amsterdam

BEKNOPT OVERZICHT

Vragen beantwoordende systemen moeten veelal bepaalde informatie geheim houden. Dit kan b.v. bereikt worden door soms te weigeren een gestelde vraag te beantwoorden.
In dit artikel worden verschillende criteria ontwikkeld die gebruikt kunnen worden om te beslissen of een vraag beantwoord zal worden. We zullen aantonen welke van deze criteria veilig zijn als de vragensteller geen enkel idee heeft van wat geheim is. Ook zullen we bewijzen dat één van deze criteria zelfs veilig is als de gebruiker van het systeem precies weet wat voor informatie geheim gehouden wordt.

1. INLEIDING

Stel dat een systeem vragen over gegevens in een gegevensbank kan beantwoorden. Een gebruiker wordt verondersteld bepaalde voorkennis te hebben, zoals b.v. aan welke integriteits-eisen de gegevens voldoen. Bepaalde feiten worden bestempeld als geheimen en mogen dus niet aan de gebruiker bekend worden. De gebruiker stelt vragen over de gegevens. Hoe kan het systeem die vragen zoveel mogelijk beantwoorden zonder informatie prijs te geven waaruit de gebruiker een geheim af zou kunnen leiden?

De studie in dit artikel staat in nauw verband met het probleem van het beschermen van persoonsgevoelige gegevens in een gegevensbank (privacy).

2. VERWANT ONDERZOEK

De laatste tijd is er uitgebreid onderzoek gedaan naar het beveiligen van informatie. Het overzichtsartikel van DENNING en DENNING [4] bespreekt vier soorten van beveiliging: access control, flow control, data encryption en inference control. Een ander overzichtsartikel van DENNING [3] behandelt alleen het laatste onderwerp. Inference controls beveiligen een gegevensbank door te voorkomen dat gebruikers geheime informatie af kunnen leiden door het stellen van vragen en het combineren van de antwoorden. De beide genoemde overzichtsartikelen behandelen alleen inference controls in statistische gegevensbanken. In dat geval kan een gebruiker alleen maar statistische vragen stellen en vertrouwelijkheid betekent dan dat ze de waarden van afzonderlijke records niet mogen weten.

Daarentegen omvat de aanpak in dit artikel ook niet-statistische vragen, doordat we bijna elke wel-gevormde formule in de eerste-orde logica toestaan als vraag. Bovendien geeft onze benadering meer mogelijkheden bij het omschrijven van welke informatie geheim is, omdat het al dan niet juist zijn van elk zo'n formule een geheim mag zijn.

Het boek samengesteld door GALLAIRE en MINKER [6] geeft een beschrijving van de wisselwerking tussen logica en gegevensbanken. Dit boek bevat onder andere een artikel van REITER [7]: "Deductive Question-Answering on Relational Data Bases", een stuk van CHANG [2]: "DEDUCE 2: Further Investigations of Deduction in Relational Data Bases" en een verhaal van FUTO, DARVAS en SZEREDI [5]: "The Application of PROLOG to the Development of QA and DBM systems". We noemen in het bijzonder deze artikelen, omdat ze interessant zijn als men te weten wil komen of het mogelijk is om de door ons beschreven systemen te implementeren.

Het meest verwant met ons werk is de studie van BANCILHON en SPYRATOS [1]: "Protection of Information in Relational Data Bases". In de introductie van hun artikel definieren zij een "beveiligings-vraagstuk" d.m.v. de antwoorden op de volgende vragen:

Q1: Wat beveiligen we?
Q2: Tegen wie beveiligen we?
Q3: Hoe beveiligen we?
Q4: Wat betekent "beveiligen"?

Voor onze aanpak luiden de antwoorden:

A1: Informatie waarvan bepaald is dat ze geheim moet blijven.
A2: Tegen een goed geïdentificeerde gebruiker die alleen met het systeem kan communiceren m.b.v. een vraag-taal.
A3: Door soms een vraag weigeren te beantwoorden.
A4: Er voor zorgen dat de gebruiker geen geheime informatie kan afleiden uit de reacties van het systeem.

Onze eerste drie antwoorden zijn hetzelfde als die van Bancilhon en Spyratos, zij het dat zij zich beperken tot relationele gegevensbanken. Ons vierde antwoord verschilt van het hunne in zoverre dat zij proberen te voorkomen dat de door de gebruiker berekende kans dat de geheime informatie juist is, groter wordt. Wij trachten alleen te verhinderen dat de gebruiker met zekerheid af kan leiden dat bepaalde geheime informatie juist is.

3. FUNDAMENTELE AANNAMES

Een veiligheids-systeem is een systeem dat vragen kan beantwoorden, maar ook kan weigeren te antwoorden teneinde te voorkomen dat de gebruiker bepaalde (geheime) informatie te weten komt. Het systeem veronderstelt dat de gebruiker bepaalde informatie heeft en herziet deze aanname na elke beantwoorde vraag. Het systeem beslist zowel of een vraag wel of niet beantwoord wordt, als hoe de aan de gebruiker bekend veronderstelde informatie aangepast wordt, afhankelijk van de inhoud van de gegevensbank, de vragen, de geheimen en de informatie die het veronderstelt dat de gebruiker al heeft. Het systeem is dus deterministisch.

We veronderstellen dat het systeem waarheidlievend is; d.w.z. het zal nooit een onjuist antwoord geven. Bovendien nemen we aan dat de

inhoud van de gegevensbank niet aan verandering onderhevig is. (Dus geen insertions, deletions of updates.) Daardoor blijft door het systeem vrijgegeven informatie altijd toepasselijk.

Om het probleem van verschillende gebruikers die hun kennis bundelen, te voorkomen, veronderstellen we dat er maar een gebruiker is. Meerdere gebruikers kunnen dan beschouwd worden als een enkele gebruiker in verschillende gedaantes. Om te voorkomen dat een gebruiker in verschillende sessies opgedane kennis kan combineren, veronderstellen we dat er maar een sessie is. Meerdere sessies kunnen dan beschouwd worden als opeenvolgende episodes van een enkele sessie.

Eenvoudigheidshalve veronderstellen we verder dat elke vraag bestaat uit een formule; i.e. een wel-gevormde formule in de eerste-orde logica, waarvan alle variabelen gekwantificeerd zijn. We beschouwen derhalve alleen vragen die met "ja" of "nee" beantwoord kunnen worden.

Deze laatstgenoemde beperking is minder streng dan hij misschien wel lijkt. Veronderstel b.v. dat de gegevensbank het feit bevat dat het haar van Mediocrates rood is. Dan kan het systeem de vraag "Wat is de kleur van Mediocrates' haar?" zelf voor intern gebruik omzetten in "Is de kleur van Mediocrates' haar rood?". Vervolgens bepaalt het veiligheids-systeem of deze overeenkomstige vraag zonder gevaar beantwoord kan worden. Is dit het geval dan kan het systeem het antwoord "ja" vrijgeven door naar buiten te antwoorden met "De kleur van Mediocrates' haar is rood;". Als de overeenkomstige vraag niet beantwoord mocht worden, weigert het systeem eenvoudigweg de oorspronkelijke vraag te beantwoorden. Op een vergelijkbare wijze kunnen ook vragen zoals b.v. "Wat zijn de namen van alle gewelddadige mensen?" afgehandeld worden.

4. EEN TOELICHTEND VOORBEELD

Om te laten zien dat het probleem niet zo eenvoudig is als het lijkt, zullen we een systeem in ogenschouw nemen dat alleen een vraag weigert te beantwoorden als het antwoord een geheim zou onthullen. Stel dat de volgende integriteits-eisen van toepassing zijn op de gegevensbank, die het gegeven bevat dat Mediocrates een Athener is:

Iedereen is een Athener, Boeotier, Corinthier of Dorier;
Alle Atheners en Corinthiers zijn geweldloos;
Alle Boeotiers en Doriers zijn gewelddadig;

Mediocrates wil niet dat het bekend wordt dat hij geweldloos is. Rhinologus, een notoire lastpost, probeert echter te weten te komen of Mediocrates geweldloos is of niet:

Rhinologus:	Is Mediocrates een Athener?
Systeem:	Dat vertel ik niet.
Rhinologus:	Is hij een Boeotier?
Systeem:	Nee.
Rhinologus:	Is hij een Corinthier?
Systeem:	Nee.
Rhinologus:	Is hij dan een Dorier?
Systeem:	Dat vertel ik niet.

Op het eerste gezicht lijkt het alsof het systeem voorkomen heeft dat Rhinologus achter Mediocrates' geheim komt. Immers, Mediocrates kan op grond van de antwoorden ofwel een geweldloze Athener zijn, ofwel een gewelddadige Dorier. Rhinologus redeneert echter aldus:

> "Als het juiste antwoord op de eerste vraag "nee" zou zijn, dan zou het systeem die vraag zonder bezwaar hebben kunnen beantwoorden. Immers, dan zou er nog niets onthuld zijn over het al dan niet geweldloos zijn van Mediocrates, maar alleen dat hij geen Athener is. En ik geloof niet dat iemand het feit dat hij geen Athener is, geheim zou willen houden. Derhalve moet het juiste antwoord wel "ja" zijn geweest en dus is Mediocrates een geweldloze Athener."

Hoe komt het dat Rhinologus ondanks de waakzaamheid van het systeem achter het geheim komt? Dat komt doordat Rhinologus niet alleen de integriteits-eisen kent, maar ook weet in wat voor gevallen het systeem weigert en begrijpt wat geheim kan zijn en wat niet. Immers, de eerste vraag zou ook best geweigerd kunnen zijn omdat het systeem de eerste vraag altijd weigert, of omdat Mediocrates niet wil dat men weet dat hij geen Athener is.

5. BESCHRIJVING VAN HET MODEL

Een veiligheids-systeem zal bij ons bestaan uit vier componenten: een gegevensbank, een verzameling geheimen, een hoeveelheid veronderstelde begin-informatie en een twee-waardige functie, censor genaamd, die beslist of een vraag wel of niet beantwoord wordt.

De gegevensbank.

De gegevensbank is niets anders dan een vaste hoeveelheid gegevens. We veronderstellen dat elke formule van de vraag-taal een welgedefinieerde geldigheids-waarde (i.e. juist of onjuist) heeft in de gegevensbank.

De geheimen.

De geheimen zijn formules die juist zijn in de gegevensbank. De gebruiker moet ervan weerhouden worden hun juistheid af te kunnen leiden.

Veronderstelde informatie; de veronderstelde begin-informatie.

We zullen onderscheid maken tussen de informatie van de gebruiker, die bestaat uit alles wat hij weet over de gegevens in de gegevensbank (zoals b.v. integriteits-eisen) en over logische afleidings-regels, en zijn kennis, die behalve uit zijn informatie ook bestaat uit wat hij van het systeem zelf weet (zoals b.v. welke censor het systeem gebruikt). Op elk moment veronderstelt het systeem dat de gebruiker bepaalde informatie heeft, weergegeven door een verzameling formules, de veronderstelde informatie. De veronderstelde begin-informatie is de informatie waarvan het systeem aanneemt dat de gebruiker die heeft voordat de sessie begint, wederom in de vorm van een verzameling formules.

We veronderstellen dat de veronderstelde begin-informatie alleen formules bevat die juist zijn in de gegevensbank van het systeem, d.w.z. dat de gebruiker geen foutieve informatie heeft. Voor onze resultaten is het verder van belang dat de veronderstelde begin-informatie alles bevat wat de gebruiker aanvankelijk weet.

na het geven van de bijbehorende reactie, uit de informatie van de gebruiker dan ofwel het aangegeven geheim, of de aangegeven ontkenning daarvan, afgeleid kan worden. Een cirkel betekent dat noch een geheim, noch een ontkenning daarvan afgeleid kan worden.

