

Nederlandse organisatie
voor toegepast
natuurwetenschappelijk
onderzoek



Fysisch en Elektronisch
Laboratorium TNO



Postbus 96864
2509 JG 's-Gravenhage
Oude Waalsdorperweg 63
's-Gravenhage

Telefax 070 - 328 09 61
Telefoon 070 - 326 42 21

TNO-rapport

DTIC FILE COPY

rapport no.
FEL-89-A312

exemplaar no.

titel

8

Kwaliteit van Expertsystemen: Algoritmen
voor Integriteits-controle

894578

Niets uit deze uitgave mag worden
vermenigvuldigd en/of openbaar gemaakt
door middel van druk, fotokopie, microfilm
of op welke andere wijze dan ook, zonder
voorafgaande toestemming van TNO.
Het ter inzage geven van het TNO-rapport
aan direct belanghebbenden is toegestaan.

Indien dit rapport in opdracht werd
uitgebracht, wordt voor de rechten en
verplichtingen van opdrachtgever en
opdrachtnemer verwezen naar de
'Algemene Voorwaarden voor Onderzoeks-
opdrachten TNO', dan wel de betreffende
terzake tussen partijen gesloten
overeenkomst.

© TNO

auteur(s):

Drs. J.H.J. Lenting (RL)

Drs. M. Perre (FEL-TNO)

DTIC
ELECTE
SEP 14 1990
E D

TDCK RAPPORTENCENTRALE
Frederikkazerne, Geb. 140
van den Burchlaan 31
Telefoon: 070-3166394/6395
Telefax : (31) 070-3166202
Postbus 90701
2509 LS Den Haag

rubricering

titel

: ongerubriceerd

samenvatting

: ongerubriceerd

rapport

: ongerubriceerd

oplage

: 45

aantal bladzijden

: 61

aantal bijlagen

: -

datum

: maart 1990

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

80-09-13 200



AD-A226 467

rapport no. : FEL-89-A312
titel : Kwaliteit van Expertsystemen: Algoritmen voor Integriteits-controle

auteur(s) : Drs. J.H.J. Lenting en Drs. M. Perre
instituut : Fysisch en Elektronisch Laboratorium TNO

datum : maart 1990
hdo-opdr.no. : A89K602
no. in iwp '89 : 704.2

=====

SAMENVATTING

Dit rapport is het resultaat van de derde fase van het technologie-project "Kwaliteit van Expertsystemen", uitgevoerd in opdracht van het Ministerie van Defensie, Directoraat-Generaal Wetenschappelijk Onderzoek en Ontwikkeling. Deelnemers aan het project zijn het Fysisch en Elektronisch Laboratorium TNO (FEL-TNO), de Rijksuniversiteit Limburg (RL) en het Research Instituut voor Kennis-Systemen (RIKS). Uitgaande van [FEL-1989-148] en [FEL-89-A267] bevat dit rapport de resultaten van een onderzoek naar algoritmen voor de integriteits-controle van knowledgebases, waarbij aandacht is besteed aan de computationele efficiëntie. Verschillende algoritmen gericht op het waarborgen van consistentie en volledigheid van de knowledgebase worden met elkaar vergeleken. Verder is nagegaan op welke wijze er een mapping gemaakt kan worden tussen specificaties opgesteld met E(xtended)NIAM en een executeerbaar Prolog-programma, teneinde een basis te verkrijgen voor een formele vaststelling van consistentie en volledigheid van knowledgebase-specificaties.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By _____	
Distribution/ _____	
Availability Codes	
Dist	Special and/or
	Special
A-1	



report no. : FEL-89-A312
title : Quality of Expert Systems: Algorithms for Integrity Control

author(s) : J.H.J. Lenting and M. Perre
institute : TNO Physics and Electronics Laboratory

date : March 1990
NDRO no. : A89K602
no. in pow '89 : 704.2

ABSTRACT

This report is the result of the third phase of the technology project 'Quality of Expert Systems', carried out under the commission of the Ministry of Defence, Director Defence Research and Development. Participants in the project are TNO Physics and Electronics Laboratory (FEL-TNO), University of Limburg (RL) and the Research Institute for Knowledge Systems (RIKS). Based on [FEL-1989-148] and [FEL-89-A267] this report contains the results of an investigation into algorithms for integrity control of knowledgebases, with specific interest in the computational efficiency. Algorithms for preserving consistency and integrity of the knowledgebase are being compared. Another subject in this report is the way in which a mapping can be made between specifications made with E(xtended)NIAM and an executable Prolog program, in order to establish a formal determination of the consistency and integrity of knowledgebase specifications.

Keywords: Netherlands, Translations. (K.P.)

	SAMENVATTING	1
	ABSTRACT	2
	INHOUDSOPGAVE	3
1	INLEIDING	4
2	SPECIFICATIE VAN DE KNOWLEDGEBASE	7
2.1	Inleiding	7
2.2	Een uitbreiding op NIAM: E(xtended)NIAM	8
2.3	Specificatie in E(xtended)NIAM	11
2.4	Representatie in Prolog	13
3	INTEGRITEIT VAN DE KNOWLEDGEBASE	15
3.1	Inleiding	15
3.2	Algoritme van Decker	15
3.3	Algoritme van Sadri en Kowalski	26
3.4	Algoritme van Shapiro	28
3.5	Algoritme van Suwa, Scott en Shortliffe	36
3.6	Algoritme van Puuronen	37
3.7	Algoritme van Nguyen	39
3.8	Algoritme van Bezem	42
4	EEN VERGELIJKENDE ANALYSE	46
4.1	Inleiding	46
4.2	Invulling van integriteits-controle	46
4.3	Computationale efficiëntie	48
4.4	Toepasbaarheid in knowledgebase-systemen	51
5	CONCLUSIES EN AANBEVELINGEN	54
	LITERATUURVERWIJZINGEN	58

1 INLEIDING

In [FEL-89-A267] is erop gewezen dat het kwaliteitsprobleem in kennissystemen (gedeeltelijk) kan worden opgelost door gebruik te maken van een 'conventionele' ontwikkelingsmethodiek als NIAM en een 'relationele expertsysteem'-architectuur. Deze beide zijn oorspronkelijk voorgesteld door Nijssen [Wintraecken85; Nijssen86]. Ook in dit rapport spelen deze elementen weer een belangrijke rol. De aandacht is echter in het bijzonder gevestigd op de integriteitscontrole van de knowledgebase. Onder integriteit wordt hier verstaan de omstandigheid dat de kennis (en informatie) die een knowledgebase bevat consistent en volledig is [FEL1989-148]. Consistentie valt weer uiteen in een viertal factoren, t.w. tegenstrijdigheid, redundantie, subsumptie en circulariteit. Ten eerste mogen er geen tegenstrijdige regels voorkomen, d.w.z. regels met dezelfde condities, maar met verschillende acties. Ten tweede is geen redundantie toegestaan, d.w.z. regels met dezelfde condities en een of meer dezelfde acties. Ten derde mag er geen subsumptie voorkomen, d.w.z. regels met als gevolg dezelfde acties maar met een of meer dezelfde condities. Ten slotte is het niet toegestaan dat de regels circulair zijn, d.w.z. als bepaalde regels zichzelf via een of meer tussenstappen aanroepen. Volledigheid is in drie factoren te verdelen, t.w. missende regels, onbereikbare regels en 'dead-end' regels. In het geval dat de attribuutwaarden van een bepaald object niet worden gedekt in de condities van een of andere regel waardoor een bepaalde actie niet kan voorkomen, wordt gesproken van een missende regel. Als het actie-gedeelte van een regel niet direct uitvoerbaar is of de condities van een andere regel waarmaakt, dan is er sprake van een onbereikbare regel. Als een gebruiker of het actie-gedeelte van een regel niet in de gelegenheid zijn om de conditie(s) van een bepaalde andere regel waar te maken, dan wordt gesproken van een 'dead-end' regel.

Het ontwikkelproces van een systeem kan (globaal) worden verdeeld in twee stappen: allereerst moet er een transformatie plaatsvinden van een probleem naar een conceptueel model, en vervolgens van dit model naar een implementatie [FEL-89-A267]. Tijdens elk van deze transformaties kunnen zich problemen voordoen die ertoe leiden dat de specificatie en/of de uiteindelijke implementatie (ernstige) tekortkomingen bevat(ten). In [Suwa82] wordt een onderscheid gemaakt tussen het verifiëren of de knowledgebase alle noodzakelijke informatie bevat en het verifiëren of het hierop gebaseerde programma

deze informatie correct interpreteert en toepast. Met dit laatste wordt aangegeven dat de werking (en correctheid) van bijvoorbeeld invoer- en uitvoer modules weliswaar van groot belang is voor een kwalitatief goed systeem, maar dat de waarborging hiervan niet direct gerelateerd is aan de kennis die in het systeem is opgenomen.

In het verleden is al door diverse auteurs aan knowledgebase-debugging aandacht besteed. Systemen als TEIRESIAS [Davis82], RENE [Reboh81], EMYCIN [VanMelle84] en CHECK [Nguyen85], die ontstaan zijn binnen het reguliere onderzoek op het gebied van de kunstmatige intelligentie, hebben tot doel de syntactische en semantische integriteit van de knowledgebase te waarborgen. Elk van deze benaderingen gaat ervan uit dat knowledgebase-debugging een integrerend onderdeel is van het kennis-verwervingsproces. Verder zijn er auteurs die zich in eerste instantie hebben gericht op het proces van program-debugging, zoals in het Programmer's Apprentice project [Rich79] en het Model Inference System [Shapiro82]. Onafhankelijk van de benadering (knowledgebase- danwel program-debugging) is de constatering dat de integriteit van produktieregels (of programma's) niet uitsluitend op logische gronden is vast te stellen. Met name wanneer het gaat om redundantie en subsumptie, is het (pragmatische) oordeel van de ontwikkelaar het belangrijkste criterium of iets al dan niet consistent wordt genoemd, waarmee de noodzaak voor gebruikersinteractie is gegeven.

De doelstelling van dit rapport valt in twee delen uiteen. Ten eerste moet inzicht worden verkregen in algoritmen die daadwerkelijk de integriteit van een knowledgebase kunnen waarborgen. Ten tweede wordt opnieuw een methode aan de orde gesteld waarmee knowledgebase-specificaties kunnen worden opgesteld, t.w. E(xtended)NIAM. Het is met name interessant welke relatie gelegd kan worden tussen de gehanteerde ontwikkelmethode en algoritmen voor integriteitscontrole. In concreto komt dit neer op:

- Evaluatie van algoritmen om de consistentie en volledigheid van een verzameling inferentieregels vast te stellen.
- Analyse van E(xtended)NIAM als methode om een conceptueel model van een knowledgebase vast te leggen.

Het rapport is verdeeld in vijf hoofdstukken. Na de inleiding volgt in hoofdstuk 2 een beschrijving van E(xtended)NIAM en de rol die deze methode speelt bij de specificatie

van een knowledgebase. Naast de representatie van afleidings- en inferentieregels in ENIAM komt ook de omzetting van het conceptuele model in Prolog aan de orde. In hoofdstuk 3 zijn enkele algoritmen voor integriteitscontrole opgenomen. Dit rapport probeert geen volledig overzicht te geven van alle benaderingen die in de loop der tijden voor het integriteitsprobleem zijn bedacht. Er zijn globaal drie lijnen gevolgd. Ten eerste komen enkele algoritmen aan de orde uit de hoek van de deductieve gegevensbanken, ontwikkeld door Decker, Sadri en Kowalski. Ten tweede een representant van de program-debugging school, t.w. Shapiro. Ten derde wordt aandacht besteed aan de 'klassieke' algoritmen van Suwa, Puuronen en Nguyen. Hoofdstuk 4 is een vergelijkende analyse van de beschreven algoritmen. Aspecten die aan de orde komen zijn de gehanteerde definitie van consistentie en volledigheid, de computationele efficiëntie en de toepasbaarheid in knowledgebase-systemen. De conclusies en aanbevelingen staan in hoofdstuk 5, dat wordt gevolgd door een lijst met literatuurverwijzingen.

2 SPECIFICATIE VAN DE KNOWLEDGEBASE

2.1 Inleiding

Een belangrijke activiteit tijdens kennissysteem-ontwikkeling is het vastleggen van een conceptueel model. Het conceptuele model moet een volledige en consistente weergave van het kennisdomein zijn waarin, net als bij een database-toepassing, een onderscheid wordt gemaakt tussen een kennisschema (de definities van alle voorkomende feiten en relaties) en de werkelijke knowledgebase. Het voordeel is dat er nu op een geabstraheerde manier over de knowledgebase kan worden gesproken. Consistentie en volledigheid zijn te bewaken door middel van beperkingsregels die aan de werkelijke knowledgebase worden opgelegd. Een conceptueel model dient te worden opgesteld met een methode die het onderscheid tussen feiten en hun definitie expliciet vastlegt. In het fase-2 rapport [FEL-89-A267] is NIAM (Nijssens Informatie-Analyse Methode) in dit opzicht als een goede kandidaat aangewezen. Dit rapport bevat enkele relativerende opmerkingen aangaande de toepasbaarheid van NIAM op het gebied van kennissystemen. Met name het gebrek aan ondersteuning bij het consistent en volledig houden van de verzameling niet-grafische beperkingsregels is een nadeel. Inmiddels zijn er door medewerkers van Nijssen, de ontwikkelaar van NIAM, uitbreidingen aangebracht in deze methode [Creasy89]. De kern hiervan wordt gevormd door de mogelijkheid om beperkingsregels (bijvoorbeeld afleidingsregels en inferentieregels) formeel te representeren binnen wat E(xtended)NIAM wordt genoemd. Dit is een eerste stap in de richting van een omvangrijker integriteitscontrole binnen de methode, omdat de representatie van beperkingsregels het 'eenvoudig' maakt om executeerbare Prolog-programma's te laten genereren. Met de algoritmen die later in dit rapport worden behandeld kunnen deze programma's, die een vastlegging van de beperkingsregels zijn, op consistentie en volledigheid worden getoetst. Dit aspect is door [Creasy88] al aan een nader onderzoek onderworpen.

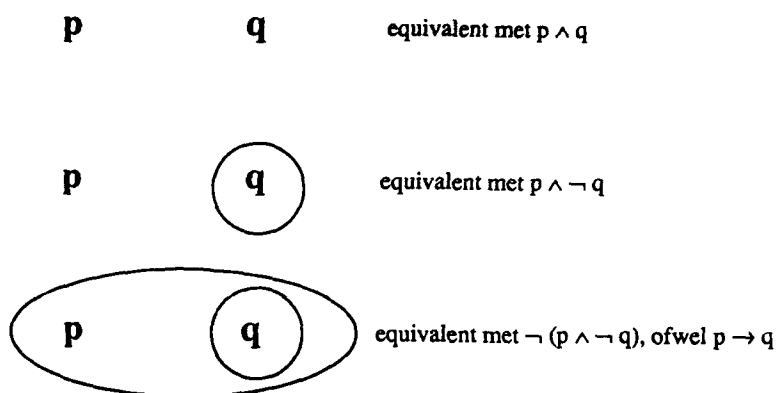
2.2 Een uitbreiding op NIAM: E(xtended)NIAM

Beperkingsregels in het algemeen zijn beschreven in [FEL-89-A267]. Een belangrijke categorie zijn de niet-grafische, zoals waarde-, cardinaliteits-, subtypebepalende, afleidings- en inferentieregels. Zij hebben gemeen dat ze niet grafisch in de informatiestructuurdiagrammen (ISD) van 'conventioneel' NIAM kunnen worden vastgelegd. Normaliter worden ze in 'natuurlijke taal' toegevoegd aan de grafische representatie. Dit is in zeker opzicht een nadeel, omdat bij de validatie van het conceptuele model de gebruiker (domeinexpert) een ISD meer 'aanspreekt' dan een stuk tekst. Bij de toepassing van NIAM in de ontwikkeling van een knowledgebase is het noodzakelijk om gebruik te maken van niet-grafische beperkingsregels. Met name de inferentieregels komen sterk overeen met wat in andere methodieken produktieregels worden genoemd. Refererend aan de KADS-methodiek [Breuker87] kan 'conventioneel' NIAM zeker op het domeinniveau een betere ondersteuning bieden dan de gangbare AI-ontwikkelmethoden. Het inferentie- en domeinniveau zijn slechts ten dele met de niet-grafische beperkingsregels te bestrijken.