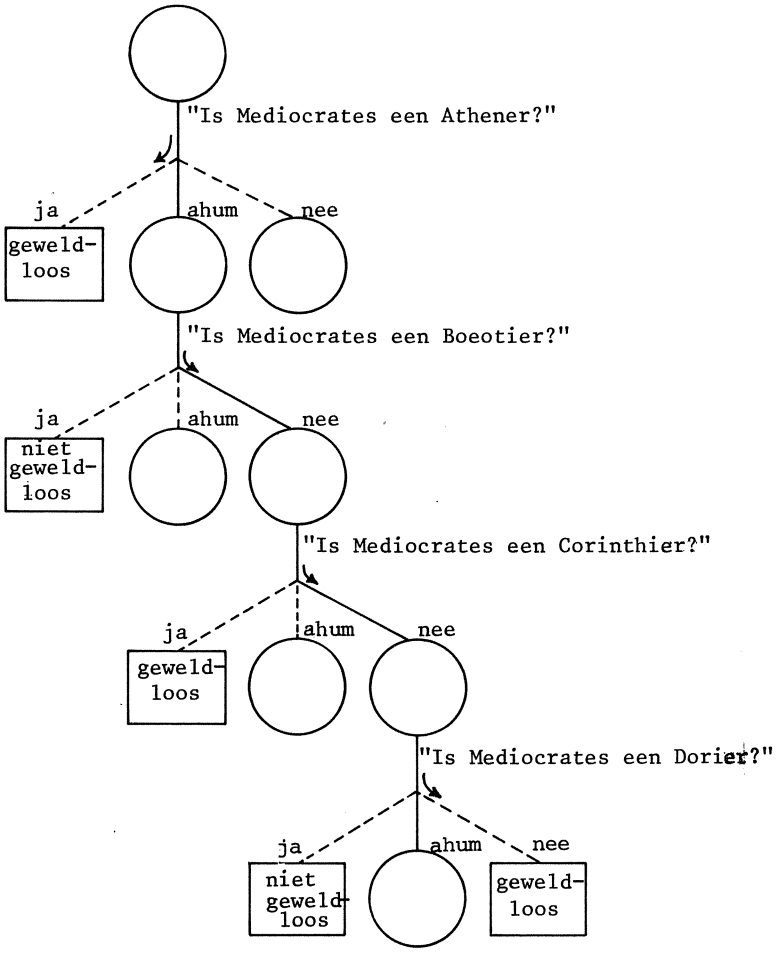


Fig. 1.

Alle systemen die hier beschouwd worden, zullen de veronderstelde informatie overeenkomstig de volgende regels herzien:

> Als het systeem het antwoord op een vraag weigert, dan blijft de veronderstelde informatie ongewijzigd;
> Als het systeem een vraag beantwoord, dan wordt de informatie die daarmee expliciet gegeven wordt, toegevoegd aan de veronderstelde informatie.

Deze veronderstellingen zijn naïef in zoverre dat, zoals we al hebben laten zien, de gebruiker ook gevolgtrekkingen kan maken uit de beslissingen van het systeem om al dan niet te antwoorden. We tonen verderop aan dat onder bepaalde omstandigheden deze regels voor het herzien van de veronderstelde informatie toereikend zijn voor het beschermen van geheimen.

De censor.

De censor is een twee-waardige functie van de gegevensbank, de geheimen, de momenteel veronderstelde informatie en de onderhavige vraag. De waarden die aangenomen kunnen worden zijn "weiger" en "weiger niet". Als de waarde van de censor voor een bepaalde vraag "weiger" is, dan zal het systeem ook daadwerkelijk het antwoord weigeren. Als de waarde "weiger niet" is, zal het systeem het juiste antwoord geven door waarheidsgetrouw "ja" of "nee" te antwoorden. We zullen de term "antwoord" gebruiken voor het juiste antwoord op een vraag en de term "reactie" voor de feitelijke reactie van het systeem op een vraag van de gebruiker. Het systeem kan derhalve op drie manieren reageren op een vraag: met "ja", "nee" of "ik weiger te antwoorden". Dit laatste zullen we afkorten met "ahum".

De reacties van een systeem op een opeenvolging van vragen kunnen weergegeven worden door een boom-structuur waarin ieder knooppunt de situatie weergeeft na alle voorafgaande vragen en elke verbindingslijn een reactie weergeeft. Figuur 1 geeft de boom behorend bij het voorbeeld van hoofdstuk 4. Bij elk knooppunt worden alle potentiele reacties weergegeven. Stippellijnen betekenen mogelijke reacties, een doorgetrokken lijn geeft de feitelijk gegeven reactie weer en een pijl geeft aan wat het juiste antwoord is op de vraag. Een rechthoek betekent dat,

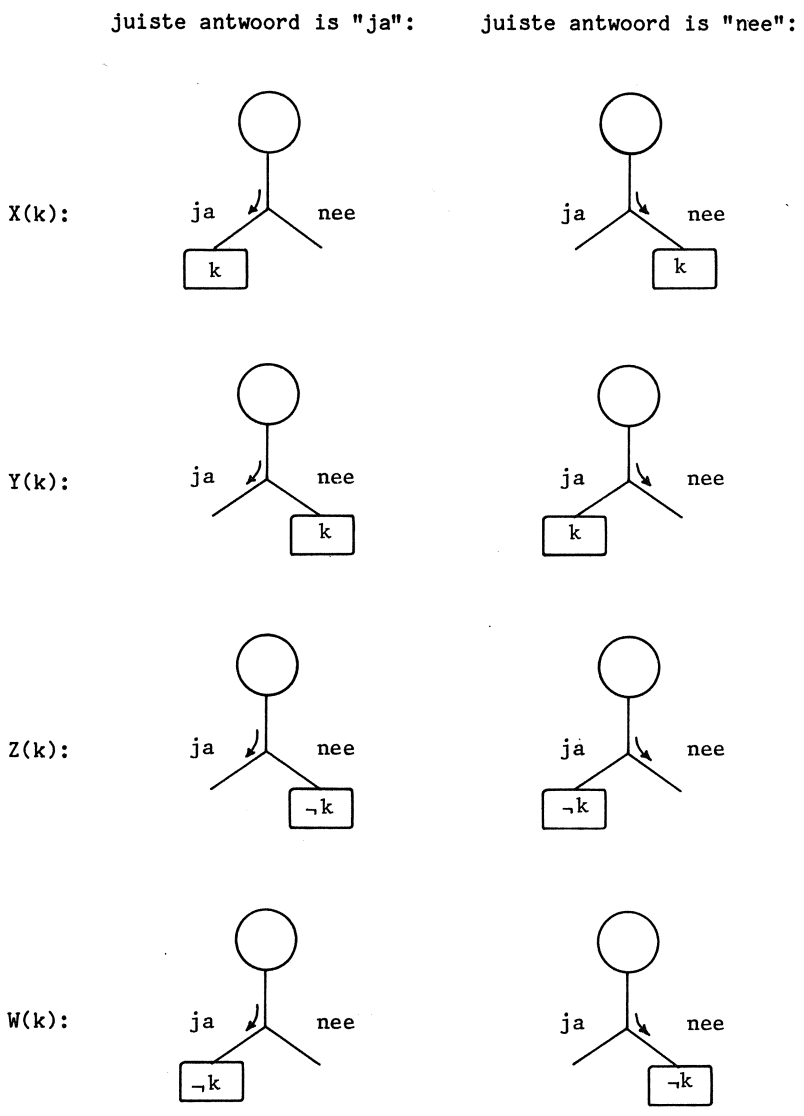

juiste antwoord is "ja":
juiste antwoord is "nee":
X(k):
Y(k):
Z(k):
W(k):
ja
nee
k
¬k

Fig.2.

6. CRITERIA VOOR WEIGEREN

Met betrekking tot een geheim k definieren we vier redenen voor het weigeren van een antwoord:

X(k) - de juistheid van k kan afgeleid worden uit
het juiste antwoord en de veronderstelde informatie samen.
Y(k) - de juistheid van k kan afgeleid worden uit
het onjuiste antwoord en de veronderstelde informatie samen.
Z(k) - de onjuistheid van k kan afgeleid worden uit
het onjuiste antwoord en de veronderstelde informatie samen.
W(k) - de onjuistheid van k kan afgeleid worden uit
het juiste antwoord en de veronderstelde informatie samen.

We zullen $\exists k$: X(k), $\exists k$: Y(k), $\exists k$: Z(k) en $\exists k$: W(k) aangeven met resp. X, Y, Z en W. (zie fig.2.).

Aangezien geheimen formules zijn die juist zijn in de gegevensbank, is W alleen mogelijk als de veronderstelde informatie ook onjuiste informatie bevat. Daarom zullen we er geen aandacht meer aan besteden. Omdat er tegelijkertijd meerdere redenen kunnen zijn om te weigeren, hebben we dus zeven gevallen waarin X(k), Y(k), of Z(k) voorkomen. We korten ze als volgt af (zie fig.3.):

t: voor een of ander geheim k
zijn X(k), Y(k) en Z(k) alle van toepassing;
u: voor een of ander geheim k
zijn X(k) en Y(k) van toepassing, maar Z(k) niet;
v: voor een of ander geheim k
zijn X(k) en Z(k) van toepassing, maar Y(k) niet;
w: voor een of ander geheim k
zijn Y(k) en Z(k) van toepassing, maar X(k) niet;
x: voor een of ander geheim k
is X(k) van toepassing, maar Y(k) en Z(k) niet;
y: voor een of ander geheim k
is Y(k) van toepassing, maar X(k) en Z(k) niet;
z: voor een of ander geheim k
is Z(k) van toepassing, maar X(k) en Y(k) niet.

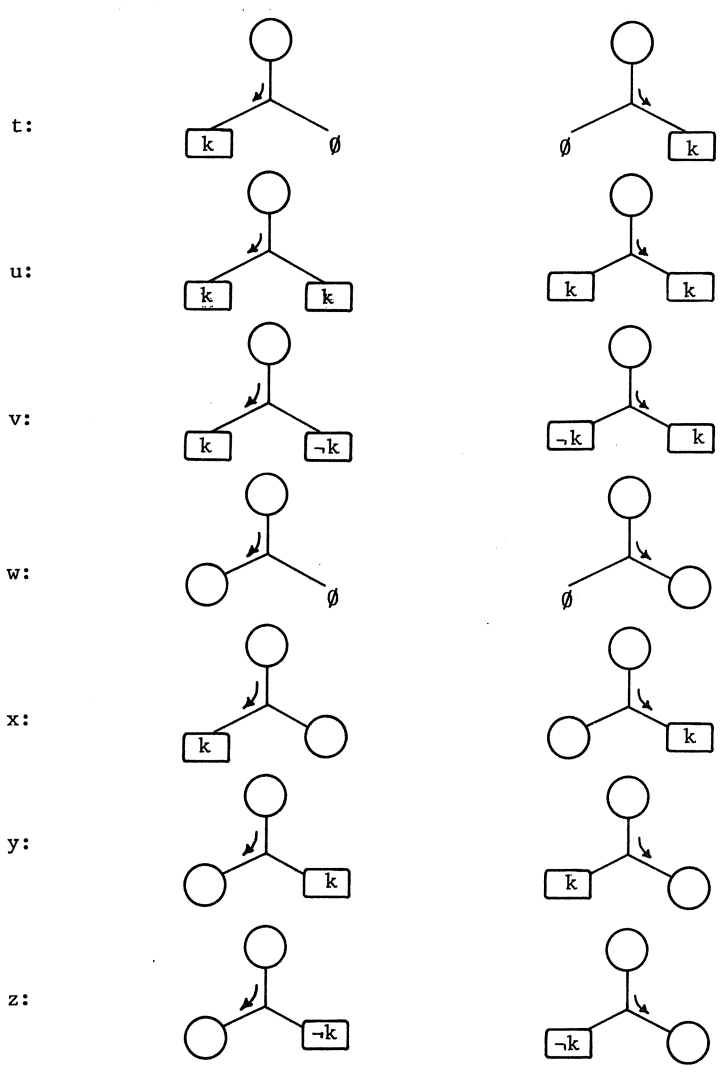


Fig.3.

Een veiligheids-systeem zou ook kunnen weigeren te antwoorden als voor een of ander geheim k noch X(k), noch Y(k), noch Z(k) van toepassing is. Maar dan impliceert noch het juiste, noch het onjuiste antwoord de juistheid of onjuistheid van dat geheim en we zullen dit geval dan ook niet verder behandelen.

Gevallen t, u en w zijn oneigenlijk. In gevallen t en u kon de juistheid van het geheim al afgeleid worden uit de veronderstelde informatie. In gevallen t en w is het onjuiste antwoord onmogelijk, omdat het tot een tegenspraak zou leiden; oftewel, het juiste antwoord kon al afgeleid worden uit de veronderstelde informatie. Daarom kan het systeem in deze gevallen net zo goed antwoorden teneinde te bereiken dat vragen zoveel mogelijk beantwoord worden.