Voordelen van de NIAM-methode zijn dat er een duidelijk onderscheid wordt gemaakt tussen type en instanties, en dat (een aantal) beperkingsregels grafisch kan worden vastgelegd. Zonder op deze punten concessies te doen moet een uitgebreide NIAM in staat zijn méér semantiek in de informatiestructuurdiagrammen aan te brengen. [Creasy89] heeft een aangepaste methode bedacht met de naam E(xtended)NIAM. Deze methode biedt niet alleen de mogelijkheid (zoals NIAM) om 'set-georiënteerde' beperkingsregels grafisch vast te leggen, maar ook 'member-georiënteerde'. Voorbeelden van op verzamelingen gebaseerde beperkingsregels zijn uniciteits-, uitsluitings- en totaliteitsregels. Een 'member-georiënteerde' beperkingsregel doet een uitspraak over het al dan niet kunnen voorkomen van een lid in een bepaalde verzameling.

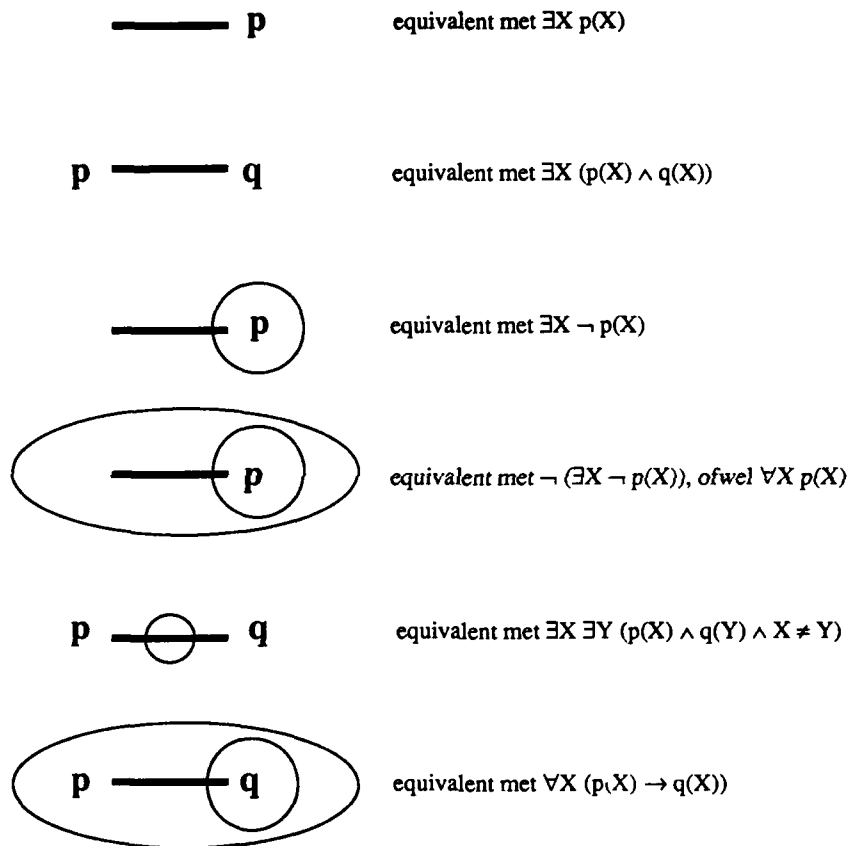
De uitbreiding is gebaseerd op 'existential graphs' [Sowa84], een grafische representatie van logische uitspraken. Er wordt gebruik gemaakt van een klein aantal symbolen met behulp waarvan met eenzelfde uitdrukingskracht heeft als de eerste orde logica. De 'existential graphs' bestaan uit twee delen, alfa en beta. Het alfa gedeelte bevat drie basiselementen, t.w. een 'sheet of assertion', een 'graph' en een 'cut'. Het 'sheet of assertion' is op te vatten als een model van de werkelijkheid. Een instantiatie van een

'graph' is gelijk aan een propositie (een uitspraak over de werkelijkheid). 'Graph'-instantiaties zijn weergegeven op het 'sheet of assertion' en worden als 'waar' verondersteld. De 'cut' is een negatie van een 'graph' en wordt weergegeven door middel van een getrokken lijn om de 'graph' heen. Voorbeelden van het alfa-gedeelte van 'existential graphs' zijn in de volgende figuur opgenomen.



Figuur 2.1: Voorbeelden van 'existential graphs' (alfa-gedeelte).

Het blijkt dat de 'existential graphs' impliciet slechts enkele logische operatoren bevatten, t.w. conjunctie en negatie. Er is echter nog een element dat afkomstig is uit de eerste orde logica, namelijk de existentiële kwantificatie $\exists$. Dit element wordt toegevoegd in het beta-gedeelte van de 'existential graphs'. Door middel van een 'line of identity' (een streep) worden individuele objecten in de werkelijkheid aangeduid. Het beta-gedeelte is te vergelijken met een predikaten-logische aanpak, terwijl het alfa-gedeelte propositie-logisch is. Voorbeelden staan in figuur 2.2.



Figuur 2.2: Voorbeelden van 'existential graphs' (beta-gedeelte).

2.3 Specificatie in E(xtended)NIAM

Uit de vorige paragraaf blijkt dat een inferentieregel (Als conditie Dan actie) betrekkelijk eenvoudig is vast te leggen met behulp van ENIAM. In deze paragraaf wordt aan de hand van een klein voorbeeld aangetoond dat (recursieve) beperkingsregels kunnen worden gerepresenteerd in ENIAM ISD's.

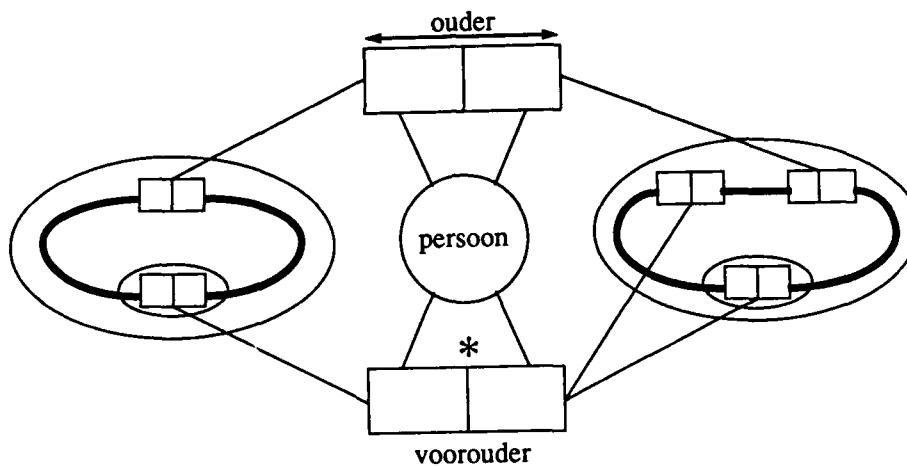
Het voorbeeld grijpt terug op het voorouder-probleem, zoals dat in [FEL-89-A267] is geschetst. Bij het oplossen van dit probleem moet eerst worden vastgesteld wanneer een persoon A voorouder is van een persoon B. Hier kunnen twee gevallen worden onderscheiden:

1. A is een voorouder van B als A een ouder is van B.
2. A is een voorouder van B als A ouder is van X en X voorouder van B.

Als men beschikt over een verzameling feiten die vastleggen wie de ouder is van welke persoon, dan kan door toepassing van een recursieve regel de verzameling voorouders van een willekeurige persoon worden bepaald. Vastlegging van een afleidingsregel als geval 2 stuit binnen ENIAM op geen enkel probleem, omdat het mogelijk is om over individuele elementen van een verzameling te spreken. Het ISD is opgenomen in figuur 2.3.

De pijl boven het feittype Ouder en het sterretje boven Voorouder hebben dezelfde betekenis als in NIAM, respectievelijk een M:N-relatie en afleidbaar feittype. Volgens de NIAM-conventies zou het ISD uitsluitend bestaan uit het objecttype Persoon en de feittypen Ouder en Voorouder. De eerder genoemde twee regels die tonen hoe het begrip voorouder is vastgelegd, zouden in NIAM slechts als een stukje tekst aan het ISD kunnen worden toegevoegd. In ENIAM kan dit grafisch gebeuren, met als toegevoegde waarde dat alle beperkingsregels binnen eenzelfde formalisme zijn vastgelegd. De algemene structuur van een als-dan regel is in de uitvergrotingen aan de linker en rechter zijde van het oorspronkelijke ISD weergegeven. In de 'existential graph' aan de linkerkant zijn de feittypen Ouder en Voorouder elk een keer opgenomen, overeenkomstig geval 1. Refererend aan het laatste voorbeeld in figuur 2.2 kan voor *p* het feittype Ouder gelezen

worden en voor q het feittype Voorouder. Er zijn twee 'lines of identity', aangezien hier met binaire feittypen gewerkt wordt. Deze geven elk één lid uit een bepaalde verzameling



Figuur 2.3: ISD van het voorouder-probleem in ENIAM.

aan. Het linkerdeel van bovenstaand ISD is equivalent met:

$$\forall \text{PersoonA, PersoonB:} \\ (\text{ouder}(\text{PersoonA}, \text{PersoonB}) \rightarrow \text{voorouder}(\text{PersoonA}, \text{PersoonB}))$$

Aan de rechterkant is het feittype Ouder een keer opgenomen en Voorouder twee keer, in overeenstemming met het (indirecte) geval 2. Dit is equivalent met:

$$\forall \text{PersoonA, TussenPersoon, PersoonB:} \\ (\text{ouder}(\text{PersoonA}, \text{TussenPersoon}) \wedge \text{voorouder}(\text{TussenPersoon}, \text{PersoonB}) \\ \rightarrow \text{voorouder}(\text{PersoonA}, \text{PersoonB}))$$

Als deze twee logische uitdrukkingen met een logische of-operator worden verbonden, dan krijgt men een expressie die het ISD van figuur 2.3 volledig vastlegt:

$$\begin{aligned} &\forall \text{PersoonA, TussenPersoon, PersoonB:} \\ &\quad (\text{ouder}(\text{PersoonA}, \text{PersoonB}) \vee \\ &\quad \text{ouder}(\text{PersoonA}, \text{Tussenpersoon}) \wedge \text{voorouder}(\text{TussenPersoon}, \text{PersoonB}) \\ &\quad \rightarrow \text{voorouder}(\text{PersoonA}, \text{PersoonB})). \end{aligned}$$

2.4 Representatie in Prolog

De logische uitdrukkingen die in paragraaf 2.4 zijn gegeven kunnen op eenvoudige wijze omgezet worden in een executeerbaar formalisme, t.w. de programmeertaal Prolog. Elk van de twee gevallen waarin het voorouder-probleem uiteen valt leiden tot een Prolog-regel. De of-relatie tussen deze regels wordt waargenomen door het interpretatie-mechanisme van Prolog; als de eerste regel, na matching van de argumenten, niet tot succes leidt, dan wordt de tweede geprobeerd. In de 'rechtstreekse' omzetting van ENIAM-specificatie naar Prolog worden feittypen weergegeven door binaire predikaten. De eerder gegeven logische uitdrukkingen hoeven slechts achterstevoren overgenomen te worden. Figuur 2.4. is hiervan een weergave.

```
voorouder (PersoonA, PersoonB) :-  
    ouder (PersoonA, PersoonB) .  
  
voorouder (PersoonA, PersoonB) :-  
    ouder (PersoonA, TussenPersoon) ,  
    voorouder (TussenPersoon, PersoonB) .
```

Figuur 2.3: Voorouder-probleem in Prolog.

Om de integriteit van het conceptuele model van het voorouder-probleem te kunnen controleren is eerst een in Prolog executeerbare versie gemaakt. De consistentie en volledigheid van dit programma, en indirect van het conceptuele model, kan nu met een van de algoritmen uit het volgende hoofdstuk worden onderzocht.

Tot slot zij hier opgemerkt dat implementatie van het voorouder-probleem in een knowledgebase-management systeem [FEL-89-A267] eveneens makkelijk is te verwezenlijken. Als voorbeeld is POSTGRES genomen [Stonebraker88]. De structuur van het Prolog-programma wordt duidelijk weerspiegeld in dat van het POSTQUEL-equivalent dat in figuur 2.4 is opgenomen.

```
range of O is OUDER
range of VO is VOOROUDER
define view VOOROUDER(O.all)
define view *
      VOOROUDER(VO.ouder_van,O.afstamming_van)
      where VO.afstamming_van = O.ouder_van
```

Figuur 2.4: Voorouder-probleem in POSTQUEL.

De twee tabellen (relaties) die hier worden gebruikt, t.w. OUDER en VOOROUDER, bevatten elk twee attributen, t.w. Ouder_van en Afstamming_van. De eerste define view maakt een kopie van alle bekende OUDER-feiten in de VOOROUDER-tabel. In de tweede (recursieve) define view, te herkennen aan de achtergevoegde asterisk, worden de indirecte feiten afgeleid.

3 INTEGRITEIT VAN DE KNOWLEDGEBASE

3.1 Inleiding

Elk van de in dit hoofdstuk opgenomen paragrafen bevat de beschrijving van een algoritme waarmee de integriteit van een kennisbank in zekere zin kan worden gecontroleerd. De toevoeging 'in zekere zin' heeft hierbij betrekking op het feit dat er tamelijk grote verschillen zijn tussen een aantal van de algoritmen wat de erin uitgedrukte *visie* op integriteit betreft. Grofweg kunnen de verschillende visies beschreven worden als:

- Deductieve database-optiek (Decker, Sadri),
- Programmeertaal-optiek (Shapiro) en
- Produktieregel-optiek (Suwa, Nguyen).

Het algoritme van Bezem valt bij deze classificatie tussen wal en schip: Wat de kennisbank betreft gaat Bezem uit van een (formalisatie van) produktieregel-systemen. Het algoritme lijkt echter meer op de algoritmen van Decker en van Sadri dan op die van Suwa en Nguyen.

Het algoritme van de laatstgenoemde kon niet uit de in artikelen gegeven beschrijvingen worden gedestilleerd. Voor de anderen is dit wel gelukt. Om de leesbaarheid van een ander te bevorderen gaat het om meer dan pure beschrijvingen. Accenten en eigen interpretaties zijn toegevoegd om een afweging van merites te vergemakkelijken. Vergelijkende beschouwingen over algoritmen alsmede computationele efficiëntie en toepasbaarheid komen echter pas aan de orde in het volgende hoofdstuk.

3.2 Algoritme van Decker

Het algoritme dat beschreven is in [Decker86] is bedoeld om de integriteit van een deductieve database te bewaken. Een deductieve database bevat drie soorten gegevens.