Elke combinatie van de overgebleven vier gevallen kan gebruikt worden als criterium voor het weigeren van het antwoord; een censor kan b.v. het antwoord laten weigeren in gevallen v en y, maar het laten geven in gevallen x en z.
(We beschouwen deze combinaties alleen als voorschriften om te weigeren en niet als voorschriften om te antwoorden. Als we dat laatste wel zouden doen, betekent dat, dat als er geen geheimen zijn, geen enkel van de gevallen zich voor kan doen en dus dat het systeem nooit zou antwoorden. Dat vinden we absurd.)
Kortom, er zijn zestien censoren mogelijk, overeenkomstig de volgende tabel. Het teken ¦ wordt gebruikt voor het scheiden van alternatieven; zo betekent "v ¦ z" dat er geweigerd wordt als v of z (of beiden) zich voordoen. Let wel, meerdere gevallen kunnen zich alleen tegelijkertijd voordoen m.b.t. verschillende geheimen!

CENSOR	CRITERIUM VOOR WEIGERING
c1	v ¦ x ¦ y ¦ z
c2	v ¦ x ¦ y
c3	v ¦ x ¦ z
c4	v ¦ x
c5	v ¦ y ¦ z
c6	v ¦ y
c7	v ¦ z
c8	v
c9	x ¦ y ¦ z
c10	x ¦ y
c11	x ¦ z
c12	x
c13	y ¦ z
c14	y
c15	z
c16	nooit

Ter toelichting beschouwen we het voorbeeld van hoofdstuk 4. De vragen luidden:

(1) Is Mediocrates een Athener?
(2) Is Mediocrates een Boeotier?
(3) Is Mediocrates een Corinthier?
(4) Is Mediocrates een Dorier?

De systemen die gebruik maken van resp. c1 tot c16, reageren als aangegeven in de volgende tabel:

	c1	c2	c3	c4	c5	c6	c7	c8
(1)	ahum	ahum	ahum	ahum	ja	ja	ja	ja
(2)	ahum	nee	ahum	nee	*	*	*	*
(3)	ahum	ahum	nee	nee	*	*	*	*
(4)	ahum	ahum	ahum	ahum	*	*	*	*

	c9	c10	c11	c12	c13	c14	c15	c16
(1)	ahum	ahum	ahum	ahum	ja	ja	ja	ja
(2)	ahum	nee	ahum	nee	*	*	*	*
(3)	ahum	ahum	nee	nee	*	*	*	*
(4)	ahum	nee	ahum	nee	*	*	*	*

Hierbij betekent een sterretje dat het geheim al onthuld is.

Om toe te lichten hoe de tabel is verkregen, geven we de berekening voor c10. De eerste vraag luidt: "Is Mediocrates een Athener?". Het juiste antwoord ("ja") impliceert het geheim en het onjuiste antwoord ("nee") niet. Dit is geval x en derhalve laat c10 het antwoord achterhouden.

De tweede vraag luidt: "Is Mediocrates een Boeotier?". Het juiste antwoord ("nee") impliceert het geheim niet en het onjuiste antwoord ("ja") zou de ontkenning van het geheim impliceren. Dit is dus geval z en c10 geeft daarom toestemming om te antwoorden.

Voor de derde vraag: "Is Mediocrates een Corinthier?", geldt dat het juiste antwoord ("nee") niets impliceert en dat het onjuiste antwoord ("ja") de juistheid van het geheim zou onthullen. Dit is geval y en dus laat c10 niet antwoorden.

De vierde vraag luidt: "Is Mediocrates een Dorier?". Het juiste antwoord ("nee") onthult samen met het antwoord "nee" op de tweede vraag het geheim, en het onjuiste antwoord ("ja") zou de onjuistheid van een geheim betekenen. Dit is geval v en c10 laat dus antwoorden.

Censoren c5 t/m c16 zijn alle ongeschikt voor een veiligheids-

systeem, omdat ofwel geval x, ofwel geval v ontbreekt. Voor het gegeven voorbeeld laat de tabel zien dat de censoren die x missen, vraag (1) beantwoorden, waarmee ze het geheim prijsgeven, en dat c10 en c12, die v missen, het geheim onthullen door vraag (4) te beantwoorden. In het gegeven voorbeeld geven c9 en c11 het geheim niet prijs, maar het is eenvoudig om een voorbeeld te maken waarin dat wel gebeurt. Zo zouden beiden b.v. antwoorden op de vraag "Is Mediocrates geweldloos?".

7. VERDERE AANNAMES

We zullen een geheim als veilig verborgen beschouwen als er nog een ander veiligheids-systeem bestaat dat overeenstemt met de kennis van de gebruiker en waarin dat geheim niet juist is. De rechtvaardiging voor deze definitie is duidelijk. Als zo'n alternatief systeem bestaat, dan kan de gebruiker het niet onderscheiden van het echte systeem, d.w.z. niet afleiden welk van beide systemen tot nu toe op zijn vragen gereageerd heeft. Dus weet hij ook niet of het geheim juist is of niet.

In het algemeen kan de gebruiker bij het afleiden zowel gebruik maken van zijn kennis van het veiligheids-systeem als van zijn informatie betreffende de gegevens. Hij kent in ieder geval de reacties van het systeem. Daarnaast kan een gebruiker weten welke censor er gebruikt wordt, wat de veronderstelde begin-informatie is, of van welke formules het juist of onjuist zijn geheim is; d.w.z. hij kan b.v. weten dat het wel of niet geweldloos zijn van Mediocrates geheim is.

Als k een geheim is, noemen we de uitdrukking "wel of niet k" een geheimenis. We hadden net zo goed af kunnen spreken dat een systeem i.p.v. een verzameling geheimen een verzameling geheimenissen heeft. Immers de inhoud van de gegevensbank en de geheimenissen bepalen samen op eenduidige wijze de geheimen.

I.h.a. zal de gebruiker de inhoud van de gegevensbank niet kennen, omdat er anders zeker niets voor hem verborgen kan blijven. Verder nemen we aan dat de veronderstellingen van het systeem omtrent de begin-informatie van de gebruiker juist zijn en dat de gebruiker weet van welke censor het systeem gebruik maakt.

8. WAT BEWEZEN ZAL WORDEN

Censor c_4, die alleen de meest voor de hand liggende reden X gebruikt, lijkt de aangewezen keus voor gebruik als censor. Maar, zoals het voorbeeld in hoofdstuk 4 al liet zien, c_4 voorkomt niet altijd dat de gebruiker een geheim te weten kan komen.

Een van onze twee stellingen beweert dat c_4 goed genoeg is om geheimen voor de gebruiker verborgen te houden, tenminste als die gebruiker niets weet over wat geheim is. Hij mag dus zeker geen geheimenissen kennen. In feite zijn onder deze voorwaarden c_1, c_2 en c_3 ook veilig, maar c_4 is beter in zoverre dat hij meer antwoorden toestaat.

De andere stelling beweert dat c_1 zelfs goed genoeg is om de geheimen verborgen te houden als de gebruiker de geheimenissen wel kent. Bijvoorbeeld, als de gebruiker niet weet of Mediocrates geweldloos is, maar wel weet dat het al dan niet geweldloos zijn van Mediocrates voor hem geheim gehouden wordt. We hebben al laten zien dat c_4 onder deze omstandigheden niet veilig is. In feite zijn c_2 en c_3 ook onveilig, zodat c_1 de enige censor van de zestien is, die veilig is als de gebruiker de geheimenissen kent.

Onze bewijzen van beide stellingen berusten op de volledigheidsstelling van K. Gödel (1930).

9. DEFINITIES EN NOTATIE

Als q en q' formules zijn, dan schrijven we $q \Rightarrow q'$ als q' logisch volgt uit q, en $q \Leftrightarrow q'$ als q en q' uit elkaar afleidbaar zijn. Als Γ een verzameling formules is en q is een formule, dan schrijven we $\Gamma \Rightarrow q$ als q logisch volgt uit de formules in Γ, en anders $\Gamma \not\Rightarrow q$. Als q een formule is en Γ is een verzameling formules, dan schrijven we $q \wedge \Gamma$ voor de logische "en" van q en alle formules in Γ; in het bijzonder $q \wedge \emptyset = q$. Als $\mathcal{U}$ een gegevensbank is en q een formule, dan schrijven we $\mathcal{U} \models q$ als q juist is in $\mathcal{U}$, en anders $\mathcal{U} \models \neg q$. Een <u>geheim</u> k voor een bepaalde gegevensbank $\mathcal{U}$ is een formule die juist is in $\mathcal{U}$. Een <u>geheimenis</u> $\hat{k}$ is een ongeordend paar formules $\{k, \neg k\}$, waarbij k een geheim is. Voor een verzameling geheimen K noteren we de

bijbehorende verzameling geheimenissen $\{\hat{k} \mid k \in K\}$ met $\hat{K}$. De waarde van een geheimenis $\hat{k}$ in een willekeurige gegevensbank W wordt aangegeven met $\hat{k}/W$ en wordt gedefinieerd door:

$$\hat{k}/W = \begin{cases} k & \text{als } W \models k \\ \neg k & \text{als } W \models \neg k. \end{cases} \qquad (1)$$

Op een gelijksoortige wijze definieren we de waarde $\hat{q}/W$ van een willekeurige formule q. Dus $\hat{q}/W$ staat voor het juiste antwoord op vraag q in W. De uitdrukking "$\neg\hat{q}/W$" wordt geïnterpreteerd als "$\neg(\hat{q}/W)$". De alternatieve interpretatie "$(\neg\hat{q})/W$" is zinloos, omdat per definitie $(\neg\hat{q})=\hat{q}$.

Een censor is een functie c(W,K,A,q) die de waarden "true" (weiger te antwoorden) of "false" (antwoord) aan kan nemen. Hierbij staat K voor een verzameling geheimen en A voor de aan de gebruiker bekend veronderstelde informatie.

Een veiligheids-systeem is een kwadrupel $S=[W,K,A_o,c]$, waarbij W een gegevensbank is, K een verzameling geheimen voor W, A_o de veronderstelde begin-informatie en c een censor. Gegeven een reeks van vragen $q_1,\ldots,q_n$, dan definieert S een reeks reacties $r_1^S,\ldots,r_n^S$ en de aan de gebruiker na elke vraag q_i bekend veronderstelde informatie A_i^S d.m.v. de recursie:

$$r_i^S = \begin{cases} \text{ja}, & \text{als } W \models q_i \wedge \neg c(W, K, A_{i-1}^S, q_i) \\ \text{nee}, & \text{als } W \models \neg q_i \wedge \neg c(W, K, A_{i-1}^S, q_i) \\ \text{ahum}, & \text{als } c(W, K, A_{i-1}^S, q_i) \end{cases} \qquad (1 \le i \le n) \qquad (2)$$

$$A_o^S = A_o \qquad (3)$$

$$A_i^S = \begin{cases} A_{i-1}^S & \text{als } c(W,K,A_{i-1}^S,q_i) \\ A_{i-1}^S \cup \{\hat{q}_i/W\} & \text{als } \neg c(W,K,A_{i-1}^S,q_i). \end{cases} \qquad (1 \le i \le n) \qquad (4)$$

Nu kunnen we reden X(k) van hoofdstuk 6 uitdrukken als:

$$A \cup \{\hat{q}/W\} \Rightarrow k \qquad (5)$$

en het nauwkeuriger noteren met $X(k,A,\hat{q}/\mathcal{W})$. We zullen $X(k)$ blijven gebruiken waar dat ondubbelzinnig is. Evenzo schrijven we $Y(k,A,\hat{q}/\mathcal{W})$ voor

$$A \cup \{\neg\hat{q}/\mathcal{W}\} \Rightarrow k \tag{6}$$

en $Z(k,A,\hat{q}/\mathcal{W})$ voor

$$A \cup \{\neg\hat{q}/\mathcal{W}\} \Rightarrow \neg k. \tag{7}$$

Bovendien kunnen we criteria c_1 en c_4 nu uitdrukken als

$$c_1(\mathcal{W},K,A,q) \iff \exists k \in K\colon X(k,A,\hat{q}/\mathcal{W}) \vee Y(k,A,\hat{q}/\mathcal{W}) \vee Z(k,A,\hat{q}/\mathcal{W}) \tag{8}$$

en

$$c_4(\mathcal{W},K,A,q) \iff \exists k \in K\colon X(k,A,\hat{q}/\mathcal{W}). \tag{9}$$

10. NIET ANTWOORDEN KAN VEILIG ZIJN

In het volgende lemma tonen we aan dat als de juistheid van een geheim niet afgeleid kan worden uit de veronderstelde begin-informatie, deze dan ook niet afgeleid kan worden uit de later aan de gebruiker bekend veronderstelde informatie, tenminste als de censor ervoor zorgt dat er geweigerd wordt als reden X geldt; d.w.z. in gevallen t, u, v en x. Aangezien t en u oneigenlijk zijn, is het voldoende dat er geweigerd wordt in gevallen v en x.