Naast de feiten en de beperkingsregels die ook in 'gewone' databases voorkomen zijn tevens afleidingsregels opgenomen. Met deze regels kunnen uit de expliciet opgenomen feiten (de extensie van de database) andere worden afgeleid. Deze afleidbare feiten zijn dus slechts impliciet aanwezig en worden alleen afgeleid als een database-query daartoe aanleiding geeft. Door de aanwezigheid van afleidingsregels kan een deductieve database als *kennisbank* worden betiteld. Het is als zodanig een alternatief voor produktieregelsystemen. De integriteit van een deductieve database is de integriteit van de 'gewone' database die door uitputtende toepassing van de afleidingsregels wordt verkregen. Terwijl de beperkingsregels bij een gewoon database-systeem beperkingen opleggen aan de verzameling van feiten die (expliciet) in de database is opgenomen, hebben de beperkingsregels van een deductief database-systeem dus betrekking op de totale verzameling van (expliciete dan wel afleidbare) feiten. Zowel bij databases als bij deductieve databases spreekt men van *integriteitsbewaking*, hetgeen uitdrukt dat het met name gaat om het in stand houden van de integriteit bij wijzigingen in de data-respectievelijk kennisbank. De algoritmes voor integriteitsbewaking zijn er dan ook op toegespitst om de controle van de integriteit na een (summiere) wijziging van het bestand op efficiënte wijze uit te voeren. Men maakt hiertoe, zowel bij databases als bij deductieve databases, gebruik van de aanname dat het bestand vóór de update aan de beperkingsregels voldeed. Met behulp van deze aanname kunnen beperkingsregels compile-time vereenvoudigd worden. Vervolgens kan bij een update, op basis van inhoud en aard van de update, bepaald worden welke van deze vereenvoudigde beperkingsregels dient/dienen te worden geëvalueerd. Hoe een en ander in zijn werk gaat zal aan de hand van voorbeelden aan het eind van deze paragraaf worden toegelicht.

Bij deductieve databases zijn zowel queries als afleidingsregels in het algemeen Horn-clauses. Voor beperkingsregels zijn Horn-clauses onvoldoende expressief. In plaats hiervan gebruikt men ruimere subklassen van de eerste orde logica zoals 'allowed formulas' [Topor86] of 'range-restricted formulas' [Nicolas82, Decker86]. Deductieve databases zijn tot nu toe voornamelijk onderwerp van theoretisch onderzoek geweest, waarbij men over het algemeen van interne Prolog-databases gebruik heeft gemaakt. In [Decker86] wordt veel aandacht besteed aan het werken met range-restricted formules, met name het controleren of een eerste-orde formule range-restricted is en het omschrijven van range-restricted formules in een kanonieke vorm, de zogenaamde 'range form'. Deze standaardvorm biedt de mogelijkheid om een eenvoudige interpreter te

schrijven in Prolog die alle range-restricted formules kan evalueren. Decker's verhandeling over range-restricted formules is moeilijk te volgen voor wie niet enigszins thuis is in logica en logic programming. Om deze reden wordt hier niet te diep ingegaan op de achtergronden en argumentaties. Onderwerpen die wel aan de orde komen zijn: de definitie van range-restricted, de verschillende gedaanten waarin 'range-forms' kunnen voorkomen en het simplificatie-algoritme dat de efficiënte(re) integriteitsbewaking mogelijk maakt. Zowel de definitie van range-restricted als het simplificatie- algoritme zijn generalisaties van de aanpak van integriteitscontrole bij gewone databases, zoals voorgesteld in [Nicolas82].

Definitie:

Zij $F=QM$ een formule in disjunctieve minimaal-vorm, een disjunctie van conjuncties dus, waarbij Q een opeenvolging van kwantificaties ($\forall x \exists y \forall z \dots$ etc) is en M een kwantorvrije formule.

1. Als X een existentieel gekwantificeerde variabele in F is (m.a.w. $\exists$ komt in Q voor) dan is F range-restricted m.b.t. X als iedere conjunctie van M waar X in voorkomt tenminste 1 positieve X -literal bevat.
2. Als X universeel gekwantificeerd is in F (m.a.w. $\forall$ komt in Q voor) dan is F range-restricted m.b.t. X als de disjunctieve minimaalvorm van $\neg F$ range-restricted is m.b.t. X .
3. F is range-restricted als F range-restricted is m.b.t. alle variabelen in Q .

(Een X literal is een atoom dat X bevat, bijv. $p(X)$ of $r(X,Y)$, of de negatie hiervan).

De essentie van deze wellicht moeilijk te doorgronden definitie moge duidelijk worden aan de hand van het onderstaande voorbeeld.

Voorbeeld:

Stel dat men als beperkingsregel in een database wil opnemen dat men bij defensie burgerpersoneel (b) en militair personeel (m) kent, maar geen personeel dat onder beide noemers valt. In eerste-orde logica kan deze beperkingsregel worden geformuleerd als

$$\forall X((b(X) \vee m(X)) \wedge \neg (b(X) \wedge m(X))) \quad (1)$$

Deze formule is echter niet range-restricted. Een disjunctieve minimaalvorm ervan is

$$F = \forall X(b(X) \wedge \neg m(X) \vee (m(X) \wedge \neg b(X))) \quad (2)$$

In de definitie van range-restricted heeft deze formule betrekking op geval (2) en F is gelijkwaardig met

$$\neg \exists X(b(X) \wedge m(X) \vee \neg b(X) \wedge \neg m(X)) \quad (3)$$

De formule is niet range-restricted omdat de conjunctie $\neg b(X) \wedge \neg m(X)$ geen positieve X-literal bevat. Als men zou pogen deze formule in Prolog te representeren en daarbij uit zou komen op,

$$\text{not} ((b(X), m(X)) ; (\text{not } b(X), \text{not } m(X))) \quad (4)$$

dan loopt de integriteitscontrole in onderstaande database spaak. Evaluatie van (4) levert het resultaat 'yes' op, terwijl van Joep niet gespecificeerd is of hij tot het burger- dan wel militair personeel behoort.

```

functie(jaap, fourier) .
functie(joop, technicus) .
functie(joep, chauffeur) .
militair(jaap) .
burger(joop) .

```

Figuur 3.1: Database behorende bij het voorbeeld.

Dat de integriteitscontrole mis loopt heeft te maken met de semantiek van de not-operator in Prolog, met andere woorden de negation-as-failure regel. De Prolog-expressie (4) is in feite niet de vertaling van de eerste-orde formule (3) maar van:

$$\neg [\exists X (b(X) \wedge m(X)) \vee (\neg \exists X b(X) \wedge \neg \exists X m(X))] \quad (5)$$

Om dergelijke subtiële vergissingen bij het gebruik van de not-operator in Prolog te vermijden kan men als stelregel hanteren dat alle variabelen in het argument van de not-operator geïnstantieerd moeten zijn (ground terms) voordat 'not' wordt geëxecuteerd. Dit is precies wat het begrip range-restricted garandeert.

Als door het inlassen van een predikaat p (van personeel) in plaats van (4):

$$\text{not}(p(X), ((b(X), m(X)); (\text{not } b(X), \text{not } m(X)))) \quad (6)$$

gebruikt wordt, verloopt de integriteitscontrole wel goed. Deze Prolog-expressie correspondeert met de range-restricted formule,

$$\forall X (p(X) \rightarrow (b(X) \wedge \neg m(X) \vee m(X) \wedge \neg b(X))) \quad (7)$$

$p(X)$ wordt hierbij de range-expressie genoemd.

Zeker geoefende Prolog-programmeurs zullen zich afvragen waarom het allemaal zo 'ingewikkeld' moet (via range-restricted formules) als het zo 'eenvoudig' kan (via ground expressies in Prolog). Het is waar dat men in feite om het begrip range-restricted heen

kan en bij gebruik van Quintus-prolog zelfs Decker's (hierna te bespreken) meta-interpreter voor beperkingsregel-evaluatie niet nodig heeft. Range-restricted beperkingsregels kunnen als gewone Prolog-goals worden opgenomen, mits men - zoals bijvoorbeeld in Quintus Prolog - in principe de beschikking heeft over *operatoren* voor *and*, *or* en *not*. Niettemin is het voor lees- en formuleergemak van beperkingsregels bevorderlijk als deze in predikaat- logische vorm kunnen worden geformuleerd. De omzetting van eerste-orde (range-restricted) expressies naar evalueerbare range forms of zelfs naar Prolog-goals kan in principe geautomatiseerd worden. Het begrip range-restricted brengt dus declaratief programmeren dichterbij en kan als zodanig bevorderlijk zijn voor de kwaliteit van kennissysteem-software.

Decker bewijst dat alle range-restricted formules kunnen worden omschreven naar de volgende standaardvorm(en):

1. Ofwel F bevat slechts existentieel gekwantificeerde variabelen (waarbij dus iedere conjunctie waar een variabele Y in voorkomt tenminste een positieve Y -literal bevat), of
2. F is van de vorm $\exists X(P \wedge F') \vee F''$, waarbij P een zogenaamde range-expressie in X is (een louter existentieel gekwantificeerde disjunctie van positieve X -literals) en F' en F'' range-restricted formules in standaardvorm, of
3. F is van de vorm $\forall X(P \rightarrow F') \wedge F''$, waarbij P wederom een range-expressie in X is en F' en F'' range restricted formules in standaardvorm.

Vervolgens merkt Decker op dat de problemen wat evaluatie van een range form in Prolog betreft beperkt zijn tot versie 3 van de range form. Als de interpreter range forms van de vorm $\forall X(P \rightarrow F')$ aankan kan zij alle range forms evalueren.

Immers, de existentiële kwantor is in Prolog clauses impliciet aanwezig, voor alle vrije variabelen. Door een hulppredikaat 'forall' te gebruiken (bijvoorbeeld met de gedaante $\text{forall}(X, X_{\text{rex}}, \text{Concl})$ als vertaling van $\forall X(X_{\text{rex}} \rightarrow \text{Concl})$) en een meta-interpreter te schrijven (zie figuur 3.2) die dit predikaat kan verwerken en alle in beperkingsregels voorkomende expressies die *géén* forall bevatten doorsluisst naar de gewone Prolog-interpreter, is het probleem geklaard.

```
evaluate(Constraint):- verify(Constraint).

verify(forall(X,Xrex,Concl) ):-
    call(Xrex),
    falsify(Concl),!,fail.
verify( forall(X,Xrex,Concl) ):- !.
verify(Constraint):- call(Constraint).
verify(Constraint):- !,fail.

falsify(Constraint):- verify(Constraint),!,fail.
falsify(Constraint).
```

Figuur 3.2: Prolog meta-interpreter.

Voor de duidelijkheid zij hier opgemerkt dat het eerste argument van het forall-predikaat hierbij ook weggelaten kan worden. Het is louter toegevoegd voor de leesbaarheid, niet voor de evaluatie. In feite vooronderstelt het algoritme dat Xrex een range-expressie in X is. Als dat niet het geval is moet de beperkingsregel anders worden geformuleerd.

Zoals reeds aangestipt bij het commentaar op het begrip range-restricted kan de meta-interpreter in feite vergeten worden als men in plaats van 'forall (X, Xrex, Concl)' (de Prolog-expressie 'not (Xrex, not Concl)' gebruikt.

De vereenvoudiging van beperkingsregels bij updates van *gewone* databases is tamelijk eenvoudig. Bij een INSERT substitueert men true in de beperkingsregel op die plaatsen waar het te inserten feit voorkomt. Bij een DELETE substitueert men false. Dit behoeft slechts eenmalig (*compile-time*) te gebeuren.

Voorbeeld:

Uit de beperkingsregel:

$$\begin{aligned} \forall X(\neg \text{personeel}(X) \vee \\ (\text{militair}(X) \wedge \neg \text{burger}(X)) \vee \\ (\text{burger}(X) \wedge \neg \text{militair}(X))) \end{aligned}$$

worden (compile-time) de volgende update-condities afgeleid voor de verschillende soorten updates:

```
INSERT personeel(X) ONLY-IF militair(X) ∧ ¬ burger(X) ∨
    burger(X) ∧ ¬ militair(X)
INSERT militair(X) ONLY-IF ¬ personeel(X) ∨ ¬ burger(X)
INSERT burger(X) ONLY-IF ¬ personeel(X) ∨ ¬ militair(X)
DELETE militair(X) ONLY-IF ¬ personeel(X) ∨ burger(X)
DELETE burger(X) ONLY-IF ¬ personeel(X) ∨ militair(X)
```

Wanneer nu een feitelijke update plaatsvindt, bijvoorbeeld INSERT burger(jan), hoeft dus alleen $\neg \text{personeel}(\text{jan}) \vee \neg \text{militair}(\text{jan})$ geëvalueerd te worden.

Bij integriteitscontrole in deductieve databases dient men niet alleen de update zelf maar ook de gevolgen van die update (in termen van *afleidbare feiten*) te bepalen om de beperkingsregels te kunnen localiseren die geëvalueerd moeten worden. In concreto moeten twee feitenverzamelingen worden berekend, namelijk de verzameling van feiten die na de update afleidbaar zijn, terwijl ze dat voor de update nog niet waren en de verzameling feiten die oorspronkelijk wel afleidbaar waren maar na de update niet meer zijn. Deze verzamelingen worden aangegeven met A+ en A-. In dit kader is het van belang te beseffen dat het toevoegen van een feit zowel aan A+ als aan A- kan bijdragen door de aanwezigheid van 'not' in afleidingsregels. Hetzelfde geldt voor het verwijderen van een feit.

Decker beschrijft eerst hoe men de sets A+ en A- recht-toe-recht-aan zou kunnen berekenen en presenteert vervolgens een efficiënte(re) procedure voor het bepalen van B+

en B- als schatters van A+ en A- met $A+ \subset B+$ en $A- \subset B-$. Decker laat aan de oplettendheid van de lezer over dat B+ en B- niet gelijk hoeven te aan A+ en A-.

De recht-toe-recht-aan-methode bestaat uit het rechtstreeks toepassen van de definities van A+ en A-. Laat DB de database voor update zijn en DBN na update, DB* de extensie van DB en DBN* die van DBN. Deze extensies zijn gedefinieerd als de vereniging van de expliciete feiten met de impliciete feiten (afleidbaar uit de afleidingsregels van de database). Als Head :- Body een afleidingsregel is dan levert set_of(Head, Body, Set) de set Set van alle instanties van Head die tot de extensie behoren en via de betreffende afleidingsregel kunnen worden afgeleid. Door al deze sets met de expliciete feiten te verenigen verkrijgt men de gehele extensie. Vervolgens kunnen A+ en A- berekend worden uit DB* en DBN* via $A+ = DBN^* \setminus DB^*$ en $A- = DB^* \setminus DBN^*$. Het op deze wijze berekenen van A+ en A- is uiteraard weinig zinvol. Het is dermate tijdsintensief dat de simplificatie van beperkingsregels netto tijdverlies i.p.v. tijdwinst zou opleveren. De efficiëntere methode die Decker aandraagt komt neer op de introductie van een gokelement en het inruilen van ruimte- tegen tijd-complexiteit: van iedere afleidingsregel in de database wordt bijgehouden welke literals kunnen bijdragen tot de conclusie ervan, in positieve of in negatieve zin. Figuur 3.3 geeft een voorbeeld.

```
p(X) :- q(X), r(X), not s(X)

occurs_positive(q(X), (p(X) :- q(X), r(X), not s(X)) ).
occurs_positive(r(X), (p(X) :- q(X), r(X), not s(X)) ).
occurs_negative(s(X), (p(X) :- q(X), r(X), not s(X)) ).
```

Figuur 3.3: Transformatie van een Prolog-regel volgens Decker.