Lemma 10.1

Stel $S=[\mathcal{W},K,A_o,c]$ is een veiligheids-systeem, waarbij censor c het antwoord laat weigeren als reden X geldt. Laat $q_1,\ldots,q_n$ de vragen zijn. Stel dat k_o een element is van K. Als $A_o \not\vdash k_o$, dan geldt $A_i^S \not\vdash k_o$ voor alle $0 \leq i \leq n$.

Bewijs van lemma 10.1.

We tonen m.b.v. van inductie naar m aan dat

$$A_m^S \not\vdash k_o \quad (0 \leq m \leq n). \tag{10}$$

Voor m=0 is dat juist volgens (3). Stel dat k_o niet afgeleid kan worden na m-1 vragen; d.w.z.

$$A_{m-1}^S \not\vdash k_o. \tag{11}$$

Geval I: Stel dat c het antwoord op q_m weigert. Dan $A_m^S = A_{m-1}^S$ en het gewenste resultaat volgt uit de inductie-aanname.

Geval II: Stel dat c toestaat dat q_m beantwoord wordt. Dan geldt $\forall k \in K: \neg X(k)$ en dus treedt $X(k_o)$ zeker niet op; d.w.z.

$$A_{m-1}^S \cup \{\hat{q}_m/W\} \not\vdash k_o. \tag{12}$$

Verder geldt volgens (4)

$$A_m^S = A_{m-1}^S \cup \{\hat{q}_m/W\} \tag{13}$$

en derhalve $A_m^S \not\vdash k_o$. □

De bewering in Lemma 10.1 garandeert niet dat de geheimen veilig zijn. Er wordt alleen beweerd dat ze niet afgeleid kunnen worden uit de veronderstelde informatie zoals die door het systeem bijgehouden wordt. De beweringen in de komende twee stellingen zijn sterker: zij zeggen dat de gebruiker zelfs geen geheimen af kan leiden als hij ook de beslissingen van het systeem om wel of niet te antwoorden in zijn beschouwingen betrekt.

Onze eerste stelling zegt dat c_1 voldoende is om te voorkomen dat de gebruiker achter de juistheid van geheimen kan komen, zelfs als hij de geheimenissen kent ($=\hat{K}$), de informatie heeft die hij wordt verondersteld te hebben ($=A_o$) en weet dat het systeem censor c_1 gebruikt.

Stelling 10.2.

Stel $S=[W,K,A_o,c_1]$ is een veiligheids-systeem met c_1 gedefinieerd als bovenvermeld. Laat $q_1,\ldots,q_n$ de vragen zijn. Voor elke $k \in K$ waarvoor geldt dat $A_o \not\vdash k$, bestaat er een veiligheids-systeem

$T=[\mathcal{W}^*,K^*,A_o,c_1]$, met

$$\hat{K}^* = \hat{K}, \tag{14}$$

zodanig dat

$$r_i^T = r_i^S \quad (1\leq i\leq n) \tag{15}$$

en

$$\mathcal{W}^* \models \neg k. \tag{16}$$

Bewijs van stelling 10.2.

Stel dat k_o een geheim van K is zodanig dat $A_o \not\models k_o$. Volgens Lemma 10.1,

$$A_n^S \not\models k_o. \tag{17}$$

Dan bestaat er volgens Gödel's volledigheids-stelling een database $\mathcal{W}^*$ waarin de formules van A_n^S juist zijn en k_o niet; d.w.z.

$$\forall q \in A_n^S: (\mathcal{W}^* \models q) \tag{18}$$

en

$$\mathcal{W}^* \models \neg k_o. \tag{19}$$

Neem $K^*=\{\hat{k}/\mathcal{W}^* \mid k \in K\}$ en $T=[\mathcal{W}^*,K^*,A_o,c_1]$. Dan bevat K^* geheimen van dezelfde vorm als die van K, d.w.z.

$$\forall k^* \in K^*: \exists k \in K \quad \text{zodanig dat } k^*=k \text{ of } k^*=\neg k \tag{20}$$

$$\text{en } \forall k \in K : \exists k^* \in K^* \text{ zodanig dat } k^*=k \text{ of } k^*=\neg k \tag{21}$$

oftewel: $\hat{K}^* = \hat{K}$.

We kunnen niet K zelf voor T gebruiken, omdat per definitie geheimen juist moeten zijn in de betreffende gegevensbank en sommige geheimen van K (in ieder geval k_o) niet waar zijn in $\mathcal{U}^*$. Door K* als bovenvermeld te definieren, hebben we precies die geheimen waarvoor dat nodig is, van een ontkenning voorzien.

We bewijzen nu d.m.v. inductie dat T precies dezelfde reacties zou hebben gegeven als S.
Stel dat voor een zekere m ($1 \leq m \leq n$):

$$r_i^T = r_i^S \qquad (1 \leq i < m). \tag{22}$$

Dan is de gebruiker's veronderstelde informatie voor vraag q_m ook hetzelfde in beide systemen; dus,

$$A_{m-1}^T = A_{m-1}^S, \tag{23}$$

en we kunnen dus voor beide A_{m-1} schrijven.

We gaan bewijzen dat T weigert om q_m te beantwoorden dan en slechts dan als S dat ook weigert. Dit is voldoende omdat volgens (4) en (18) de antwoorden die door S gegeven zijn, ook alle in T gelden, en als dus T besluit om te antwoorden, kan T alleen hetzelfde antwoord geven als S.

Stel dat k een geheim van K is en dat $k^* = \hat{k}/\mathcal{U}^* \in K^*$.

Geval A.

Stel dat $k^* = k$ en $\hat{q}_m/\mathcal{U}^* = \hat{q}_m/\mathcal{U}$.

Dan:

$$X(k, A_{m-1}, \hat{q}_m/\mathcal{U}) \Longleftrightarrow X(k^*, A_{m-1}, \hat{q}_m/\mathcal{U}^*)$$

$$\text{en } Y(k, A_{m-1}, \hat{q}_m/\mathcal{U}) \Longleftrightarrow Y(k^*, A_{m-1}, \hat{q}_m/\mathcal{U}^*)$$

$$\text{en } Z(k, A_{m-1}, \hat{q}_m/\mathcal{U}) \Longleftrightarrow Z(k^*, A_{m-1}, \hat{q}_m/\mathcal{U}^*)$$

naar het juiste antwoord kijkt en waarvan de beslissingen dus afhangen van wat er in de betreffende gegevensbank staat.

V: Stel dat het feit dat Mediocrates rood haar heeft, een geheim is. Dan zal een vraag zoals "Heeft Mediocrates zwart haar?" niet beantwoord worden door een systeem dat c_1 gebruikt, ook al denkt men misschien dat antwoorden met "nee" veilig is, omdat Mediocrates dan nog best bruin of blond haar zou kunnen hebben. Dus een systeem gebaseerd op een (b.v. relationele) gegevensbank en gebruik makend van c_1 zal het antwoord weigeren op elke vraag over de waarde in een record (tupel) als die waarde geheim moet blijven. Dat een systeem dat c_1 gebruikt, weinig mededeelzaam kan zijn, kan men ook zien aan de eerste kolom van de tabel in hoofdstuk 6.

De volgende stelling zegt dat als het systeem c_4 gebruikt en als de gebruiker niets over de geheimen weet, hij dan de juistheid van geen enkel geheim af kan leiden ook al beschikt hij over de veronderstelde begin-informatie A_0 en weet hij dat c_4 de gebruikte censor is.

Stelling 10.3.

Stel $S=[W,K,A_0,c_4]$ is een veiligheids-systeem, met c_4 gedefinieerd als bovenvermeld. Laat $q_1,\ldots,q_n$ de vragen zijn. Voor elke $k \in K$ waarvoor geldt dat $A_0 \not\vdash k$, bestaat er een veiligheids-systeem $T=[W^*,K^*,A_0,c_4]$ zodanig dat

$$r_i^T = r_i^S \quad (1 \leq i \leq n) \tag{24}$$

en

$$W^* \models \neg k. \tag{25}$$

Bewijs van stelling 10.3.

Stel dat k_0 een geheim van K is waarvoor geldt dat $A_0 \not\vdash k_0$. Volgens Lemma 10.1 geldt dan

$$A_n^S \not\vdash k_0. \tag{26}$$

Geval B.

Stel dat $k^* = k$ en $\hat{q}_m/W^* = \neg\hat{q}_m/W$.

Dan:

$$X(k,A_{m-1},\hat{q}_m/W) \Longleftrightarrow Y(k^*,A_{m-1},\hat{q}_m/W^*)$$

$$\text{en } Y(k,A_{m-1},\hat{q}_m/W) \Longleftrightarrow X(k^*,A_{m-1},\hat{q}_m/W^*)$$

Z kan zich in dit geval in geen van beide systemen voordoen. Immers, stel dat $Z(k,A_{m-1},\hat{q}_m/W)$ wel geldt. Dan:

$$A_{m-1} \cup \{\neg\hat{q}_m/W\} \Rightarrow \neg k$$

en dus:

$$A_{m-1} \cup \{\hat{q}_m/W^*\} \Rightarrow \neg k^*.$$

Omdat $\hat{q}_m/W^*$ en alle formules in A_{m-1} juist zijn in W^*, betekent dit dat:

$$\neg k^* \text{ juist is in } W^*,$$

wat een tegenspraak oplevert, omdat ook k^* (per definitie) juist is in W^*. Evenzo kan $Z(k^*,A_{m-1},\hat{q}_m/W^*)$ niet waar zijn.

Geval C.

Stel dat $k^* = \neg k$ en $\hat{q}_m/W^* = \hat{q}_m/W$.

Dan:

$$Y(k,A_{m-1},\hat{q}_m/W) \Longleftrightarrow Z(k^*,A_{m-1},\hat{q}_m/W^*)$$

$$\text{en } Z(k,A_{m-1},\hat{q}_m/W) \Longleftrightarrow Y(k^*,A_{m-1},\hat{q}_m/W^*)$$

In dit geval kan X zich in geen van beide systemen voordoen, omdat dat zou betekenen dat zowel een geheim als de ontkenning daarvan juist zijn in een en dezelfde gegevensbank.

Geval D.

Stel dat $k^* = \neg k$ en $\hat{q}_m/W^* = \neg\hat{q}_m/W$.

Dan:

$$X(k, A_{m-1}, \hat{q}_m/W) \iff Z(k^*, A_{m-1}, \hat{q}_m/W^*)$$

$$\text{en } Z(k, A_{m-1}, \hat{q}_m/W) \iff X(k^*, A_{m-1}, \hat{q}_m/W^*).$$

In dit geval kan Y zich in geen van beide systemen voordoen.

Uit alle gevallen tesamen volgt dat als S weigert om q_m te beantwoorden, T dat ook zal doen, en omgekeerd. □

Opmerkingen:

I: Als de gebruiker de juistheid van een geheim al kent voordat de sessie begint, is het natuurlijk onmogelijk om dat geheim voor hem verborgen te houden. Vandaar dat alleen beweerd wordt dat geheimen $k \in K$ waarvoor geldt dat $A_0 \not\vdash k$, veilig verborgen blijven.

II: Voorwaarde (14) is nodig, omdat we verondersteld hebben dat de gebruiker de geheimenissen kent.

III: De gedachte achter de gevallen A t/m D is als volgt:

Censor c_1 laat q_m alleen beantwoorden als noch X, noch Y, noch Z zich voordoet, d.w.z. alleen als m.b.t. alle geheimen de situatie is als aangegeven in fig.4. Derhalve betekent een weigering van S om q_m te beantwoorden, dat de situatie zodanig is dat voor een of ander geheim $k \in K$ tenminste een van de takken uitmondt in een rechthoek gemerkt met k of $\neg k$. (Zie de niet oneigenlijke gevallen v, x, y en z in fig.3.). Maar m.b.v. (21) volgt hieruit dat de situatie dan ook zodanig is, dat T zal weigeren te antwoorden. Bijvoorbeeld, als q_m juist is in W en als S weigert te antwoorden omdat geval z zich voordoet (zie fig.5.a.), dan zal, verondersteld dat k en q_m niet juist zijn in W*, ook T weigeren te antwoorden omdat voor T zich dan geval x voordoet (zie fig.5.b.).

juiste antwoord is "ja": juiste antwoord is "nee":

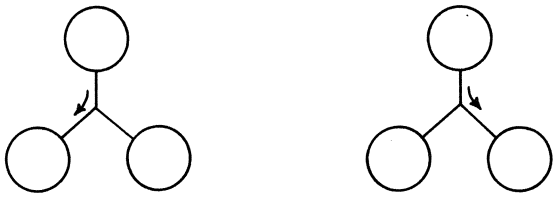
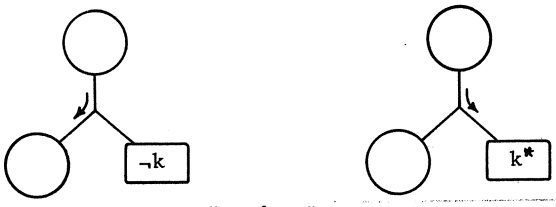


Fig.4.

juiste antwoord m.b.t. $\mathcal{W}$ is "ja": juiste antwoord m.b.t. $\mathcal{W}^*$ is "nee":



a. $k^* = \hat{k}/\mathcal{W}^* = \neg k$ b.