Nu is het niet meer nodig van *alle* afleidingsregels in DB en DBN de extensie te bepalen. In plaats daarvan worden slechts de extensies van de afleidingsregels in de update bepaald en deze leveren de eerste bijdragen aan B+ en B- (tezamen met de feiten in de update).

Vervolgens worden aan de hand van de in 'occurs'-feiten vastgelegde meta-informatie over de regels de transitieve afsluitingen van B+ en B- bepaald. Van ieder feit in B+ en B- wordt daartoe nagegaan of het unificeerbaar is met een literal L die als eerste argument in een occurs-feit voorkomt. Als dit zo is wordt van de corresponderende instantiatie van het tweede argument (het *wellicht* afleidbare feit) gecontroleerd of het *daadwerkelijk* in A+ respectievelijk A- thuishoort, door het als goal te evalueren tegen DB* en DBN.

Is dat het geval dan wordt de extensie van de hiermee corresponderende instantiatie van het tweede argument aan B+ of B- toegevoegd (als een feit uit B+ unificeerbaar is met een literal in een occurs-negative-feit aan B-, als een B- feit met een occurs-positive-literal unificeert aan B- ... etc.). In feite wordt na iedere additie aan B+ of B- meteen gekeken of het toegevoegde element unificeerbaar is met een literal in een update constraint. Als dat het geval is wordt meteen de corresponderende (eventueel geïntantieerde) conditie gecontroleerd. Dit 'heen en weer springen' (Eng: interleaving) tussen expansie van B+ en B- en de feitelijke integriteitscontrole verhoogt de kans op snel ontdekken van een overtreding tegen de integriteit.

Voorbeeld:

Beschouw de deductieve database, bestaande uit:

- Feiten: $p(a)$, $q(a)$, $p(b)$, $q(c)$.
- Afleidingsregels:
 $r(X) :- p(X), \text{not } q(X)$.
 $s(X) :- q(X), \text{not } p(X)$.
 $t(X) :- q(X), \text{not } r(X)$.
- Beperkingsregels:
 $\text{forall}(X, t(X), \text{not } r(X))$.
 $\text{forall}(X, s(X), \text{not } r(X))$.

Naast de expliciete feiten bestaat de extensie dus uit:

$r(b). s(c). t(a). t(c).$

- **Occurs-feiten zijn:**

$\text{occurs_pos}(p(X), (r(X):- p(X), \text{not } q(X))) .$
 $\text{occurs_pos}(q(X), (s(X):- q(X), \text{not } p(X))) .$
 $\text{occurs_pos}(q(X), (t(X):- q(X), \text{not } r(X))) .$
 $\text{occurs_neg}(q(X), (r(X):- p(X), \text{not } q(X))) .$
 $\text{occurs_neg}(b(X), (s(X):- q(X), \text{not } p(X))) .$
 $\text{occurs_neg}(r(X), (t(X):- q(X), \text{not } r(X))) .$

- **Update-constraints zijn:**

$\text{INSERT } t(X) \text{ ONLY-IF not } r(X).$
 $\text{INSERT } s(X) \text{ ONLY-IF not } r(X).$
 $\text{INSERT } r(X) \text{ ONLY-IF not } t(X), \text{ not } s(X).$

De integriteits-controle voor de update $U = \{C_1, C_2, C_3\}$ met:

$C_1 = \text{INSERT } u(a).$
 $C_2 = \text{DELETE } q(a).$
 $C_3 = \text{INSERT } q(X):- p(X), u(X).$

verloopt nu als volgt:

1. **Voor C_1 :**

- a. Bepaal de uit C_1 in DBN afleidbare feiten-updates. $B+ = \{ u(a) \} ; B- = \{ \} .$
- b. Ga na welke elementen van $B+$ en $B-$ unificeren met literals in update-constraints: geen (Er is geen 'INSERT $u(X) \dots$ ' - constraint en ook geen 'DELETE $u(X) \dots$ ' - constraint.
- c. Ga na welke eerste argumenten van occurs-feiten unificeren met $u(a)$: geen.
Resultaat: $B+ = \{ \} \text{ en } B- = \{ \} .$

2. **Voor C_2 :**

- a. $B+ = \{ \} ; B- = \{ q(a) \} .$
- b. geen.

- c. Drie van de zes occurs-feiten hebben $q(X)$ als eerste argument. Dit levert op:
- $$B+ = \text{ext}\{ r(a) :- p(a), \text{not } q(a) \};$$
- $$B- = \text{ext}\{ (s(a) :- q(a), \text{not } p(a)), \\ (t(a) :- q(a), \text{not } r(a)) \}.$$
- De extensie van DBN is
- $$\{p(a), p(b), q(c), u(a), q(a), r(b), t(a)\}, \text{ dus}$$
- $$B+ = \{ \} \text{ en } B- = \{ t(a) \}$$
- d. Ga na welke van de huidige elementen in $B+$ en $B-$ unificeren met literals in update-constraints (zie stap 1b); Dit levert geen treffer: Er is geen DELETE $t(a)$ - constraint
- e. Herhaal stap c met de huidige $B+$ en $B-$.
Resultaat: $B+ = \{ \}$ en $B- = \{ \}$.
3. Voor C_3 :
- a. $B+ = \{ q(a) \}; B- = \{ \}$.
- b. geen.
- c. $B+ = \text{ext}\{ r(a) :- p(A), \text{not } q(A) \};$
 $B- = \text{ext}\{ (s(a) :- q(a), \text{not } p(a)), \\ (t(a) :- q(a), \text{not } r(a)) \}.$
- d. geen treffers: INSERT $t(a)$... is niet aanwezig
- e. Herhaal stap c met de huidige $B+$ en $B-$.
Resultaat: $B+ = \{ \}$ en $B- = \{ \}$.

De update $\{C_1, C_2, C_3\}$ wordt dus geaccepteerd.

3.3 Algoritme van Sadri en Kowalski

Sadri en Kowalski [Sadri88] beschrijven een algoritme voor integriteitscontrole in deductieve databases dat in zekere zin een generalisatie is van dat van [Decker86]. Om precies te zijn kan het algoritme van Decker en ook dat van anderen als [Lloyd86] benaderd worden door een bepaalde literal-selectie- methode te kiezen. Dit geeft al

enigszins aan dat in [Sadri88] eigenlijk een raamwerk voor een algoritme gegeven wordt in plaats van (slechts) één uitwerking.

Net als bij het algoritme van Decker wordt de consistentie (in dit verband het voldoen van de extensie van een deductieve database aan de geformuleerde beperkingsregels) gecontroleerd met gebruikmaking van theorem-proving methoden. De voornaamste verschillen met de methode van Decker zijn:

- De wijze van literal selection (conflict resolution in meer traditionele kennissysteem-termen) is min of meer vrij te kiezen.
- Er wordt consequent voorwaarts geredeneerd vanaf de geplande update in plaats van eerst voorwaarts vanaf de update en dan achterwaarts vanaf de relevante update-conditie(s).

In dit rapport wordt niet diep ingegaan op de finesses van het algoritme. Door de (nog) grotere flexibiliteit is het minder gemakkelijk compact te beschrijven dan het algoritme van Decker. De discussie beperkt zich tot het aangeven van de belangrijkste consequenties van de hierboven genoemde verschillen. Het eerstgenoemde verschil wordt door Sadri en Kowalski als een argument vóór hun methode gebruikt: Het biedt de mogelijkheid om een resolutie-algoritme te gebruiken dat sneller is dan de 'gewone' Prolog SLDNF-resolutie. Als concrete kandidaat wordt daarbij de connection-graph-methode genoemd [Kowalski75]. Het is echter niet onmiddellijk duidelijk hoeveel (tijd-)winst dit oplevert en het vergt in ieder geval extra inspanning voor het programmeren van de integriteitscontrole als we uitgaan van Prolog-databases. Het feit dat de connection-graph-methode in [Sadri88] slechts tussen neus en lippen genoemd wordt doet vermoeden dat we van de tijdwinst die het oplevert geen al te spectaculaire staaltjes moeten verwachten. Het tweede verschil maakt het onmogelijk om zonder meer Prolog-resolutie te gebruiken binnen het door Sadri en Kowalski geschetste raamwerk. Voorwaarts redeneren vanuit ontkenningen (denials) als 'not p(X)' vereist uitprogrammeren van de resolutie. Wanneer men dit in Prolog zou doen is het de vraag of dit niet te veel ten koste gaat van de evaluatie-snelheid.

3.4 Algoritme van Shapiro

In het voorgaande is al aan de orde gekomen dat het algoritme van Shapiro uitgaat van enkele veronderstellingen aangaande het formalisme waarop de debugging-algoritmen worden toegepast [Shapiro83]. Dit formalisme, i.c. een programma in een bepaalde programmeertaal, is op te vatten als de neerslag van de specificaties van een probleemdomen. Ten eerste is het basis-element van deze taal de procedure (of functie). Tijdens het debugging-proces zijn alleen de aanroepen van deze procedures van belang, samen met de in- en uitvoer. Ten tweede heeft elke procedure een naam, ariteit (het aantal argumenten) en bijbehorende code. Ten derde legt de programma-code een verzameling gedragingen vast. Ten vierde zijn alle 'berekeningen' van het programma, d.w.z. reeksen opeenvolgende procedure-aanroepen, te beschrijven met een boomstructuur die als knopen de aanroepen bevat en als bladen succes- of faalpunten.

Het algoritme van Shapiro is in eerste instantie bedoeld voor programma-debugging; het localiseren en corrigeren van fouten in de programma-code [Hamdan89]. Deze fouten kunnen verschillende oorzaken hebben, zoals een gebrekkig functioneel ontwerp of een tekortschietende systeemontwikkelingsmethode. Volgens Shapiro kan er echter een meer fundamentele oorzaak worden aangewezen. Hij ziet een programma als de belichaming van een complexe verzameling assumpties. Het gedrag van het programma is een afgeleide van deze assumpties, waardoor het moeilijk is om dit gedrag volledig voorspelbaar te maken. Deze overwegingen hebben Shapiro ertoe aangezet een theorie te ontwikkelen die een antwoord moet geven op twee vragen:

- Hoe kan een fout gedetecteerd worden in een programma dat niet correct werkt ?
- Hoe kan deze fout worden gecorrigeerd ?

De theorie heeft geleid tot de constructie van algoritmen voor elk van deze twee problemen. Met behulp van een diagnose-algoritme moeten fouten worden gevonden, waarna met een fout-correctie algoritme verbeteringen kunnen worden aangebracht. Deze twee algoritmen zijn samen opgenomen in een zogenaamd 'Model Inferentie Systeem' (MIS), dat als invoer een te debuggen programma heeft en een lijst met in- en uitvoer voorbeelden, die gedeeltelijk het gedrag van het programma bepalen.

Het is noodzakelijk dat tijdens het debugging-proces duidelijkheid bestaat omtrent het verwachte gedrag van het programma. Steeds wordt ervan uitgegaan dat er een 'instantie' is die de vragen kan beantwoorden welke door het systeem worden gesteld over de 'Universe of Discourse'. Deze antwoorden kunnen door de ontwikkelaar van het programma worden gegeven, of door een ander programma. Wanneer namelijk een goed werkende versie van een programma wordt gewijzigd, kunnen vragen gegenereerd tijdens het debuggen hiervan worden 'beantwoord' door de oude versie. Het is zelfs mogelijk om een (eenvoudig) programma van de grond af aan op te bouwen door met een 'leeg' programma te beginnen. Het MIS en de te debuggen programma's zijn geschreven in de taal Prolog.

De integriteitscontrole van knowledgebases heeft in eerste instantie betrekking op het detecteren van fouten in 'executeerbare specificaties'. Hier wordt ervan uitgegaan dat de specificatie van een kennisdomein is opgesteld met behulp van ENIAM. Het resulterende conceptuele model is om te zetten naar een formele, op een computer executeerbare representatie in de taal Prolog. In dit rapport gaat de aandacht in het bijzonder uit naar wat in traditioneel NIAM niet-grafische beperkingsregels werden genoemd. Deze regels vormen een belangrijk gedeelte van de kennis in een knowledgebase en zijn te vergelijken met produktieregels. Het diagnose-gedeelte van Shapiro's algoritme kan de integriteit van deze Prolog-code controleren.

Door toepassing van het algoritme van Shapiro op 'pure' Prolog programma's, zijn drie soorten fouten te detecteren; terminatie van een programma met onjuiste resultaten, terminatie met missende resultaten en niet-terminatie. De basis van het diagnose-algoritme wordt gevormd door een meta-interpreter, dat is een interpreter voor een programmeertaal geschreven in die taal zelf. De meta-interpreter stelt de gebruiker in staat het 'berekenningsproces' van een programma aan een nader onderzoek te onderwerpen. Het is aldus te vergelijken met een trace-faciliteit die bij talen als C, Pascal, Prolog en Lisp kunnen worden gebruikt. Een meta-interpreter is een meta-programma dat andere programma's als data kan beschouwen [Sterling86], waardoor manipulatie, analyse en simulatie hiervan eenvoudig wordt. Op dit punt aangekomen is het instructief om een voorbeeld te geven van een meta-interpreter voor 'puur' Prolog, d.w.z. een eenvoudige Prolog, o.a. zonder operatoren als 'cut' en 'not':

```
solve(true,true).
solve((A,B),(BewijsA,BewijsB)):-
    solve(A,BewijsA),
    solve(B,BewijsB).
solve(A,(A:-Bewijs)):-
    clause(A,B),
    solve(B,Bewijs).
```

Figuur 3.4: Meta-interpreter voor eenvoudig Prolog.

Deze meta-interpreter houdt een 'proof tree' bij van elk bewijs dat geleverd moet worden. In het kort is de werking als volgt. Het atoom 'true' is altijd waar (true). De conjunctie van twee goals is waar als ze beide waar zijn. Een goal die een body bevat met subgoals is waar als alle subgoals waar zijn. De laatste solve/2-regel van figuur 3.4 bevat het predikaat 'clause(A,B)'. Hiermee worden de head en body van een regel gescheiden; B bevat de conjunctie van subgoals van een goal A. Uitgaande van de regel 'A:- B₁, B₂, B₃, B₄.', heeft het tweede argument van solve/2 de vorm: (B₁, (B₂, (B₃, B₄))). Om een programma te kunnen debuggen moet de ontwikkelaar een idee hebben hoe het programma toegepast op een bepaald applicatiegebied, zich moet gedragen. Met behulp van een meta-interpreter kan nu dit programma gesimuleerd worden. Gebruikmakend van een debugging-algoritme wordt geprobeerd een verschil tussen het verwachte en werkelijke gedrag vast te stellen [Sterling86].

Alvorens een preciezer beschrijving van Shapiro's algoritmen te geven is het verhelderend enkele opmerkingen te maken over de *semantiek* van 'logic programs', i.e. Prolog [Shapiro84; Sterling87].

De clause A:- B₁, B₂, ..., B_n, omvat A' in de interpretatie *M* (lees *Meaning*) als er een substitutie θ is die A met A' unificeert en de goals B₁, B₂, ..., B_n in *M* waarmaakt. Een clause is 'waar' in *M* als elke variabele-vrije goal van deze clause in *M* zit, en anders 'onwaar'. Deze bewering is eenvoudig te spiegelen aan het gebruik van Prolog in de praktijk. In deze omgeving zijn feiten waar, wanneer ze zich in de interne database bevinden, of wanneer ze door toepassing van een regel kunnen worden afgeleid. Een

programma P in M is waar als elke clause in P waar is in M . De notie 'waar' kan in een logische context worden opgevat als 'correct', 'onwaar' als 'niet correct'.

De interpretatie (betekenis) van een programma P is $M(P)$, wat overeenkomt met de verzameling variabele-vrije (volledig geïnstantieerde) goals in P die waar zijn. Nu kan worden gezegd dat een programma P compleet is in M als $M \subseteq M(P)$. Een programma P is alleen dan compleet en correct in M als $M = M(P)$.

Het domein D van een programma is de verzameling variabele-vrije goals van P . Een programma P is *terminerend* op een domein D als er geen enkel goal A in D is die leidt tot een oneindige afleiding van goal A in P .

In het voorgaande zijn de drie soorten fouten die met behulp van Shapiro's algoritme kunnen worden gedetecteerd, nader omschreven. De basis is steeds de 'proof tree' die wordt opgebouwd tijdens uitvoering van het programma. Dit kan ook worden weergegeven door een 'top level trace' van een procedure (predikaat in Prolog) p met invoer x en uitvoer y . Deze trace bestaat uit een (mogelijk lege) verzameling tripletten van de vorm $\{ \langle p_1, x_1, y_1 \rangle, \langle p_2, x_2, y_2 \rangle, \dots, \langle p_n, x_n, y_n \rangle \}$. Als de verzameling leeg is dan kan procedure p met invoer x de uitvoer y opleveren, zonder procedureaanroepen. Als de verzameling niet leeg is dan levert een keten van procedureaanroepen met aanvankelijke invoer x uiteindelijk een uitvoer y op.

Een 'proof tree' is *compleet* als alle knooppunten tripletten zijn van de vorm $\langle p, x, y \rangle$ en alle bladen als 'top level trace' de lege verzameling hebben. Er kan worden gesteld dat y de *correcte* uitvoer van de procedure-aanroep $\langle p, x \rangle$ in M is, als $\langle p, x, y \rangle$ zich in M bevindt.

Een programma P is *correct* in M als de wortel van elke complete 'proof tree' van P zich bevindt in M . Een programma P is *compleet* in M als elk triplet in M de wortel is van een complete 'proof tree' van P . Een programma P is *terminerend* als de 'proof tree' geen oneindig pad bevat.

Nu volgt een voorbeeld van een algoritme voor het detecteren van een onjuiste oplossing, gebaseerd op [Shapiro84; Sterling87]. Een programma P kan alleen een onjuiste oplossing

geven als het een onjuiste clause bevat. Een clause C is niet correct binnen de interpretatie M als er een instantiatie is van C 's body die waar is in M en een head die onwaar is in M . De beschrijving luidt:

- Invoer: procedure p in P en een invoer x , zodanig dat $\langle p, x \rangle$ een uitvoer y geeft die niet correct is in M .
- Uitvoer: een triplet $\langle q, u, v \rangle$ niet in M .
- Algoritme: simuleer de executie van p op invoer x met als resultaat y ; als een aanroep $\langle q, u \rangle$ leidt tot de uitvoer v , stel dan vast (door de gebruiker een vraag te stellen) of $\langle q, u, v \rangle$ zich in M bevindt. Als dit niet zo is geef dan als resultaat de onjuiste oplossing $\langle q, u, v \rangle$ en stop.

Het is moeilijk om te bepalen of een programma niet termineert, d.w.z. oneindig 'doorloopt'. Shapiro kiest voor een pragmatische oplossing van dit probleem door in de meta-interpreter (solve/3) van figuur 3.6 een vooraf bepaalde grens aan de recursie-diepte mee te geven. Als het programma termineert voordat de diepte bereikt is, bevat het derde argument de waarde 'geen_overflow'. Als de maximale diepte wordt bereikt zonder dat het programma een oplossing heeft gevonden, dan vindt terminatie plaats. Het derde argument van solve/3 bevat dan een 'proof tree' van procedure-aanroepen tot op het moment van termineren. Aan de hand van deze trace kan een nadere diagnose van het probleem plaatsvinden. Naast fouten in het programma kan de oorzaak bijvoorbeeld ook liggen in te weinig geheugen-capaciteit, waardoor 'stack overflow' optreedt. De beschrijving van het algoritme voor het detecteren van niet-terminatie luidt als volgt [Shapiro84;Sterling87]:

- Invoer: procedure p in P , een invoer x en een integer $d > 0$, zodanig dat de aanroep $\langle p, x \rangle$ de maximale recursiediepte d niet overschrijdt.
- Uitvoer: bij overschrijding van d bevat y de verzameling procedure-aanroepen ter lengte d , bestaande uit tripletten van de vorm $\langle p, x, y \rangle$.
- Algoritme: simuleer de executie van p op x ; als de diepte d bereikt wordt zonder dat een oplossing is gevonden, dan termineert het programma.

```
onjuiste_oplossing(A,Clause):-
    solve(A,Bewijs),
    false_goal(Bewijs,Clause).

false_goal((A:-B),Clause):-
    false_conjunctie(B,Clause),!.
false_goal((A:-B),(A:-B1)):-
    extract_body(B,B1).

false_conjunctie(((A:-B),Bs),Clause):-
    query_goal(A,false),!,
    false_goal((A:-B),Clause).
false_conjunctie((A:-B),Clause):-
    query_goal(A,false),!,
    false_goal((A:-B),Clause).
false_conjunctie((A,As),Clause):-
    false_conjunctie(As,Clause).

extract_body(true,true).
extract_body((A:-B),A).
extract_body(((A:-B,Bs),(A,As)):-
    extract_body(Bs,As).

query_goal(A,true):-
    system(A).
query_goal(Goal,Antwoord):-
    not system(Goal),
    writeln(['Is ',Goal,' waar ?']),
    read(Antwoord).
```

Figuur 3.5: Algoritme voor het detecteren van een onjuiste oplossing.

```
solve(true,D,geen_overflow).
solve(A,0,overflow([])).
solve((A,B),D,Overflow):-
    D > 0,
    solve(A,D,OverflowA),
    solve_conjunctie(OverflowA,B,D,Overflow).
solve(A,D,Overflow):-
    D > 0,
    clause(A,B),
    D1 is D - 1,
    solve(B,D1,OverflowB),
    return_overflow(OverflowB,A,Overflow).
solve(A,D,geen_overflow):-
    D > 0,
    system(A),
    A.

solve_conjunctie(overflow(S),B,D,overflow(S)).
solve_conjunctie(geen_overflow,B,D,Overflow):-
    solve(B,D,Overflow).

return_overflow(geen_overflow,A,geen_overflow).
return_overflow(overflow(S),A,overflow([A|S])).
```

Figuur 3.6: Algoritme voor het detecteren van non-terminatie.

Van de drie soorten fouten die het algoritme van Shapiro kan ontdekken is het vaststellen of er een missende oplossing het minst eenvoudig op te lossen. Een belangrijke rol speelt het begrip 'cover'. Een bepaalde clause C is een cover van een goal A m.b.t. een interpretatie M , als er een instantiatie bestaat van C waarvan de head overeenkomt met A en de body zich in M bevindt. Als een programma P een missende oplossing heeft in een interpretatie M , dan is er een goal A in M die geen cover heeft in een clause van P .

Tijdens de werking van onderstaand algoritme moet de gebruiker vragen beantwoorden omtrent de 'geldigheid' van bepaalde instantiaties.

```
missende_oplossing((A,B),Goal):-
    !,
    ( not A, missende_oplossing(A,Goal);
      A, missende_oplossing(B,Goal) ).
missende_oplossing(A,Goal):-
    clause(A,B),
    vraag_clause((A:-B)),
    !,
    missende_oplossing(B,Goal).
missende_oplossing(A,A):-
    not system(A).

vraag_clause(Clause):-
    writeln(['Geef, zo mogelijk, een correct voorbeeld
van',Clause,' en anders "nee".']),
    read(Antwoord),
    !,
    check_antwoord(Antwoord,Clause).

check_antwoord(neo,Clause):- !,fail.
check_antwoord(Clause,Clause):- !.
check_antwoord(Antwoord,Clause):-
    write('Verkeerd !'),
    !,
    vraag_clause(Clause).
```

Figuur 3.7: Algoritme voor het detecteren van een missende oplossing.

Een programma P is compleet in M als voor ieder triplet $\langle p, x, y \rangle$ in M de aanroep $\langle p, x \rangle$ de uitvoer y oplevert. D.w.z. p moet een cover zijn voor $\langle p, x, y \rangle$, zo niet dan is er

sprake van een missende oplossing. De remedie is dan het wijzigen van de procedure p , zodanig dat y als resultaat kan worden opgeleverd. Het algoritme voor het detecteren van een missende oplossing luidt als volgt [Shapiro84; Sterling87]:

- Invoer: procedure p in P , een invoer x en een uitvoer y allen in M , waarop p faalt.
- Uitvoer: een triplet $\langle q, u, v \rangle$ in M , waarvoor q geen cover is.
- Algoritme: simuleer de executie van p op x met als resultaat y , gebruikmakend van existentiële queries en onderwijl een 'proof tree' bijhoudend; in geval van een 'fail' wordt het triplet $\langle q, u, v \rangle$ opgeleverd en het algoritme stopt.

3.5 Algoritme van Suwa, Scott en Shortliffe

Suwa, Scott en Shortliffe geven in [Suwa82] een beschrijving van integriteitscontrole van kennisbanken in termen van 'consistency and completeness'. Eigenlijk is de vertaling naar 'consistentie en volledigheid' enigszins misleidend. Consistency verwijst niet naar consistentie in de zin van de logische modeltheorie (Eng.: soundness), maar naar (een inkleding van) het begrip correctheid. Aangezien ook deze term correctheid niet geheel op zijn plaats is, zal in het vervolg niettemin de term consistentie gehanteerd worden. Consistentie bij Suwa komt in concreto neer op het detecteren van drie soorten 'potentiële fouten' in de kennisbank: tegenstrijdigheid, subsumptie en redundantie. In [Suwa82] worden hiervoor de volgende definities gehanteerd.

- Tegenstrijdigheid: twee regels vuren in dezelfde situatie met tegenstrijdig resultaat.
- Redundantie: twee regels vuren in dezelfde situatie met hetzelfde resultaat.
- Subsumptie: een regel vuurt in alle gevallen dat een andere vuurt met hetzelfde resultaat, maar niet andersom (doordat laatstgenoemde regel extra condities bevat).
- Volledigheid: alle situaties worden door tenminste één regel afgedekt ('ontbrekende regels ontbreken').

De context van Suwa's integriteitscontrole is de ONCOCIN-rule base, welke regels bevat

van de volgende gedaante:

REGEL <regelnummer>

context: alle geneesmiddelen in klasse

actie-parameter: de huidige voor te schrijven dosis

conditie 1: parameter X heeft waarde

conditie 2: parameter Y heeft waarde

actie: De huidige voor te schrijven dosis bedraagt 75% van de laatst voorgeschrevene.

Naast een dergelijke regel-syntaxis wordt uitgegaan van enkelwaardige variabelen. Het algoritme dat zorgdraagt voor de op deze wijze gespecificeerde integriteitscontrole wordt in [Suwa82] *globaal* beschreven als volgt:

1. Maak een tabel waarin de waarde van een actieparameter wordt geassocieerd met de tupels van conditieparameter-waarden die tot eerstgenoemde waarde aanleiding geven.
2. Controleer aan de hand van deze tabel of aan de integriteitsbeperkingen voldaan is en geef een overzicht van de overtredingen als dit niet het geval is.

In [Suwa82] wordt geen gedetailleerde beschrijving gegeven van het algoritme, waarmee de stappen 1 en 2 worden verwezenlijkt.

3.6 Algoritme van Puuronen

De tweede stap die aan het einde van de vorige paragraaf is beschreven wordt in [Puuronen87] wél omgezet in een algoritme. Puuronen gaat uit van een regelbestand zonder ketens. Alle actie-parameters corresponderen met *eind*-conclusies en geen enkele conditie-parameter is dus actie-parameter in een andere regel. In geval van een dergelijk 'plat' regelbestand is stap 1 triviaal. Puuronen beperkt zich dan ook tot stap 2 wat de detaillering betreft. Overigens wordt noch uit [Puuronen87] noch uit [Suwa82] duidelijk of Suwa dezelfde beperkingen heeft gehanteerd, maar een opmerking in [Nguyen87b]

suggereert dat dit in *essentie* inderdaad het geval is: Suwa's algoritme onderzoekt slechts 'platte delen' van een kennisbank. Het algoritme van Puuronen komt grosso modo op het volgende neer: Een parameterwaarden-tupel is de compacte representatie van een conjunctie van "attribuut = waarde"-statements. $A = 1 \wedge B = 0 \wedge C = 0$ correspondeert bijvoorbeeld met het tupel (1,0,0). Wanneer we met C_i de verzameling van parameterwaarde-tupels aanduiden, waarbij de regel R_i vuurt (met resultaat A_i) en met Ω de verzameling van alle mogelijke parameterwaarde-tupels, dan kunnen tegenstrijdigheid, redundantie, subsumptie en volledigheid als volgt formeel worden gedefinieerd:

Voor zekere i, j met $i \neq j$:

- Tegenstrijdigheid: $A_i \leftrightarrow \text{niet } A_j \text{ en } C_i \cap C_j \neq \emptyset$
- Redundantie: $A_i = A_j \text{ en } C_i \cap C_j \neq \emptyset$
- Subsumptie: $A_i = A_j \text{ en } C_i \subset C_j \text{ terwijl } C_i \neq C_j$
- Onvolledigheid: $\cup_i C_i \neq \Omega$

Aangezien Puuronen uitgaat van enkelwaardige parameters geldt $A_i \leftrightarrow \text{niet } A_j$ als A_i en A_j verschillende waarden toekennen aan dezelfde parameter. In [Puuronen87] wordt vrij veel nadruk gelegd op de efficiënte implementatie van de verzamelingstheoretische bewerkingen (vereniging, doorsnijding) die in deze definities voorkomen. Een conditie C wordt gerepresenteerd middels zijn kardinaalgetallen. Ieder kardinaalgetal representeert daarbij één conjunctie van parameter-waarde-paren. Zo zou de conditie $X\text{-ja}$ en $Y\text{-nee}$ door één kardinaalgetal gerepresenteerd worden, de conditie $X\text{-ja}$ door twee (namelijk een corresponderend met ' $X\text{-ja}$ en $Y\text{-ja}$ ' en de tweede met ' $X\text{-ja}$ en $Y\text{-nee}$ ').

Wanneer het aantal legale parameterwaarde-tupels klein is, dan is de verzameling van bijbehorende kardinaalgetallen dat ook. Een set van kardinaalgetallen (C_i en C_j in de bovenstaande definitie) kan in het geval van een voldoende kleine parameterwaarde-ruimte worden gerepresenteerd in één machine-woord, zodat een verzamelingstheoretische voorwaarde als $\cup_i C_i \neq \Omega$ efficiënt naar machine-instructies vertaald kan worden.

De berekening van de afzonderlijke kardinaalgetallen komt neer op de vertaling van conjuncties naar gecomprimeerde bitpatronen. Wanneer bijvoorbeeld X een tweewaardige

(ja/nee) variabele is en Y een driewaardige (bijvoorbeeld hoog/normaal/laag) en we 'nee' associëren met 0, 'ja' met 1, 'laag' met 0, 'normaal' met 1 en 'hoog' met 2, dan kan als kardinaalgetal van de conditie $X=ja$ en $Y=normaal$ worden gekozen voor $[ja]$ maal $Card(Y) + [normaal]$, dus 1 maal 3 + 1 = 4. Hierbij staat $Card(Y)$ voor de kardinaliteit (het aantal mogelijke waarden) van de variabele Y. De keuze die in dit geval is gemaakt is de minst significante 'bits' van het kardinaalgetal te gebruiken voor de codering van Y. Het kan ook andersom, mits men maar eenzelfde 'rangorde' van variabelen hanteert bij alle condities.