$\hat{q}_m/\mathcal{W} = q_m$

$\hat{q}_m/\mathcal{W}^* = \neg q_m$

Fig.5.

IV: Een essentiële eigenschap van c_1 is dat hij laat weigeren te antwoorden zowel als het juiste antwoord als wanneer het onjuiste antwoord een geheim of de ontkenning ervan zou onthullen. M.a.w., bij het beslissen of een vraag beantwoord mag worden, kijkt c_1 niet naar wat het juiste antwoord is. Derhalve zijn de beslissingen van c_1 niet afhankelijk van de gegevensbank die het systeem feitelijk tot zijn beschikking heeft. Dit in tegenstelling to c_4, die alleen

Volgens Gödel's volledigheids-stelling bestaat er dan een database $\mathcal{W}^*$ waarin de formules van A_n^S juist zijn en k_o niet; d.w.z.

$$\forall q \in A_n^S\colon (\mathcal{W}^* \models q) \tag{27}$$

en

$$\mathcal{W}^* \models \neg k_o. \tag{28}$$

De gevallen waarbij c_4 het antwoord achterhoudt, zijn juist die waarbij reden X van toepassing is. Definieer een nieuwe verzameling geheimen

$$K^* = \{\ (\hat{q}_i/\mathcal{W}^*) \wedge A_{i-1}^S \ \mid\ \exists k \in K\colon X(k, A_{i-1}^S, \hat{q}_i/\mathcal{W})\ \ (1 \leq i \leq n)\ \}. \tag{29}$$

Elk geheim in K^* bestaat dus uit het antwoord in $\mathcal{W}^*$ op een door S geweigerde vraag, samen met de aan de gebruiker bekend veronderstelde informatie in S juist voordat die vraag gesteld werd.

Neem $T = [\mathcal{W}^*, K^*, A_o, c_4]$. We gaan nu d.m.v. inductie bewijzen dat T precies dezelfde reacties zou hebben gegeven als S. Stel dat voor een zekere m $(1 \leq m \leq n)$:

$$r_i^T = r_i^S \qquad (1 \leq i < m). \tag{30}$$

Dan is de veronderstelde informatie voor vraag q_m ook hetzelfde in beide systemen; d.w.z.

$$A_{m-1}^T = A_{m-1}^S, \tag{31}$$

en dus kunnen we beiden aanduiden met A_{m-1}.

Geval I.

Stel dat S vraag q_m weigerde; d.w.z.

$$\exists k \in K\colon X(k, A_{m-1}, \hat{q}_m/\mathcal{W}). \tag{32}$$

Neem $k^* = \hat{q}_m/W^* \wedge A_{m-1}$. Dan geldt volgens (29) dat: $k^* \in K^*$.

Dus: $\exists k^* \in K^*: A_{m-1} \cup \{\hat{q}_m/W^*\} \Rightarrow k^*$ (33)

oftewel: $\exists k^* \in K^*: X(k^*, A_{m-1}, \hat{q}_m/W^*)$, (34)

en dus weigert ook T vraag q_m te beantwoorden.

Geval II.

Stel dat S de vraag q_m beantwoordt. Dan geldt volgens (4)

$$\hat{q}_m/W \in A_m^S \tag{35}$$

en volgens (27)

$$\hat{q}_m/W^* = \hat{q}_m/W. \tag{36}$$

Dus als T q_m beantwoordt, dan zal het antwoord hetzelfde zijn als dat gegeven door S. Het is daarom voldoende te bewijzen dat T niet zal weigeren; d.w.z. dat

$$\neg c_4(W^*, K^*, A_{m-1}, q_m).$$

Veronderstel eens dat T wel zou weigeren q_m te beantwoorden. Dat zou betekenen, vgl. (9), dat een of ander geheim k^* in K^* onthuld zou worden als T zou antwoorden; oftewel

$$A_{m-1} \cup \{\hat{q}_m/W^*\} \Rightarrow k^*. \tag{37}$$

Toepassen van (36) geeft

$$A_{m-1} \cup \{\hat{q}_m/W\} \Rightarrow k^*. \tag{38}$$

Volgens (29) is er een i, $1 \leq i \leq n$, zodanig dat

$$k^* = (\hat{q}_i/W^*) \wedge A_{i-1}^S, \tag{39}$$

waarbij S weigerde om q_i te beantwoorden. Dus

$$A_{m-1} \cup \{\hat{q}_m/\mathcal{W}\} \Rightarrow \hat{q}_i/\mathcal{W}^* \wedge A^S_{i-1}. \tag{40}$$

En omdat S q_i weigerde, bestaat er dus een geheim k' in K zodanig dat

$$A^S_{i-1} \cup \{\hat{q}_i/\mathcal{W}\} \Rightarrow k'. \tag{41}$$

Derhalve volgt, als tenminste $\hat{q}_i/\mathcal{W} = \hat{q}_i/\mathcal{W}^*$, uit (40) en (41) dat

$$\exists k' \in K: A_{m-1} \cup \{\hat{q}_m/\mathcal{W}\} \Rightarrow k' \tag{42}$$

wat in tegenspraak is met onze aanname dat S de vraag q_m beantwoordt.
Omdat $\hat{q}_m/\mathcal{W}$ en alle formules in A_{m-1} juist zijn in $\mathcal{W}$, volgt uit (40) dat

$$\mathcal{W} \models \hat{q}_i/\mathcal{W}^* \tag{43}$$

en dus dat inderdaad geldt dat

$$\hat{q}_i/\mathcal{W} = \hat{q}_i/\mathcal{W}^* \tag{44}$$

waarmee ons bewijs voltooid is. □

11. ENKELE SLOTOPMERKINGEN

We hebben een tamelijk formele beschrijving gegeven van vragen beantwoordende systemen die, door soms een antwoord te weigeren, proberen geheimen te bewaren. Met behulp van deze beschrijving hebben we twee stellingen bewezen over het geheim houden van bepaalde informatie.

Onze resultaten zijn nogal theoretisch en zijn ook algemeen van toepassing. We hebben niets verondersteld over hoe een gegevensbank er uit moet zien. Als hij maar een hoeveelheid gegevens bevat waarover vragen gesteld kunnen worden. Ook hebben we ons niet beperkt tot eindigheid; ook al worden de alternatieve geheimen in stelling 10.3

oneindige expressies als een oneindige verzameling als veronderstelde begin-informatie wordt toegestaan. Tenslotte staan we alle mogelijke afleidingsregels toe. Deze algemeenheid heeft ook een praktisch nadeel: er bestaat geen algemene procedure die kan beslissen of een bepaalde formule logisch af te leiden valt uit een gegeven verzameling formules. En als zo'n procedure al bestaat voor een gegeven systeem, dan zal hij meestal teveel tijd en geheugenruimte vergen om bruikbaar te zijn.

Toekomstige onderzoek zou zich kunnen richten op twee problemen. Ten eerste: wat gebeurt er als we onze aannames afzwakken; b.v. als het systeem ook foute antwoorden mag geven of willekeurig zo nu en dan een antwoord mag weigeren, of als de gebruiker niet weet welke censor het systeem gebruikt of welke informatie hij verondersteld wordt te hebben, of als de inhoud van de gegevensbank aan veranderingen onderhevig mag zijn. Vooral dit laatste lijkt veelbelovend. Immers, als er gegevens veranderd zijn, zal een deel van de vrijgegeven informatie niet meer juist zijn. Bovendien weet een gebruiker dan niet meer of eerder verkregen informatie nog wel juist (genoeg) is. Daardoor zou het wel eens mogelijk kunnen zijn dat, als het aantal vragen en het aantal veranderingen zich gunstig genoeg verhouden, het systeem zich niet na verloop van tijd zal hoeven te beperken tot het beantwoorden van vragen die al eerder gesteld waren.

Ten tweede: hoe kunnen zulke veiligheids-systemen gemaakt worden? Voor wat voor vragen is het voor het systeem doenlijk om te bepalen of een antwoord een geheim zou onthullen? En als dat niet doenlijk is, bestaat er dan een wel uitvoerbare benadering die aan de veilige kant blijft?

12. DANKBETUIGINGEN

W. de Jonge heeft dit onderzoek gedaan in dienst van de Nederlands Organisatie voor Zuiver Wetenschappelijk Onderzoek (ZWO). G.L. Sicherman genoot een beurs van IBM ter gelegenheid van het 100-jarig bestaan van de Vrije Universiteit.
Tenslotte bedanken we A.S. Tanenbaum voor het maken van enkele nuttige opmerkingen.

LITERATUUR

[1] BANCILHON, F.M. & N. SPYRATOS, Protection of Information in Relational Data Bases, Proc. VLDB 1977, blz. 494-500.

[2] CHANG, C.L, DEDUCE 2: Further Investigations of Deduction in Relational Data Bases, in GALLAIRE, H. & J. MINKER (eds.), "Logic and Data Bases", Plenum Press, New York, 1978, blz. 201-236.

[3] DENNING, D.E, Are statistical databases secure?, Proc. AFIPS 1978 NCC, Vol.47, AFIPS Press, Montvale, N.J., blz. 525-530.

[4] DENNING, D.E. & P.J. DENNING, Data Security, ACM Computing Surveys, Vol.11, No.3, Sept. 1979, blz. 227-249.

[5] FUTO, I., F. DARVAS & P. SZEREDI, The Application of PROLOG to the Development of QA and DBM Systems, in GALLAIRE, H. & J. MINKER (eds.), "Logic and Data Bases", Plenum Press, New York, 1978, blz. 347-376.

[6] GALLAIRE, H. & J. MINKER (eds.), Logic and Data Bases, Plenum Press, New York, 1978,

[7] REITER, R, Deductive Question-Answering on Relational Data Bases, in GALLAIRE, H. & J. MINKER (eds.), "Logic and Data Bases", Plenum Press, New York, 1978, blz. 149-178.

THE CONCEPT OF DATA MODEL*

W.J. KLEEFSTRA
Technische Hogeschool, Twente

0. INTRODUCTION

The research in the area of data models, reported in this paper, was carried out by the author during the time he was attached to the Informatics group of the Department of Applied Mathematics at Twente University of Technology. A full report is provided by KLEEFSTRA [1].

In this paper an informal definition of the concept of data model is given, and a formalism is presented for partially specifying data models in accordance with that definition. With the help of this specification method, a comparison between several data models is made.

1. DATA MODELS: AN INFORMAL DISCUSSION

The term "data model" is easily misunderstood. According to WEBSTER [2] a model is "a small copy or imitation of an existing object (e.g. of a ship); a hypothetical or stylized representation, e.g. of an atom". A data model, however, in the sense in which the word is mostly used, is not some stylized representation or the like of something else. A data model is not in itself a model of something; it is rather a way of modelling - i.e. representing - (information about) object systems, and a way of manipulating representations.

The relational model, for instance, involves a certain way of representing information; a relational database is a collection of data items represented in that way. The relational model in addition involves a certain way of managing relational databases and of processing relational data [3].

---

* Huidige werkkring: Centrale Directie Organisatie en Efficiëncy, Ministerie van Cultuur, Recreatie en Maatschappelijk Werk, Postbus 5406, 2280 HK Rijswijk.

The smallest data items that can be individually stored into (and removed from) a database and that assumedly represent facts about some object system will be called formulas *). The formulas are also the data items whose presence in the database can be individually inspected. Both aspects of a data model, viz. the one concerning the way of representing information and the one concerning the way of manipulating representations, will now be discussed in terms of formulas.

The first aspect, called structural, determines the way in which information can be represented, that is

(1) what kind of formulas are available for representation;

(2) what kind of collections of formulas can constitute a database state; and

(3) what kind of transitions can occur between database states.

The second aspect, called operational, determines the way in which representations can be manipulated, that is

(1) how formulas (and their components) can be processed;

(2) how the presence of formulas in a database can be inspected; and

(3) how formulas can be stored into and removed from a database.

The structural and operational aspects of a data model are obviously connected. The way in which formulas can be processed is connected with the kind of formulas. The way in which the presence of formulas can be inspected is connected with the kind of formula collections that can constitute a database state. And the way in which formulas can be stored into and removed from a database is connected with the kind of database state transitions. In this paper only the structural aspects of data models and in particular the first of these, viz. about the kind of formulas available for representation, are considered.

The kind of formulas is of course not related to specific symbols occuring in formulas, but to certain structural properties of formulas. These structural properties concern the way in which formulas of the given kind are composed. For example: of what kind the (predicate logic) formula '$p(i_1,i_2,i_3)$' is, is not determined by the particular constants 'p','i_1', 'i_2' and 'i_3' that occur in it, but is determined by the fact that the formula is composed of a predicate constant and a tuple of individual constants.

---

*) These formulas correspond broadly to the atomic formulas of logic.

Another example of a structural property is the constraint that the attribute names within a relation must be distinct.*)

In addition to structural properties of individual formulas, such as those mentioned above, a data model may involve structural properties concerning the set of available formulas as a whole. For example: according to the relational model formulas with the same relation name also include the same set of attribute names. Similarly, structural properties may concern database states (which are composed of formulas) or database state transitions (which are pairs of database states). A data model can therefore be considered a collection of structural properties. This notion of structural property will be formalized in section 2.

2. CONSTRUCT TYPES: A FORMALISM

I will first discuss ways in which formal objects such as formulas can be *composed* of other formal objects, starting with some given objects regarded as non-decomposable. All these formal objects are called *constructs*. Non-decomposable constructs are called *atoms*.

Constructs can be composed of atoms in the following ways. A construct is *atomic* (i.e. identical to an atom), *labelled* or *composite*. A labelled construct consists of another construct, its *direct component*, and an associated *label*. A composite construct is either a finite *bag* (i.e. an unordered collection of other constructs, which are its direct components) or a finite *tuple* (i.e. an ordered collection of direct component constructs). For example: if c_1 and c_2 are constructs and L is a label, then the following also are constructs:

- labelled constructs: $L \div c_1$ and $L \div c_2$;
- bags: $[]$ and $[c_1,c_2]$ (which is the same as $[c_2,c_1]$);
- tuples: $<>$, $<c_1,c_2>$ and $<c_2,c_1>$.

The direct components of a composite or labelled construct are the different construct occurrences within it. For instance, the tuple <c,c> has

---

*) In case of the relational model, a formula corresponds to a tuple or a row of a table, but includes the relation name and the attribute names as well.

two direct components: c and c. Accordingly the direct components of a construct form a bag, not a set. The - direct or indirect - *components* of a construct are defined as follows. An atomic construct has no components. The components of a labelled construct are its direct component and the components thereof. A composite construct has the following components: (1) its direct components, and (2) the bag-union of the components of each of its direct components. For example: if c_1, c_2 and c_3 are atomic, then $\langle\langle c_1,c_2\rangle,[c_3,c_2,c_3]\rangle$ has seven components: $\langle c_1,c_2\rangle$ and $[c_3,c_2,c_3]$, and c_1, c_2, c_3, c_2 and c_3.

The atoms of which the constructs are composed are drawn from finite *atom classes* (i.e. sets of atoms). Different atom classes need not be disjoint. Atom classes are associated with *atom types*. The atoms in a class associated with a certain atom type are said to be of that type.

Constructs have formal properties related to the way in which they are composed. All such *structural properties* can be defined (using e.g. first-order logic) in terms of a small number of basic structural predicates and functions:

(In the following c,c_1 and c_2 are constructs; n, n_1 and n_2 are natural numbers; L is a label; t is an atom type; and x_1 and x_2 are any of these.)

- IS-ATOMIC(c) means that c is an atom.
- IS-LABELLED(c) means that c is a labelled construct.
- IS-BAG(c) means that c is a bag.
- IS-TUPLE(c) means that c is a tuple.
- HAS-TYPE(c,t) means that c is an atom of type t.
- DEGREE(c) = n means that the degree (i.e. the number of direct components) of c is n.
- LABEL(c) = L means that L is the label of c.
  LABEL(c) is undefined if c is not labelled.
- MULT(c_1,c_2) = n means that the multiplicity with which c_1 occurs as a direct component of c_2 is n.
  MULT(c_1,c_2) = 0 if c_2 is atomic.
- HAS-POS(c_1,c_2,n) means that c_1 occurs as a direct component of c_2 at position n.
- $x_1 = x_2$ means that x_1 is identical with x_2.
- $n_1 > n_2$ means that n_1 is greater than n_2.

The following are for instance the structural properties of a construct c that is a 2-tuple of which the first component is an atom drawn from a class associated with the atom type A and the second component is an atom

drawn from a class associated with the atom type $\dot{B}$.

IS-TUPLE(c)

DEGREE(c) = 2

$\exists\, c_1$: (HAS-POS(c_1,c,1) $\wedge$ HAS-TYPE(c_1,$\dot{A}$))

$\exists\, c_2$: (HAS-POS(c_2,c,2) $\wedge$ HAS-TYPE(c_2,$\dot{B}$))

In fact the universe of all such constructs is precisely characterized by the conjunction of these four properties. (One may therefore introduce a defining predicate, for instance IS-A/B-PAIR(c), which is equivalent to this conjunction.) Such a characterization of constructs is in terms of their composition structure and may include references to particular labels, particular atom types and particular natural numbers, but must not include references to particular (atomic) constructs. (In terms of first-order logic: the language may include the predicate symbols and the function symbols for the basic predicates and functions listed above, and it may include constants for labels, atom types and natural numbers, but it must not include constants for atoms or other constructs.) A particular class of constructs having the given properties is only determined after specific atom classes have been associated with the atom types in question (e.g. the cartesian product *A X B* after associating the classes *A* and *B* with the types $\dot{A}$ and $\dot{B}$ respectively).

A *construct type* is a collection of structural properties. The atom types playing a role in (the properties of) a construct type form the *basis* of that construct type. Not all collections of structural properties are construct types, however. Which collections are will not be defined in a direct way, but in terms of the *construct classes* determined by the construct types under an arbitrary association of atom classes with the atom types in their bases. The collections of structural properties that all constructs of such classes have in common are the construct types. This way of defining construct types is reflected by the denotation used for them. Construct types are denoted in terms of their atom types. (These denotations are not unique).

In the following I will specify - using induction - how to generate the set T^O of all construct types the basis of which is a subset of a given set T of atom types. The rules in the specification are of four kinds:

(a) a denotation and terminology for the construct types in T^O;

(b) a specification of the bases of the construct types in T^O (the basis of t will be denoted by Basis_t);

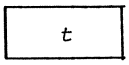
(c) a specification of the construct types in T^O in terms of the classes of constructs that they determine under the association A, which associates the atom classes C_{t_i} with the atom types t_i in T (the class of constructs determined by t under A will be denoted by $Class^A_t$); and

(d) a diagram notation for the construct types in T^O.

Examples of construct types will also be given. These examples show construct types that are collections of structural properties of formulas according to different data models.

1. (a) If $t \in T$, then $t \in T^O$.
   t is called an *atom type*.
   (b) $Basis_t = \{t\}$.
   (c) $Class^A_t = C_t$.
   Constructs of an atom type are atomic constructs.

   (d)

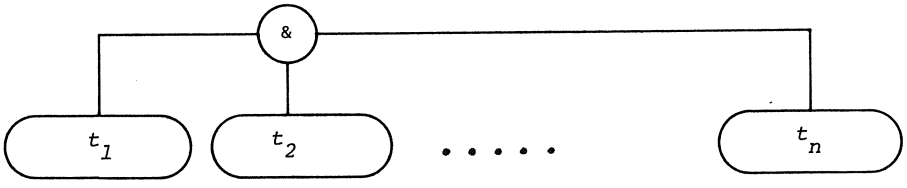


EXAMPLE: *Proposition-Constant*

When proposition logic is used as the basis of a data model, a formula is just a single symbol, viz. a proposition constant.

2. (a) For every finite $n(n\geq 2)$, if $t_i \in T^O$ $(1\leq i\leq n)$ and not all t_i are the same, then $(t_1\&t_2\&\ldots\&t_n) \in T^O$.
   $(t_1\&t_2\&\ldots\&t_n)$ is called an *n-bag type*.
   $t_1, t_2, \ldots$ and t_n are called the *component types* of the type.
   n is called the *degree* of the type.
   (b) $Basis_{(t_1\&t_2\&\ldots\&t_n)} = \coprod_{i=1}^{n} Basis_{t_i}$.
   (c) $Class^A_{(t_1\&t_2\&\ldots\&t_n)} = \{[c_1,c_2,\ldots,c_n]$: for all $1 \leq i \leq n$ $c_i \in Class^A_{t_i}\}$.
   Constructs of an n-bag type are bags of degree n.

   (d)



3. (a) For every finite $n(n \geq 2)$, if $t_i \in T^O$ $(1 \leq i \leq n)$ and not all t_i are the same, then $(t_1 * t_2 * \ldots * t_n) \in T^O$.

$(t_1 * t_2 * \ldots * t_n)$ is called an *n-tuple type*.

$t_1, t_2, \ldots$ and t_n are called the *component types* of the type.

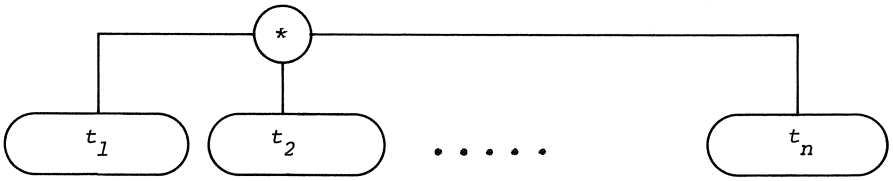
n is called the *degree* of the type.

(b) $\text{Basis}_{(t_1 * t_2 * \ldots * t_n)} = \bigcup_{i=1}^{n} \text{Basis}_{t_i}$.

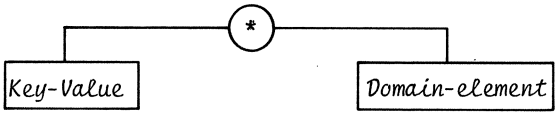
(c) $\text{Class}^A_{(t_1 * t_2 * \ldots * t_n)} = \{\langle c_1, c_2, \ldots, c_n \rangle$: for all $1 \leq i \leq n$ $c_i \in \text{Class}^A_{t_i}\}$.

Constructs of an n-tuple type are tuples of degree n.

(d)



EXAMPLE:



In the simple binary relational model, mentioned by COLOMBETTI et al. [4], a formula is a pair consisting of a key value and a domain element.

4. (a) If $t \in T^O$, then $(t)^{\&} \in T^O$.

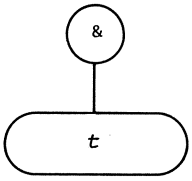
$(t)^{\&}$ is called a *bag type*.

(b) $\text{Basis}_{(t)^{\&}} = \text{Basis}_t$.

(c) $\text{Class}^A_{(t)^{\&}} = \{[c_1, c_2, \ldots, c_k]$:
$k \geq 0$ and finite, and for all $1 \leq i \leq k$ $c_i \in \text{Class}^A_t\}$.

Constructs of a bag type are finite bags.

(d)



5. (a) If $t \in T^O$, then $(t)^* \in T^O$.

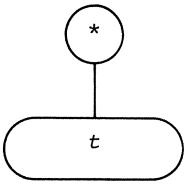
   $(t)^*$ is called a *tuple type*.

   (b) $Basis_{(t)^*} = Basis_t$.

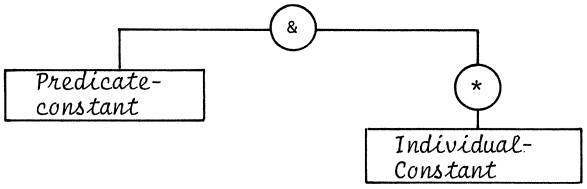
   (c) $Class^A_{(t)^*} = \{\langle c_1, c_2, \ldots, c_k \rangle :$
   $k \geq 0$ and finite, and for all $1 \leq i \leq k$ $c_i \in Class^A_t\}$.

   Constructs of a tuple type are finite tuples.

   (d)



EXAMPLE:



When predicate logic is used as the basis of a data model, a formula - in which function constants may not occur - is a 2-bag consisting of a predicate constant and a tuple of individual constants.

6. (a) If $t \in T^O$, then $L \div t \in T^O$.

   $L \div t$ is called a *labelled type*.

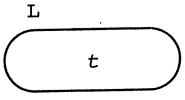
   L is called the *label* of the type.

   (b) $Basis_{L \div t} = Basis_t$.