3.7 Algoritme van Nguyen

Nguyen heeft, voortbordurend op de in [Suwa82] geïntroduceerde concepten, een tweetal algoritmes ontwikkeld om consistentie en volledigheid te controleren. Het CHECK-algoritme [Nguyen85, Nguyen87b] staat het dichtst bij de aanpak van Suwa. Naast tegenstrijdigheid, redundantie en subsumptie wordt door CHECK onder de noemer consistentie tevens gespeurd naar onnodige condities en circulaire regelketens. Een voorbeeld van onnodige condities:

De regels

IF $p = 1$ AND $q \neq 1$ THEN A en
IF $p \neq 1$ AND $q \neq 1$ THEN A

kunnen worden samengevoegd tot:

IF $q \neq 1$ THEN $\bar{A}$.

Het feit dat er op circulaire regelketens wordt gecontroleerd geeft al aan dat CHECK niet alleen kijkt naar 'platte' (delen van) regelbestanden. Dit heeft ook zijn weerslag in de controle van de volledigheid. Naast niet door condities afgedekte parameterwaarden wordt ook gezocht naar onbereikbare conclusies, doodlopende condities en doodlopende goals.

Het CHECK-algoritme is geschikt om de middels een 'generieke expert-system shell' - in casu Lockheed Expert System [Laffey86] - gemaakte kennisbanken te diagnostiseren. Regels in LES zijn van de vorm IF <conjunctie van condities> THEN <conjunctie van conclusies>. Naast backward-chaining kan men in LES ook forward-chaining gebruiken. In dit rapport is de beschrijving beperkt tot tot backward-chaining, omdat er weinig verschil zit tussen de integriteitscontrole bij forward- en backward-chaining. Alleen de notie van onbereikbare conclusies is bij forward-chaining niet van betekenis.

Net als ONCOCIN gebruikt ook LES een soort context-mechanisme. In ieder geval wordt de integriteitscontrole door Nguyen slechts toegepast op bijeenhorende *delen* van het totale regelbestand van een applicatie, waardoor de computationele kosten van het algoritme flink gedrukt kunnen worden. Verder kunnen de LES-regels variabelen bevatten.

In [Nguyen85] wordt 'in an Algol-like notation' een beschrijving van het algoritme gegeven. Deze is echter dermate *onvolledig* dat het niet duidelijk wordt hoe het de consistentie-controle uitvoert laat staan dat men iets over efficiëntie kan zeggen: Declaraties van variabelen zijn niet opgenomen en naar het resultaat-type van de functie Compare_clauses (... , ...) moet men gissen. De functie zelf is niet uitgewerkt. Hetzelfde geldt voor de functie Transform in de toekenning 'matched_rule = Transform (i, k, clause-relations)'.

In [Nguyen87b] is textueel (dat wil zeggen niet 'Algol-achtig') beschreven hoe het algoritme in grote lijnen werkt. Een en ander is uiteraard gekoppeld aan de werkwijze van LES. De kernpunten zijn hieronder samengevat. Nguyen merkt op dat er in LES meerdere goals tegelijk 'actief' kunnen zijn, waarbij aan elke goal een of meer rule-sets gekoppeld is. Een rule-set bevat die produktieregels die kunnen bijdragen tot het verwezenlijken van de betreffende goals. Nguyen merkt op dat de rule-sets van de goals *afzonderlijk* getest kunnen worden. Dit is geen wezenlijke beperking voor het type produktieregelsystemen waarvoor het algoritme gebruikt kan worden. In geval van mogelijke ongewenste interactie/afhankelijkheid tussen rules in verschillende sets, worden die sets verenigd tot een nieuwe, grotere set.

De LES-context is als volgt: Een rule-set bestaat uit een aantal rules. Iedere rule bevat een IF-part en een THEN-part. Het IF-part bevat een of meer condities, het THEN-part een of meer conclusies. Hoe de afzonderlijke conclusies en condities er precies uitzien wordt niet duidelijk (aannemende dat 'IF ?X has a fever, AND ?X has flat pink spots on his skin THEN type of disease of ?X is measles' een 'natuurlijke taal'-weergave is van 'IF symptom(?X, fever) AND symptom(?X, flat_pink_spots_on_skin) THEN disease_type(?X, measles)' of iets dergelijk. In deze laatste vorm bevat het IF-part dus twee condities, te weten 'symptom (?X, fever)' en 'symptom (?X, flat-pink-spots-on-skin)'. Nguyen [1987b] schetst nu de volgende procedure: Eerst worden alle afzonderlijke condities en conclusies (paarsgewijs) vergeleken. Het resultaat is een tabel met voor ieder paar n van de betitelingen SAME, DIFFERENT, CONFLICT, SUBSET of SUPERSET. Uit deze clause-clause relaties (Nguyen spreekt nu eens over condities/conclusies, dan weer over IF-clauses/THEN-clauses) worden vervolgens de relaties tussen de IF- en THEN-parts van de verschillende rules gedestilleerd. Tenslotte worden deze gebruikt om bij paren rules aan te geven of ze conflicterend, subsumerend, redundant (... etc.) zijn.

Naast het CHECK-algoritme beschrijft Nguyen het ARC-algoritme. ARC staat voor ART-rule-checker [Nguyen87a]. ARC biedt (volgens Nguyen) additionele mogelijkheden in de vorm van detectie van redundante regelketens, gesubsumeerde regelketens en conflicterende regelketens. Nguyen beschrijft niet hoe deze uitbreidingen zijn gerealiseerd. Zoals reeds eerder opgemerkt (in andere bewoordingen) zouden de drie bestudeerde artikelen [Nguyen85, Nguyen87a, Nguyen87b] zich bijzonder goed lenen voor een 'text-checker' in de geest van Nguyen zelf. Ze bevatten veel inconsistenties en - vooral - onvolledigheden. Waarschijnlijk is het van de grond af construeren van een rule-checker op basis van bovenstaande informatie, de beschrijving van Puuronen's algoritme en eigen ideeën veel effectiever dan een poging tot reconstructie van het algoritme van Nguyen zelf. Voor een dergelijke onderneming ontbreekt echter de tijd. Derhalve wordt dit verhaal over consistentie en volledigheid à la Nguyen besloten met het uitspreken en beargumenteren van een *vermoeden* omtrent de beperkte controle-mogelijkheden die de algoritmes bieden.

Vermoeden: ARC detecteert alleen flagrante fouten. De bewering dat ARC in staat is om conflicterende en redundante regelketens te detecteren is misleidend. Uit [Nguyen87a] valt te destilleren dat ARC dit alleen kan als de beginregels van de twee ketens dezelfde

of equivalente IF-parts hebben. Twee IF-parts zijn equivalent als ze hetzelfde aantal condities bevatten en deze condities paargewijs equivalent zijn. Twee condities zijn equivalent als ze unficeerbaar zijn. Dit alles impliceert dat het conflict in de onderstaande regels (bijvoorbeeld) niet door ARC gedetecteerd kan worden.

Voorbeeld:

IF A=1 THEN B=1

IF A=1 and C=1 THEN D=1

IF B=1 and C=1 THEN D=0

Hetzelfde geldt overigens voor subsumptie. En wat voor ARC geldt, geldt a fortiori voor CHECK.

Het bovenstaande betekent niet dat CHECK (of ARC) broddelwerk is. Een algoritme dat een grondiger controle uitvoert is weliswaar uit oogpunt van kwaliteitszorg zeer gewenst, maar er hangt ook een prijskaartje aan (in termen van complexiteit).

3.8 Algoritme van Bezem

In [Bezem87] wordt een op hyper-resolutie [Robinson65] gebaseerd algoritme beschreven dat zeer sterk doet denken aan de eerder besproken algoritmes van [Decker87] en [Sadri88]. Een deductieve-database- aanpak dus? Niet helemaal. Bezem gaat namelijk wel uit van een produktieregelsysteem. Via een formalisatie met behulp van meersoortige logica komt hij tot een *polynomiaal* algoritme voor consistentiecontrole, zij het voor een beperkte subklasse van kennissystemen. Deze subklasse wordt door Bezem geïndexeerd-propositionele expertsystemen genoemd. Zij omvat die expertsystemen waarbij het domein van ieder attribuut eindig is, waarin geen variabelen voorkomen en waarin voor een attribuut a met domein $\{c_1, \dots, c_n\}$ niet uit $a \neq c_i$ voor $i=1, \dots, n-1$ mag worden afgeleid dat $a = c_n$. Deze laatste voorwaarde mag wat vreemd overkomen, maar men moet zich realiseren dat in het algemeen geldt: "hoe krachtiger de inference engine, hoe duurder de consistentie-controle". Het is dus eigenlijk niet zo verwonderlijk dat een zwakte-eis aan de inferentie moet worden opgelegd om te

voorkomen dat consistentie-controle NP-moeilijk is. Ook de voorwaarde dat er geen variabelen in de 'produktieregels' mogen voorkomen valt in dit licht te begrijpen. Men kan zich geïndexeerd-proportionele expert systemen het beste voorstellen als bestaande uit produktie-regels, in de vorm van Horn-clauses met als basispredikaten 'attribuut (object, waarde)' voor meerwaardige en 'attribuut (object) = waarde' voor enkelwaardige attributen. Daarnaast mogen de *condities* van de produktieregels ook van de vorm 'attribuut (object) < waarde' zijn (of $\leq$, $>$, $\geq$). Als conclusie mag een dergelijke uitdrukking echter niet voorkomen op straffe van verlies van de polynomiale complexiteit.

Naast produktieregels zijn ook feiten en 'denials' toegestaan als expliciet verwoorde regels. Denials zijn van de vorm $\neg p(X) \vee \neg q(X) \vee \dots \vee \neg z(X)$ en kunnen opgevat worden als beperkingsregels die aangeven welke predikaten elkaar uitsluiten. Naast deze drie soorten expliciete axioma's zijn er dan nog impliciete: rekenregels voor de 'interne' (on)gelijkheid-predikaten $=$, $<$, en $>$. De belangrijkste in [Bez87] gepresenteerde bevinding is dat - indien men wil voorkomen dat consistentie-controle van exponentiële complexiteit is - deze impliciete axioma's zo moeten zijn dat men *niet* $a = c_i$ kan concluderen uit $\forall j (j \neq i \rightarrow a \neq c_j)$ of uit $a \geq c_i \wedge a \leq c_i$.

Bez87 tracht dit te bewerkstelligen voor het gelijkheidspredikaat door een 'speciale' waarde op te nemen in het waardengebied van ieder attribuut, die niet in de expliciete axioma's mag voorkomen. De ongelijkheidspredikaten mogen slechts 'ontkennend' (dus in beperkingsregels of in bodies van afleidingsregels) worden gebruikt.

Als bovenstaande beperkingen op de (on)gelijkheidspredikaten effectief geïmplementeerd worden blijven van de door Bez87 genoemde impliciete axioma's de onderstaande over:

- Voor gelijkheid:
 $\forall x (x = x)$
 $\forall x, y (x = y \rightarrow y = x)$
 $\forall x, y, z ((x = y \wedge y = z) \rightarrow x = z)$
 $\forall x, y ((x = y \wedge P(x)) \rightarrow P(y))$

- Voor ongelijkheid:
 $\forall x \neg(x < x)$
 $\forall x, y (x > y \rightarrow y < x)$
 $\forall x, y, z ((x < y \wedge y < z) \rightarrow x < z)$

De expliciete axioma's van DB zijn gegeven als clauses in de vorm van:

- Feiten:
 A_1
 $\vdots$
 A_p
- Beperkingen:
 $\neg B_{11} \vee \neg \dots \vee \neg B_{1m}$
 $\vdots$
 $\neg B_{n1} \vee \neg \dots \vee \neg B_{nk}$
 (equivalent met $\neg(B_{n1} \wedge \dots \wedge B_{nk})$)
- Afleidingsregels:
 $D_1 \vee \neg C_{11} \vee \neg \dots \vee \neg C_{1m}$
 $\vdots$
 $D_r \vee \neg C_{r1} \vee \neg \dots \vee \neg C_{rk}$
 (equivalent met $D_r :- C_{r1}, C_{r2}, \dots, C_{rk}$)

Het algoritme wordt dan als volgt door Bezem in pseudo-code gepresenteerd (een beperking bestaande uit 1 literal wordt als feit behandeld):

ZOLANG wegstrepingen mogelijk zijn en er geen lege clause is DOE

1. Streep alle *clauses* weg die een literal $\neg B_i$, $\neg C_{ij}$ of D_k bevatten die als feit voorkomt of (via de impliciete axioma's) afleidbaar is.

2. Streep alle *literals* $\neg B_i$, $\neg C_{ij}$ of D_k weg waarvan de ontkenning als feit voorkomt of (via de impliciete axioma's) afleidbaar is.

EINDE ZOLANG

3. Als er een lege clause is ontstaan door het wegstrepen van een literal of twee onverenigbare feiten $f(c, \dots, c') = c_i$ en $f(c, \dots, c') = c_j$ met $c_j \neq c_i$ dan is KB inconsistent.
4. Anders is KB consistent.

4 EEN VERGELIJKENDE ANALYSE

4.1 Inleiding

In het voorgaande hoofdstuk zijn al enige uitspraken gedaan over de verwachtingen m.b.t. de afzonderlijke algoritmen voor integriteitsbewaking. Dit hoofdstuk is specifiek bedoeld om tot een afweging van de voor- en nadelen van de verschillende methodes te komen. Daarbij zal het minder gaan om het vergelijken van zeer op elkaar gelijkende methoden (zoals [Decker86] en [Sadri88]). Juist de drie hoofdtypen zullen met elkaar vergeleken worden. Hier wordt gedoeld op het eerder aangebrachte onderscheid tussen database-gerichte, programma-gerichte en produktieregel-gerichte methoden. In de eerstvolgende paragraaf zal daarbij de aandacht met name gericht zijn op de invulling van het begrip integriteit (of, zo men wil, consistentie en volledigheid). De daarop volgende paragraaf is gewijd aan de haalbaarheid van de verschillende algoritmen in de zin van computationele efficiëntie. In de laatste paragraaf zullen we ingaan op de algemene vraag wat nu de beste aanpak lijkt. Is een van de drie beter? In hoeverre kunnen de voordelen van de verschillende aanpakken gecombineerd worden in een gemengde aanpak? In hoeverre is de beste invulling van integriteitsbewaking afhankelijk van het probleemgebied?

4.2 Invulling van integriteits-controle

Na lezing van het voorgaande hoofdstuk zal het duidelijk zijn dat de drie verschillende typen integriteitscontrole (database-gericht, programma-gericht en produktieregel-gericht) grote verschillen vertonen wat de concrete invulling van de integriteitscontrole betreft. In deze paragraaf zullen de onderlinge verschillen op een rijtje worden gezet.

Binnen de categorie van produktieregel-gerichte methoden wordt de meest uitgebreide integriteitscontrole uitgevoerd door de algoritmen van Nguyen. Aangenomen dat Puuronen's algoritme niet wezenlijk verschilt van dat van Suwa, is Nguyen de enige van de drie die verder gaat dan het controleren van 'eenstaps-inferentie' en ook de welgevormdheid van *regelketens* onderzoekt. Hierbij moet wel opgemerkt worden dat de

algemeenheid van deze regelketen-controle bij nader inzien toch wat teleurstellend is. Redundante en tegenstrijdige regelketens worden bijvoorbeeld alleen ontdekt als de begin-regels van beide ketens dezelfde of equivalente IF-parts hebben. Dit type redundantie c.q. tegenstrijdigheid is het meest flagrante en zal dus het gemakkelijkst te voorkomen zijn door zorgvuldiger te programmeren. Ook Nguyen's consistentie-controle mag dus allerminst als volledig betiteld worden.