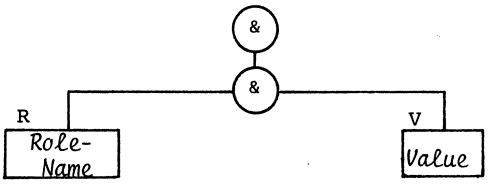
   (c) $Class^A_{L \div t} = \{L \div c : c \in Class^A_t\}$.

   Constructs of a labelled type are labelled constructs.

   (d)



EXAMPLE:



In the object/role model, used for instance by FALKENBERG [5], a formula is a set - hence a bag - of 2-bags consisting of a role name and a value. Because a symbol may be used both as a role name and as a value, the labels have been added for identification.

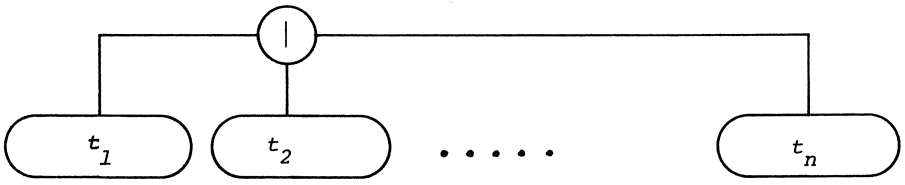
7. (a) For every finite $n(n\geq 2)$, if $t_i \in T^O$ $(1\leq i\leq n)$, then
$(t_1|t_2|\ldots|t_n) \in T^O$.
$(t_1|t_2|\ldots|t_n)$ is called a *selection type*.
$t_1, t_2, \ldots$ and t_n are called the *alternatives* of the type.

   (b) $\text{Basis}_{(t_1|t_2|\ldots|t_n)} = \bigcup_{i=1}^{n} \text{Basis}_{t_i}$.

   (c) $\text{Class}^{A}_{(t_1|t_2|\ldots|t_n)} = \bigcup_{i=1}^{n} \text{Class}^{A}_{t_i}$.

   Constructs of a selection type are constructs of at least one of its alternatives.

   (d)



8. (a) If $t \in (T\cup\{m\})^O$, where $m \in \text{Basis}_t$ but m is neither a type in T nor a type marker in t, then $^m(t)^m \in T^O$.
$^m(t)^m$ is called a *recursive type*.
m is called the *type marker* of the type.

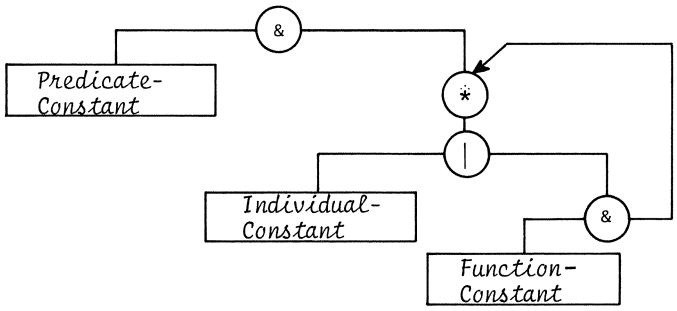
   (b) $\text{Basis}_{^m(t)^m} = \text{Basis}_t - \{m\}$.

   (c) $\text{Class}^{A}_{^m(t)^m} = \bigcup_{i\geq 1} \text{Class}^{A_i}_{t}$, where all A_i $(i\geq 1)$ are like A, except
that A_1 associates $\{\}$ with m and
that $A_i (i>1)$ associates $\bigcup_{j=1}^{i-1} \text{Class}^{A_j}_{t}$ with m.

Constructs of type $^{m}(t)^{m}$ are like constructs of type t, except that the components of the atom type m have been replaced with constructs of type $^{m}(t)^{m}$ itself.

(d) See example.

EXAMPLE:



When predicate logic is used as the basis of a data model, a formula - which may include function constants - is a 2-bag consisting of a predicate constant and a tuple of further components. Such a component is either an individual constant or a 2-bag consisting of a function constant and - again - a tuple of such components.

9. Every construct type in T^{O} can be generated by finite use of the rules 1 - 8.

Note: parentheses may be omitted if no confusion can occur. The suffix constructors for bag types and tuple types ($^{\&}$ and *) are assumed to have the highest priority, followed by the prefix constructor for labelled types (L÷). The infix constructors for selection types, n-bag types and n-tuple types (|, & and *) are assumed to have the lowest priority.

As mentioned before, one construct type may have several denotations. The most apparent forms of construct type identities are the following:
(t_1, t_2 and t_3 are construct types; L is a label; and m is a type marker)

(a) arising from the idempotency, commutativity and associativity of the selection type constructor:

$t|t = t$

$t_1|t_2 = t_2|t_1$

$t_1|(t_2|t_3) = t_1|t_2|t_3$

(b) arising from the distributivity of the labelled type constructor over the selection type constructor:

$L \div (t_1 | t_2) = (L \div t_1) \mid (L \div t_2)$

(c) arising from the commutativity of the n-bag type constructor:

$t_1 \,\&\, t_2 = t_2 \,\&\, t_1$

(d) arising from the distributivity of the n-bag and n-tuple type constructors over the selection type constructor:

$t_1 \,\&(t_2 | t_3) = (t_1 \& t_2) \mid (t_1 \& t_3)$

$t_1 * (t_2 | t_3) = (t_1 * t_2) \mid (t_1 * t_3)$

Other forms of construct type identities are possible with recursive types. For example:

$$^{m}(t_1 \& (t_2 | m)^{*})^{m} = t_1 \,\&\, ^{m}((t_2 | (t_1 \& m))^{*})^{m}.$$

In section 1, a data model has been presented as a collection of structural properties, viz. of formulas, of sets of formulas, of database states and of database state transitions. Certain collections of structural properties are construct types, as defined above. Construct types can consequently be used to partially specify data models. Any remaining structural properties of data models must be specified otherwise, e.g. using first-order logic with the basic predicates and functions discussed in this section.

3. COMPARING DATA MODELS USING CONSTRUCT TYPES

Construct types are used in this section as partial specifications of structural formula properties of data models. As these construct types specify the skeletons of the formulas for the different data models, they might be called formula skeletons.

The most simple formula skeleton is the propositional one (A): a formula is a single symbol. Next comes formula skeleton (B), in which a formula is a bag of symbols. Probably because of the semantic ambiguity of such formulas, no known data model has this skeleton. The ambiguity may be avoided in two ways: by ordering the symbols or by replacing them with 2-bags of labelled symbols. The first way leads to formula skeleton (C) of the single name category data model [6]. The second leads to formula skeleton (D) of the object/role data model [5]. Both skeletons can be made still more

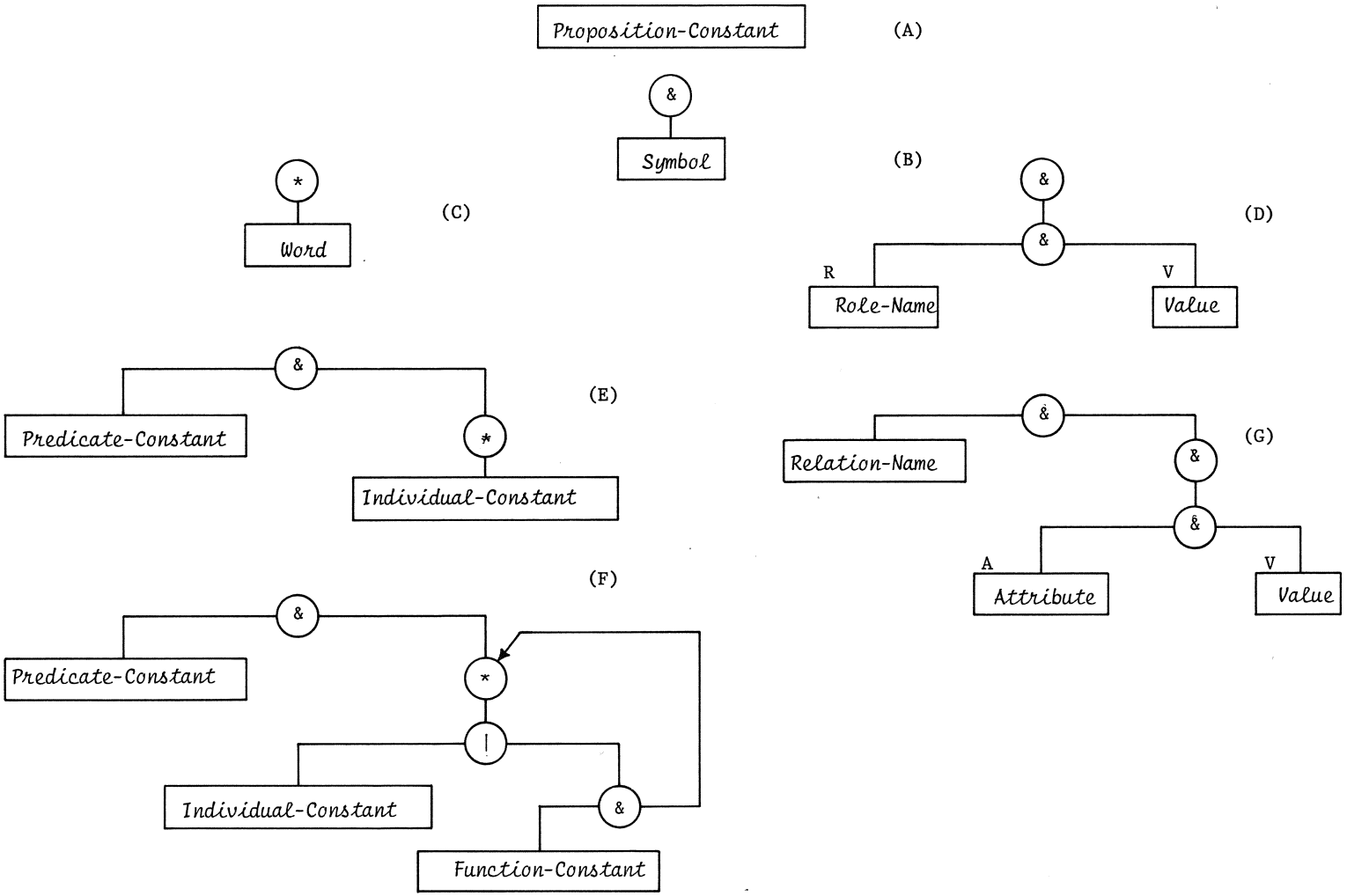
complex by the addition of an atomic component to the formula. Skeleton (C) then becomes the formula skeleton of predicate logic (E), which can be further extended by allowing function constants (F); skeleton (D) on the other hand becomes the formula skeleton of the relational model (G).

In these examples three ways of replacing a formula skeleton (or a construct type) with a more complex one are illustrated.

(1) Skeleton (F) may be obtained from (E) by the introduction of a selection type. As a result, every (E)-formula is also an (F)-formula. (E) is therefore called a sub-skeleton of (F).

(2) Skeleton (E) can be obtained from (C) by the use of (C) as a component type within (E). As a result, (C)-formulas also occur as components of (E)-formulas. (C) is therefore called a component skeleton of (E). Similarly, (D) is a component skeleton of (G), and (A) is a component skeleton of (B).

(3) Skeleton (C) can be obtained from (B) by the replacement of a bag type with a tuple type. As a result, (B)-formulas become (C)-formulas when their components are ordered. (B) is therefore called a disordering of (C).


Proposition-Constant
(A)
&
Symbol
(B)
*
Word
(C)
&
&
R
Role-Name
V
Value
(D)
&
Predicate-Constant
*
Individual-Constant
(E)
&
Relation-Name
&
&
A
Attribute
V
Value
(G)
(F)
&
Predicate-Constant
*
|
Individual-Constant
&
Function-Constant

4. CONCLUSION

A formalism - in terms of structural properties and construct types - has been presented, on the basis of which the concept of data model can be formally defined and a data model specification method can be developed.

Definition and specification method have only been discussed with regard to the structural aspects of data models. It is my opinion, however, that they can form the basis of a formalization of the operational aspects as well.

REFERENCES

[1] KLEEFSTRA, W.J., *The concept of data model in database systems,* Thesis, Twente University of Technology (1980).

[2] Webster's New world dictionary (2nd ed.).

[3] CODD, E.F., *Extending the data base relational model to capture more meaning,* ACM Transactions on Database Systems 4,4 (Dec. 1979), 397-434.

[4] COLOMBETTI, M., et al., *Nondeterministic languages used for the definition of data models*. In H. Gallaire, J. Minker (eds): Logic and data bases. Plenum Press, New York (1978), 237-257.