Wanneer de concepten consistentie en volledigheid naast het integriteitsconcept van deductieve databases worden gelegd, dan blijkt dat er geen sprake is van subsumptie: beide invullingen van integriteit bevatten onderdelen die de ander niet bevat. Van de verschillende facetten van 'productieregel-consistentie' wordt in de database-gerichte aanpak slechts één bestreken (tegenstrijdigheid), maar dit gebeurt wel in veel ruimere zin, doordat laatstgenoemde aanpak de kennisingenieur in staat stelt in zeer algemene bewoordingen semantische restricties aan de kennisbank op te leggen. Het gebeurt bovendien grondiger. Iedere tegenstrijdigheid die afleidbaar is wordt ook feitelijk gedetecteerd. Door dit gegeven is het niet zo vreselijk dat subsumptie en redundantie niet detecteerbaar zijn. Immers, ze zijn niet onmiddellijk funest voor de betrouwbaarheid van het kennissysteem. Ze kunnen na onderhoud (veranderingen in de kennisbank) wel een bron van tegenstrijdigheden zijn, maar zodra een voorgenomen update hiertoe aanleiding dreigt te geven, wordt dit bij integriteitscontrole *wel* gedetecteerd. De afwezigheid van controle op circulariteit is ernstiger. Circulariteit wordt in zekere (wrange) zin gedetecteerd doordat de integriteitscontrole vastloopt. Ook de volkomen afwezigheid van volledigheidscontrole bij de 'deductieve aanpak' is een wezenlijke tekortkoming in de methode.

Voor het algoritme van Bezem gelden bovengenoemde nadelen des te sterker omdat slechts beperkingsregels van de vorm $\forall X \neg (p(X) \wedge \dots \wedge z(X))$ kunnen worden geformuleerd.

Het algoritme van Shapiro is bedoeld om drie potentiële fouten in programma's te detecteren: non-terminatie, onjuiste oplossingen en missende oplossingen. Afgaande op de omschrijvingen van consistentie en volledigheid, zoals die in de hoofdstukken 1 en 3 zijn gegeven, kan worden vastgesteld dat Shapiro's interpretatie verschillende aspecten van deze begrippen dekt. Wat betreft consistentie worden de aspecten tegenstrijdigheid

en circulariteit geanalyseerd met de algoritmen voor detectie van onjuiste oplossingen en non-terminatie. Het algoritme voor het detecteren van een missende oplossing geeft aan waar een regel in het programma ontbreekt om deze oplossing te kunnen genereren; dit is een aspect van volledigheid.

Non-terminatie wordt door Shapiro gedetecteerd doordat een vooraf bepaalde maximale recursiediepte door het programma wordt bereikt, zonder dat het tot een oplossing is gekomen. In vergelijking met de andere in hoofdstuk 3 besproken algoritmen is deze aanpak zeer pragmatisch, maar daarom niet minder doeltreffend. Tegenstrijdigheid is hier enigszins anders opgevat. Shapiro's gebruik van de term correctheid geeft aan deze problematiek een iets formelere grondslag. Het resultaat van de aanroep van een procedure p in een interpretatie M is correct, als alle subgoals van p , direct of indirect, een resultaat opleveren dat correct is in M . Shapiro's algoritme levert in dit geval een triplet van de vorm $\langle p, x, y \rangle$ op, dat de oorzaak is van het onjuiste resultaat. Als het gaat om een missende oplossing, dan moet er een onderscheid worden gemaakt tussen deterministische (bijv. Algol-derivaten en functionele talen als Lisp) en non-deterministische talen (bijv. logische programmeertalen, zoals Prolog). Stel dat y een correcte uitvoer van de aanroep $\langle p, x \rangle$ is. Als $\langle p, x \rangle$ termineert en een andere waarde dan y oplevert, dan is deze uitvoer, in het geval van een deterministische taal, *niet correct*. In een non-deterministische taal geldt dat als elke aanroep $\langle p, x \rangle$ een uitvoer correct in M oplevert, maar geen enkele aanroep y , dan faalt de aanroep $\langle p, x, y \rangle$. In dit geval is de procedure p *niet compleet*. Een programma P is compleet in M voor elk triplet $\langle p, x, y \rangle$ in M , als er een aanroep $\langle p, x \rangle$ is die y als resultaat heeft. Als het programma P faalt op enig triplet $\langle p, x, y \rangle$ in M , dan is P niet compleet in M . het probleem kan worden opgelost door de procedure p aan te passen, zodanig dat y wel kan worden opgeleverd.

4.3 Computationale efficiëntie

Grondige integriteitscontrole vertoont een sterke overeenkomst met goede raad: het is duur. In het vakgebied van de kunstmatige intelligentie zijn dure problemen meer regel

dan uitzondering. In principe staan een kunstmatig intelligentie onderzoeker drie wegen open om tot aanvaardbare oplossingen te komen tegen een aanvaardbare prijs:

- Inruilen van executietijd-complexiteit tegen geheugenruimte-complexiteit.
- Heuristische benadering.
- Op zeker spelen en minder hoge inhoudelijke eisen stellen.

Bij gebruikmaking van heuristieken moet men op de koop toe nemen dat men mis kan gokken en dat over het risico (de kans op missers) meestal grote onzekerheid bestaat. Indien men op zeker speelt - bijvoorbeeld door genoeg te nemen met een grovere representatie of minder flexibiliteit - weet men precies wat men opgeeft, maar daar staat tegenover dat er ook geen enkele kans meer is op de honderdduizend.

Heuristieken mogen zich binnen de kunstmatige intelligentie in een grote populariteit verheugen. Wanneer we echter de verschillende typen integriteitscontrole in dit licht onder de loep nemen blijkt dat bij geen van de methoden sprake is van een heuristische aanpak.

Bezem lijkt bij de conceptie van zijn algoritme expliciet te zijn uitgegaan van de vraagstelling "Hoe moet men de flexibiliteit van representatie en inferentie in produktieregel-systemen beperken opdat consistentie-controle (in de strikte, logische zin) kan worden gerealiseerd met een algoritme van polynomiale (tijd-)complexiteit". Op het eerste gezicht heeft hij hiermee goede resultaten geboekt. Bij nadere beschouwing blijkt echter dat het algoritme in de praktijk onvoldoende krachtig is om van consistentie-controle te kunnen spreken. Er wordt namelijk uitgegaan van een *vaste* verzameling expliciet gerepresenteerde feiten, terwijl die feiten uitgaande waarvan een produktieregel-systeem in de praktijk tot haar conclusies komt *per sessie* zullen verschillen. Men denke hierbij bijvoorbeeld aan een diagnostisch systeem dat bij raadpleging in verschillende sessies in het algemeen niet dezelfde waarden voor 'user-askable' attributen zal krijgen toegeschoven. Als het systeem slechts voor één situatie een behoorlijke oplossing kan aandragen is de betiteling expertsysteem een farce. Als een uitspraak over de consistentie van een kennisbank wordt gebaseerd op slechts één van de mogelijke situaties, is de consistentie-controle een farce. Het algoritme moet dus voor alle realistische situaties worden uitgevoerd. En het aantal realistische situaties is in het algemeen exponentieel

afhankelijk van het aantal 'user-askable' attributen. Hoewel Bezem's conclusie dat de berekening polynomiaal is in het totale aantal voorkomens van predikaten juist is, is de generalisatie naar 'een polynomiaal algoritme' dus onterecht.

Gezien het feit dat - zoals in hoofdstuk 3 reeds opgemerkt - Bezem's algoritme zeer dicht bij dat van Decker en andere database-gerichte integriteitsbewakings-algoritmen staat, zal het de lezer niet verbazen dat hetzelfde euvel optreedt bij de algoritmen van Decker en van Sadri en Kowalski. Een duidelijk verschil tussen de hybride aanpak van Bezem en de (volledig) database-gerichte aanpak is dat bij de laatste de computationele complexiteit met name bestreden wordt door het inruilen van spatiële tegen temporele resources. De occurs_pos en occurs_neg meta-predikaten van Decker reduceren de benodigde executie-tijd, terwijl een veelvoud van de geheugenruimte ingenomen door de afleidingsregels nodig is voor het opslaan van de occurs-informatie.

Ook Nguyen maakt gebruik van tabellen om de rekentijd te reduceren. Nog afgezien van het feit dat op basis van Nguyen's artikelen in dit rapport geen algoritme kon worden geconstrueerd, lijkt het toch niet het ei van Columbus voor integriteitsbewaking. De consistentie-controle is daarvoor te beperkt.

Shapiro hanteert twee maten voor computationele complexiteit, t.w. lengte-complexiteit en diepte-complexiteit. Het begrip lengte heeft betrekking op het aantal knooppunten in een 'proof tree', diepte op de 'diepte' (aantal niveaus waarop subgoals worden uitgevoerd) van de 'proof tree'. Alvorens hier verder op in te gaan, moet eerst het concept 'grootte van een goal' worden geïntroduceerd. Volgens [Sterling87] is de grootte van een term gelijk aan het aantal symbolen in de textuele representatie, waarbij atomen en variabelen grootte 1 hebben. Een samengestelde term heeft een grootte die gelijk is aan de som van de grootten van de argumenten, plus 1. De term 'voorouder(PersoonA,PersoonB)' heeft grootte 4. Een programma P heeft een lengte-complexiteit $L(n)$, als voor elk goal A in $M(P)$ met grootte n geldt dat de lengte $\leq L(n)$. de lengte-complexiteit is verbonden met de eerder genoemde executietijd-complexiteit. Een programma P heeft een diepte-complexiteit $D(n)$, als voor elk goal A in $M(P)$ met grootte n geldt dat er een bewijs van A in P is met diepte $\leq D(n)$. Diepte-complexiteit is gerelateerd met geheugenruimte-complexiteit. De gegeven algoritmen zijn lineair in de lengte van de berekeningen. Shapiro is weliswaar van mening dat computationele efficiëntie een goede maatstaf is

voor de beoordeling van de algoritmen, hij geeft echter de voorkeur aan het optimaliseren van de query-complexiteit, d.w.z. het aantal en soort vragen dat een 'ground oracle' (veelal een menselijke gebruiker, maar mogelijk ook een ander programma) moet beantwoorden. Met name voor het algoritme dat onjuiste oplossingen moet detecteren is een variant ontwikkeld, genoemd 'divide and query', dat een aantal vragen stelt logaritmisch in de lengte van het bewijs.

4.4 Toepasbaarheid in knowledgebase-systemen

Het is niet goed mogelijk om een van de typen integriteitscontrole te bestempelen als de beste in alle omstandigheden. Wat de beste methode is zal waarschijnlijk afhangen van de beschikbare executie-tijd en geheugenruimte. Ook de vraag hoe omvangrijk de kennisbank mag zijn voordat de voor de integriteitscontrole benodigde rekentijd onaanvaardbaar wordt blijft een open vraag, omdat de verschillende algoritmen niet op realistische applicaties konden worden getest. Wel kan een - redelijk gefundeerd - antwoord gegeven worden op de vraag welke algoritmes als eerste in aanmerking komen voor nader onderzoek en van welke *combinatie* een kwalitatief hoogstaand resultaat mag worden verwacht.

Van het algoritme van Bezem kan in de praktijk weinig heil verwacht worden. Slechts voor kennissystemen die heel duidelijk meer database dan rulebase zijn mag er iets van verwacht worden en dan nog alleen als de regels geen variabelen bevatten. Men moet zich hierbij goed het verschil tussen variabelen en parameters (attributen) realiseren. Ook parameters en attributen zijn onderworpen aan waarde-toekenning. Het verschil met variabelen zit hem *in deze context* hierin dat variabelen *ongetypeerd* zijn. Via unificatie kan voor een variabele in feite een willekeurige expressie worden gesubstitueerd, terwijl een parameter een vast object is dat slechts een waarde toegemeten kan krijgen die binnen zijn domein ligt. Een verbod op variabelen in afleidingsregels is tamelijk destructief en kan leiden tot een onoverzichtelijke hoop ad-hoc-regeltjes.

Het algoritme van Nguyen kon niet worden gereconstrueerd en het is beslist af te raden dat alsnog te proberen als men geen andere bronnen dan de in dit rapport opgenomen

literatuur-verwijzingen tot zijn beschikking heeft. Ook als men die wel heeft is het twijfelachtig of het Nguyen-algoritme voldoende betrouwbaarheid kan garanderen met het oog op de in de voorgaande paragrafen - met name 4.2 - aangestipte tekortkomingen. Hoewel ook niet zaligmakend, biedt het algoritme van Puuronen/Suwa een beter perspectief, mits het kan worden gegeneraliseerd naar AND/OR-grafen met diepte groter dan 1. Mogelijk is het RETE pattern-matching algoritme [Forgy82] hiervoor een goed (dat wil zeggen krachtig en efficiënt) startpunt. Het is wel denkbaar dat dit de ruimte-complexiteit flink doet toenemen. Wanneer men er op gebrand is een systeem met hoge betrouwbaarheid te maken is het echter het overwegen waard om van produktieregelsystemen af te zien. Bij veel inferentieprocessen zijn produktieregels namelijk ook uit conceptuele overwegingen te versmiden [Nau87] en verdient een meer grafen-theoretische aanpak de voorkeur.

De deductieve database-aanpak is met name geschikt voor kennis-systemen van de tweede generatie. De interne modellen van een dergelijk systeem vormen in zo'n geval een *vaste* (niet per sessie veranderende) feitenbank. Ook wanneer men van een produktieregel-formalisme afziet biedt een deductieve database goede mogelijkheden (dankzij de grote flexibiliteit van Prolog en het feit dat de integriteitscontrole in laatste instantie van de Prolog-interpretator gebruik maakt. ENIAM is voor deductieve databases een zeer geschikte specificatie-methode, zowel voor afleidings- als beperkingsregels. Van de twee besproken methoden in deze categorie is die van Decker verreweg het gemakkelijkst te implementeren. De methode zou verrijkt kunnen worden met een circulariteits-detector à la Shapiro en eventueel volledighedscontrole à la Nguyen. Verder is het in geval van 'user askable' parameters absoluut noodzakelijk dat de integriteitscontrole alle mogelijke (toegestane) parameterwaarde-combinaties omvat.

Het algoritme van Shapiro biedt goede mogelijkheden voor de integratie van conceptuele modellering (ENIAM) en integriteitscontrole. Hoofdstuk 2 bood daarvan een klein voorbeeld. Bij het vaststellen van non-terminatie is het een gemis dat het algoritme niet meer ondersteuning biedt bij de verdere diagnose van de 'proof tree' die wordt opgeleverd. Aanvulling met technieken zoals voorgesteld door Nguyen valt te overwegen. In de analyse van correctheid en compleetheid is het duidelijk geworden dat de gebruiker een belangrijke (interactieve) rol speelt. Door het kiezen van varianten met een zo gunstig mogelijke query-complexiteit en het uitbreiden van de algoritmen naar

'volledig' Prolog, kan een geïntegreerde AI-ontwikkelomgeving gerealiseerd worden. De mogelijkheden die worden geboden door het 'Model Inference System', waaronder automatische 'reparatie' van foute procedures, kunnen hierbij zeker een rol spelen.