[5] FALKENBERG, E.D., *Significations: the key to unify database management,* Information Systems 2,1 (1976), 19-28.

[6] KLEEFSTRA, W.J., *Database description with a single name category data model*. In S. Bing Yao (ed.): Procs 4th Int. Conf. on 'Very Large Data Bases', West-Berlin, September 1978. IEEE Publ. 78CH 1389-6C, New York (1978), 177-185.

UITGAVEN IN DE SERIE MC SYLLABUS

Onderstaande uitgaven zijn verkrijgbaar bij het Mathematisch Centrum, 2e Boerhaavestraat 49 te Amsterdam-1005, tel. 020-947272.

MCS 1.1 F. GÖBEL & J. VAN DE LUNE, *Leergang Besliskunde, deel 1: Wiskundige basiskennis*, 1965. ISBN 90 6196 014 2.

MCS 1.2 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 2: Kansberekening*, 1965. ISBN 90 6196 015 0.

MCS 1.3 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 3: Statistiek*, 1966. ISBN 90 6196 016 9.

MCS 1.4 G. DE LEVE & W. MOLENAAR, *Leergang Besliskunde, deel 4: Markovketens en wachttijden*, 1966. ISBN 90 6196 017 7.

MCS 1.5 J. KRIENS & G. DE LEVE, *Leergang Besliskunde, deel 5: Inleiding tot de mathematische besliskunde*, 1966. ISBN 90 6196 018 5.

MCS 1.6a B. DORHOUT & J. KRIENS, *Leergang Besliskunde, deel 6a: Wiskundige programmering 1*, 1968. ISBN 90 6196 032 0.

MCS 1.6b B. DORHOUT, J. KRIENS & J.TH. VAN LIESHOUT, *Leergang Besliskunde, deel 6b: Wiskundige programmering 2*, 1977. ISBN 90 6196 150 5.

MCS 1.7a G. DE LEVE, *Leergang Besliskunde, deel 7a: Dynamische programmering 1*, 1968. ISBN 90 6196 033 9.

MCS 1.7b G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7b: Dynamische programmering 2*, 1970. ISBN 90 6196 055 X.

MCS 1.7c G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7c: Dynamische programmering 3*, 1971. ISBN 90 6196 066 5.

MCS 1.8 J. KRIENS, F. GÖBEL & W. MOLENAAR, *Leergang Besliskunde, deel 8: Minimaxmethode, netwerkplanning, simulatie*, 1968. ISBN 90 6196 034 7.

MCS 2.1 G.J.R. FÖRCH, P.J. VAN DER HOUWEN & R.P. VAN DE RIET, *Colloquium Stabiliteit van differentieschema's, deel 1*, 1967. ISBN 90 6196 023 1.

MCS 2.2 L. DEKKER, T.J. DEKKER, P.J. VAN DER HOUWEN & M.N. SPIJKER, *Colloquium Stabiliteit van differentieschema's, deel 2*, 1968. ISBN 90 6196 035 5.

MCS 3.1 H.A. LAUWERIER, *Randwaardeproblemen, deel 1*, 1967. ISBN 90 6196 024 X.

MCS 3.2 H.A. LAUWERIER, *Randwaardeproblemen, deel 2*, 1968. ISBN 90 6196 036 3.

MCS 3.3 H.A. LAUWERIER, *Randwaardeproblemen, deel 3*, 1968. ISBN 90 6196 043 6.

MCS 4 H.A. LAUWERIER, *Representaties van groepen*, 1968. ISBN 90 6196 037 1.

MCS 5 J.H. VAN LINT, J.J. SEIDEL & P.C. BAAYEN, *Colloquium Discrete wiskunde*, 1968.
ISBN 90 6196 044 4.

MCS 6 K.K. KOKSMA, *Cursus ALGOL 60*, 1969. ISBN 90 6196 045 2.

MCS 7.1 *Colloquium Moderne rekenmachines, deel 1*, 1969. ISBN 90 6196 046 0.

MCS 7.2 *Colloquium Moderne rekenmachines, deel 2*, 1969. ISBN 90 6196 047 9.

MCS 8 H. BAVINCK & J. GRASMAN, *Relaxatietrillingen*, 1969.
ISBN 90 6196 056 8.

MCS 9.1 T.M.T. COOLEN, G.J.R. FÖRCH, E.M. DE JAGER & H.G.J. PIJLS, *Elliptische differentiaalvergelijkingen, deel 1*, 1970.
ISBN 90 6196 048 7.

MCS 9.2 W.P. VAN DEN BRINK, T.M.T. COOLEN, B. DIJKHUIS, P.P.N. DE GROEN, P.J. VAN DER HOUWEN, E.M. DE JAGER, N.M. TEMME & R.J. DE VOGELAERE, *Colloquium Elliptische differentiaalvergelijkingen, deel 2*, 1970.
ISBN 90 6196 049 5.

MCS 10 J. FABIUS & W.R. VAN ZWET, *Grondbegrippen van de waarschijnlijkheidsrekening*, 1970. ISBN 90 6196 057 6.

MCS 11 H. BART, M.A. KAASHOEK, H.G.J. PIJLS, W.J. DE SCHIPPER & J. DE VRIES, *Colloquium Halfalgebra's en positieve operatoren*, 1971.
ISBN 90 6196 067 3.

MCS 12 T.J. DEKKER, *Numerieke algebra*, 1971. ISBN 90 6196 068 1.

MCS 13 F.E.J. KRUSEMAN ARETZ, *Programmeren voor rekenautomaten; De MC ALGOL 60 vertaler voor de EL X8*, 1971. ISBN 90 6196 069 X.

MCS 14 H. BAVINCK, W. GAUTSCHI & G.M. WILLEMS, *Colloquium Approximatietheorie*, 1971. ISBN 90 6196 070 3.

MCS 15.1 T.J. DEKKER, P.W. HEMKER & P.J. VAN DER HOUWEN, *Colloquium Stijve differentiaalvergelijkingen, deel 1*, 1972. ISBN 90 6196 078 9.

MCS 15.2 P.A. BEENTJES, K. DEKKER, H.C. HEMKER, S.P.N. VAN KAMPEN & G.M. WILLEMS, *Colloquium Stijve differentiaalvergelijkingen, deel 2*, 1973. ISBN 90 6196 079 7.

MCS 15.3 P.A. BEENTJES, K. DEKKER, P.W. HEMKER & M. VAN VELDHUIZEN, *Colloquium Stijve differentiaalvergelijkingen, deel 3*, 1975.
ISBN 90 6196 118 1.

MCS 16.1 L. GEURTS, *Cursus Programmeren, deel 1: De elementen van het programmeren*, 1973. ISBN 90 6196 080 0.

MCS 16.2 L. GEURTS, *Cursus Programmeren, deel 2: De programmeertaal ALGOL 60*, 1973. ISBN 90 6196 087 8.

MCS 17.1 P.S. STOBBE, *Lineaire algebra, deel 1*, 1974. ISBN 90 6196 090 8.

MCS 17.2 P.S. STOBBE, *Lineaire algebra, deel 2*, 1974. ISBN 90 6196 091 6.

MCS 17.3 N.M. TEMME, *Lineaire algebra , deel 3*, 1976. ISBN 90 6196 123 8.

MCS 18 F. VAN DER BLIJ, H. FREUDENTHAL, J.J. DE IONGH, J.J. SEIDEL & A. VAN WIJNGAARDEN, *Een kwart eeuw wiskunde 1946-1971, Syllabus van de Vakantiecursus 1971*, 1974. ISBN 90 6196 092 4.

MCS 19 A. HORDIJK, R. POTHARST & J.Th. RUNNENBURG, *Optimaal stoppen van Markovketens*, 1974. ISBN 90 6196 093 2.

MCS 20 T.M.T. COOLEN, P.W. HEMKER, P.J. VAN DER HOUWEN & E. SLAGT, *ALGOL 60 procedures voor begin- en randwaardeproblemen*, 1976. ISBN 90 6196 094 0.

MCS 21 J.W. DE BAKKER (red.), *Colloquium Programmacorrectheid*, 1975. ISBN 90 6196 103 3.

MCS 22 R. HELMERS, F.H. RUYMGAART, M.C.A. VAN ZUYLEN & J. OOSTERHOFF, *Asymptotische methoden in de toetsingstheorie; Toepassingen van naburigheid*, 1976. ISBN 90 6196 104 1.

MCS 23.1 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 1*, 1976. ISBN 90 6196 105 X.

MCS 23.2 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 2*, 1976. ISBN 90 6196 115 7.

MCS 24.1 P.J. VAN DER HOUWEN, *Numerieke integratie van differentiaalvergelijkingen, deel 1: Eenstapsmethoden*, 1974. ISBN 90 6196 106 8.

MCS 25 *Colloquium Structuur van programmeertalen*, 1976. ISBN 90 6196 116 5.

MCS 26.1 N.M. TEMME (ed.), *Nonlinear analysis, volume 1*, 1976. ISBN 90 6196 117 3.

MCS 26.2 N.M. TEMME (ed.), *Nonlinear analysis, volume 2*, 1976. ISBN 90 6196 121 1.

MCS 27 M. BAKKER, P.W. HEMKER, P.J. VAN DER HOUWEN, S.J. POLAK & M. VAN VELDHUIZEN, *Colloquium Discretiseringsmethoden*, 1976. ISBN 90 6196 124 6.

MCS 28 O. DIEKMANN, N.M. TEMME (EDS), *Nonlinear Diffusion Problems*, 1976. ISBN 90 6196 126 2.

MCS 29.1 J.C.P. BUS (red.), *Colloquium Numerieke programmatuur, deel 1A, deel 1B*, 1976. ISBN 90 6196 128 9.

MCS 29.2 H.J.J. TE RIELE (red.), *Colloquium Numerieke programmatuur, deel 2*, 1976. ISBN 90 6196 144 0

* MCS 30 P. GROENEBOOM, R. HELMERS, J. OOSTERHOFF & R. POTHARST, *Efficiency begrippen in de statistiek*, . ISBN 90 6196 149 1.

MCS 31 J.H. VAN LINT (red.), *Inleiding in de coderingstheorie*, 1976. ISBN 90 6196 136 X.

MCS 32 L. GEURTS (red.), *Colloquium Bedrijfssystemen*, 1976. ISBN 90 6196 137 8.

MCS 33 P.J. VAN DER HOUWEN, *Differentieschema's voor de berekening van waterstanden in zeeën en rivieren*, 1977. ISBN 90 6196 138 6.

MCS 34 J. HEMELRIJK, *Oriënterende cursus mathematische statistiek*, ISBN 90 6196 139 4.

MCS 35 P.J.W. TEN HAGEN (red.), *Colloquium Computer Graphics*, 1977. ISBN 90 6196 142 4.

MCS 36 J.M. AARTS, J. DE VRIES, *Colloquium Topologische Dynamische Systemen*, 1977. ISBN 90 6196 143 2.

MCS 37 J.C. van Vliet (red.), *Colloquium Capita Datastructuren*, 1978. ISBN 90 6196 159 9.

MCS 38.1 T.H. KOORNWINDER (ED.), *Representations of locally compact groups with applications*, 1979. ISBN 90 6196 161 0.

MCS 38.2 T.H. KOORNWINDER (ED.), *Representations of locally compact groups with applications*, 1979. ISBN 90 6196 181 5.

MCS 39 O.J. VRIEZE & G.L.Wanrooij, *Colloquium Stochastische Spelen*, 1978. ISBN 90 6196 167 X.

MCS 40 J. VAN TIEL, *Convexe Analyse*, 1979. ISBN 90 6196 187 4.

MCS 41 H.J.J. TE RIELE (ED.), *Colloquium Numerical Treatment of Integral Equations*, 1979. ISBN 90 6196 189 0.

MCS 42 J.C. VAN VLIET (RED.), *Colloquium Capita Implementatie van Programmeertalen*, 1980. ISBN 90 6196 191 2.

MCS 43 A.M. COHEN & H.A. WILBRINK, *Eindige groepen (Een inleidende cursus)*, 1980. ISBN 90 6196 203 X

MCS 44 J.G. VERWER (ED.), *Numerical solution of partial differential equations*, 1980. ISBN 90 6196 205 6.

MCS 45 P. KLINT (red.), *Colloquium hogere programmeertalen en computerarchitectuur*, 1980. ISBN 90 6196 206 4.

MCS 46.1 P.M.G. APERS (RED.), *Colloquium Databankorganisatie*, 1981. ISBN 90 6196 212 9.

De met een * gemerkte uitgaven moeten nog verschijnen.