5 CONCLUSIES EN AANBEVELINGEN

5.1 Conclusies

1. E(xtended)NIAM biedt betere mogelijkheden om de specificaties van een knowledgebase vast te leggen dan NIAM. Met name de mogelijkheden om niet-grafische beperkingsregels van NIAM toch grafisch weer te geven is van groot belang bij een directe validatie van het conceptuele model door een expert. Het is echter jammer dat aan een andere tekortkoming van NIAM in de E(xtended) versie nog geen aandacht is besteed, namelijk de expliciete vastlegging van temporele aspecten.
2. Het ARC-algoritme van Nguyen biedt meer mogelijkheden voor integriteitscontrole dan dat van Suwa (Puuronen). Het is echter bij lange na niet zo grondig als op het eerste gezicht lijkt. Weliswaar besteedt het aandacht aan regelketens in plaats van regels, maar het zou absoluut niet terecht zijn te spreken van een generalisatie naar regelketens. Slechts van regelketens met gelijke of equivalente beginpunten kan ARC namelijk vaststellen of ze tegenstrijdig of subsumerend zijn. Bij gebrek aan een exacte specificatie van het algoritme kan over de computationele complexiteit van ARC geen verantwoorde uitspraak gedaan worden.
3. In de deductieve-database aanpak wordt in vergelijking met die van Nguyen (en Shapiro) te weinig - namelijk geen - aandacht besteedt aan circulariteit en onvolledigheid. Daar staat tegenover dat semantische (deductieve-)database integriteit veel meer omvat dan Nguyen's tegenstrijdigheid. Bovendien is een algoritme als dat van Decker gemakkelijk in Prolog te implementeren. Door de vereenvoudiging van beperkingsregels tot update-constraints en vooral door het benutten van meta-informatie over de afleidingsregels wordt de computationele complexiteit sterk gereduceerd. De prijs die hiervoor betaald moet worden is verveelvoudiging van de benodigde geheugenruimte: Bij een gemiddelde van k literals per rule-body is ongeveer $k+1$ maal zoveel geheugenruimte nodig om naast

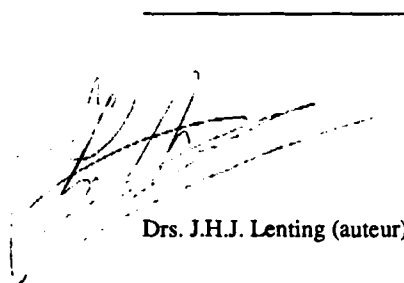
de oorspronkelijke database de meta-informatie (occurs_in-predikaten) en een copie van de database op te slaan.

4. Door gebruik te maken van het Shapiro algoritme is het mogelijk om een in Prolog executeerbare knowledgebase-specificatie te onderzoeken op consistentie en volledigheid. Het is wel noodzakelijk dat het met behulp van ENIAM opgestelde conceptuele model is om te zetten naar een Prolog-programma. Een aanzet hiertoe, die zich beperkt tot de formalisering van niet-grafische beperkingsregels (zoals afleidings- en inferentieregels) in Prolog, is in dit rapport weergegeven. De computationele efficiëntie van dit algoritme is gerelateerd aan een tweetal maten, t.w. de 'diepte' van de 'bewijs-boom (proof tree) en de 'lengte' van een bewijs. Voorop staat het streven naar een zo gering mogelijk aantal 'vragen' dat door het systeem aan de ontwikkelaar wordt gesteld (eventueel aan een ander programma). Voor het gedeelte van het algoritme dat non-terminatie van programma's detecteert is een kunstmatige grens gesteld, t.w. de maximale recursiediepte. Bij het algoritme voor het detecteren van onjuiste uitkomsten is de complexiteit lineair in de lengte van de bewijs-boom, waarbij moet worden opgemerkt dat er een gewijzigde versie is waarbij de complexiteit lineair in de diepte van de bewijs-boom is. De complexiteit van het algoritme voor het detecteren van een missende oplossing is lineair in de lengte van de 'falende' berekening.
5. In tegenstelling tot de algoritmen van o.a. Suwa, Nguyen en Puuronen, is dat van Shapiro oorspronkelijk niet bedoeld om te worden toegepast in de context van kunstmatig intelligente systemen. Het doel was een formele theorie te ontwerpen voor het debugging-proces. Het gebruik van de taal Prolog is ingegeven door de omstandigheid dat er een directe relatie bestaat tussen de syntax (structuur) en semantiek (betekenis) van Prolog. Dit maakt het eenvoudiger om foute programma's te diagnostiseren en corrigeren. Deze eigenschap komt zeer van pas bij de analyse van knowledgebase-specificaties, omdat een Prolog-programma kan worden opgevat als een formalisering van deze specificaties. Het is aldus méér dan alleen maar een 'programma'. Hier moet wel worden opgemerkt dat in dit rapport is uitgegaan van een 'beperkte' Prolog. Een uitbreiding in de richting van een 'volwassen' Prolog wordt door Shapiro goed mogelijk geacht [Shapiro83].

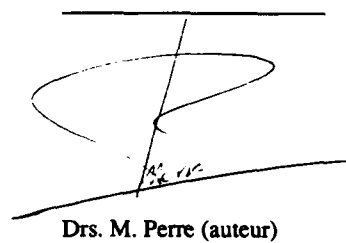
6. Een combinatie van ENIAM, Prolog en de algoritmen van Decker en Shapiro stelt de ontwikkelaar van knowledgebase-specificaties in staat om binnen eenzelfde theoretisch raamwerk (incrementeel) te werken aan een consistent en volledig conceptueel model van het probleemdomein. Het feit dat hier de methode (ENIAM) het uitgangspunt is, maakt dat deze aanpak degelijker is dan die van Suwa, Nguyen en Puuronen. Deze laatsten voeren weliswaar een tamelijk uitgebreide *syntactische* analyse van de knowledgebase uit, maar besteden hoegenaamd geen aandacht aan *semantische* integriteit (beperkingsregels). Het onderliggende (conceptuele) model van de werkelijkheid speelt bij hen vrijwel geen rol.

5.2 Aanbevelingen

1. De algoritmen van Decker en Shapiro lijken de beste papieren te hebben, zowel wat *performance*, conceptuele helderheid en implementatie-gemak (in Prolog) betreft.
2. Nadere toetsing van deze algoritmen aan de hand van een praktijkvoorbeeld is geboden om een gefundeerd oordeel te kunnen vormen over toepasbaarheid en computationele efficiëntie.
3. Indien bij een dergelijke toetsing zou blijken dat de computationele efficiëntie onvoldoende is zijn de volgende alternatieven voorhanden: a.) Men kan van het algoritme van Decker overstappen op dat van Sadri met gebruikmaking van een efficiëntere resolutie (dan Prolog-SLDNF). Hoeveel winst dit oplevert is een open vraag en het zal enige extra inspanning vergen. b.) Men kan afstappen van de eerste aanbeveling en het algoritme van Puuronen proberen in te passen in een RETE-achtige meta-informatie-structuur om zo een volledige generalisatie naar regelketens te verkrijgen. Dit zal veel extra inspanning vergen en draagt het risico in zich dat semantische aspecten uiteindelijk te weinig aandacht krijgen. Bovendien is het algoritme van Puuronen slechts efficiënt indien de knowledgebase kan worden opgesplitst in kleine, onafhankelijke modulen.



Drs. J.H.J. Lenting (auteur)



Drs. M. Perre (auteur)



Ir. J. Bruin (projectleider)

LITERATUURVERWIJZINGEN

- [Akkermans89] Akkermans, J.M. (e.a.), "Naar een Formele Specificatie van Kennismodellen", In: "Proceedings NAIC '89", 1989, pp. 105-119.
- [Atzeni88] Atzeni, P. en D.S. Parker Jr., "Formal Properties of Net-based Knowledge Representation Schemes", In: "Data & Knowledge Engineering", Vol. 3, 1988.
- [Bakker87] Bakker, R.R., "Knowledge Graphs: Representation and Structuring of Scientific Knowledge", Universiteit Twente, 1987.
- [Bezem87] Bezem, M., "Consistency of Rule-Based Expert Systems", Report CS-R8736, CWI, Computer Science/Department of Software Technology, 1987.
- [Breuker87] Breuker, J. (ed.), "Model-Driven Knowledge Acquisition: Interpretation Models", University of Amsterdam, 1987.
- [Creasy88] Creasy, P.N., "Extending Graphical Conceptual Schema Languages", University of Queensland, 1988.
- [Creasy89] Creasy, P.N., "ENIAM: A More Complete Conceptual Schema Language", In: "Proceedings of the Fifteenth International Conference on Very Large Databases", 1989, pp. 107-114.
- [Davis82] Davis, R. en D. Lenat, "Knowledge-based Systems in Artificial Intelligence", McGraw-Hill, 1982.
- [Decker88] Decker, H., "Integrity Enforcement on Deductive Databases", In: Kerschberg, L. (ed.), "Expert Database Systems, Proceedings of the First International Conference on Expert Database Systems", 1988, pp. 381-395.
- [Deutsch82] Deutsch, M.S., "Software verification and validation: Realistic project approaches", 1982.
- [FEL-89-A267] Lenting, J.H.J. en M. Perre, "Kwaliteit van Expertsystemen: Methoden en Technieken", FEL-89-A267, 1989.
- [FEL1989-148] Dekker, S.T., Lenting J.H.J., Perre M., Rutten J.J.C.R., "Kwaliteit van Expertsystemen: Een Oriëntatie", FEL 1989-148, 1989.
- [Forgy82] Forgy C.L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", In: "Artificial Intelligence", Vol. 19, 1982, 17-37.
- [Hamdan89] Hamdan, A.R. en C.J. Hinde, "Fault Detection Algorithm for Logic Programs", In: "Knowledge-Based Systems", Vol. 2, Nr. 3, 1989.

- [Kowalski75] Kowalski R.A., "A Proof Procedure Using Connection Graphs", In: "Journal of the ACM", Vol. 22, Nr. 4, 1975, pp. 572-595.
- [Laffey86] Laffey T.J., Perkins W.A., Nguyen T.A., "Reasoning about Fault Diagnosis with LES", In: "IEEE Expert", Vol. 1, Nr. 1, 1986, pp. 13-20.
- [Lloyd85] Lloyd J.W., Topor R.W., "A Basis for Deductive Database Systems", In: "Journal of Logic Programming", Vol. 2, Nr. 2, 1985, pp. 93-109.
- [Lloyd86a] Lloyd J.W., Sonenberg E.A., Topor, R.W., "Integrity Constraint Checking in Stratified Databases," Technical Report 86/5, Department of Computing Science, University of Melbourne, 1986.
- [Lloyd86b] Lloyd J.W., Topor R.W., "A Basis for Deductive Database Systems II", In: "Journal of Logic Programming", Vol. 3, Nr. 1, 1986, pp. 55-67.
- [Marek87] Marek, W., "Completeness and Consistency in Knowledge Base Systems", In: Kerschberg L. (ed.), "Expert Database Systems", Benjamin/Cummings, 1987.
- [Nau87] Nau D.S., Reggia J.A., "Relationships between Deductive and Abductive Inference in Knowledge-Based Diagnostic Problem Solving". In: L. Kerschberg (ed.), "Proceedings of the 1st International Workshop on Expert Database Systems", 1987.
- [Nguyen85] Nguyen, T.A., Perkins, W.A., Laffey, T.J., Pecora, D., "Checking an Expert Systems Knowledge Base for Inconsistency and Completeness", In: "Proceedings of the Ninth International Joint Conference on AI", Vol. 1, 1985, pp. 375-378.
- [Nguyen87a] Nguyen, T.A., "Verifying Consistency of Production Systems", In: "Proceedings of the IEEE 3rd Conference on AI", 1987, pp. 4-8.
- [Nguyen87b] Nguyen, T.A., Perkins W.A., Laffey T.J., Pecora D., "Knowledge Base Verification", In: "AI Magazine", 1987, pp. 69-75.
- [Nicolas82] Nicolas, J.M., "Logic for Improving Integrity Checking in Relational Data Bases", In: "Acta Informatica", Vol. 18, 1982, pp. 227-253.
- [Nijssen86] Nijssen, G.M., "Knowledge Engineering, Conceptual Schemas, SQL and Expert Systems: A Unifying Point of View", In: "Relationele Database Software, 5e Generatie Expertsystemen en Informatie-analyse", Congres-syllabus NOVI, 1986, pp. 1-38.
- [Puuronen87] Puuronen S., "A tabular Rule-Checking Method", In: "Proceedings 7th International Workshop on Expert Systems and their Applications", 1987, pp. 257-266.
- [Reboh81] Reboh, R., "Knowledge Engineering Techniques & Tools in the Prospector Environment", SRI Technical Note 243, Stanford research Institute, 1981.
- [Rich79] Rich, C., "Initial report on the LISP Programmer's Apprentice, In: "IEEE Transactions on Software Engineering", Vol. 4, Nr. 6, 1979, pp. 342-376.

- [Robinson65] Robinson J.A., "Automatic Deduction with Hyper-resolution", In: "International Journal of Computer Mathematics", Vol. 1, 1965, pp. 227-234.
- [Sadri88] Sadri, F., Kowalski R., "A Theorem-Proving Approach to Database Integrity", In: Minker J. (Ed), "Deductive Databases and Logic Programming", Morgan Kaufmann, 1988.
- [Shapiro84] Shapiro, E.Y., "Algorithmic Program Debugging", The MIT Press, 1984.
- [Soper86] Soper P.J.R., "Integrity Checking in Deductive Databases", MSc. Thesis, Department of Computing, Imperial College, University of London, 1986.
- [Sowa84] Sowa, J.F., "Conceptual Structures: Information Processing in Mind and Machine", Addison Wesley, 1984.
- [Sterling87] Sterling, L. en E.Y. Shapiro, "The Art of Prolog", The MIT Press, 1987.
- [Stonebraker86] Stonebraker, M. en L.A. Rowe, "The design of POSTGRES", In: "Proceedings of the ACM-SIGMOD Conference on Management of Data", 1986, pp. 340-355.
- [Stonebraker88] Stonebraker, M. (e.a.), "The POSTGRES Rule Manager", In: "IEEE Transactions on Software Engineering", Vol. 14, Nr. 7, Juli 1988, pp. 897-907.
- [Stonebraker89] Stonebraker, M. en M. Hearst, "Future Trends in Expert Database Systems", In: Kerschberg, L. (ed.), "Expert Database Systems, Proceedings from the Second International Conference", 1989, pp. 3-20.
- [Suwa82] Suwa, M., A.C. Scott en E.H. Shortliffe, "Completeness and Consistency in Rule-Based Expert Systems", In: Buchanan, B.G. (ed.), "Rule-Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project", Addison-Wesley, 1984.
- [VanHekken89] Hekken, M.C. van, P.J.M. van Oosterom en M.R. Woestenburg, "A Geographical Extension to the Relational Data Model", Internal Report FEL-89-B207, 1989.
- [VanMelle84] Van Melle, W. (e.a.), "EMYCIN: A Knowledge Engineer's Tool for constructing Rule-Based Expert Systems", In: Buchanan, B.G. (ed.), "Rule-Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project", Addison-Wesley, 1984.
- [Vieille88] Vieille, L., "Recursive Query Processing: Fundamental Algorithms and the Ded Gin System", In: Cray, P.M.D. (ed.), "Prolog and Databases", Halsted Press/Wiley & Sons, 1988.
- [Wensel88] Wensel, S., "The POSTGRES reference manual", Memo Nr. UCD/ERL M88/20, University of California, Berkeley, 1988.

[Wintraecken85] Wintraecken, J.J.V.R., "Informatie-Analyse volgens NIAM", Academic Service, 1985.

[Wintraecken88] Wintraecken, J.J.V.R., "Informatiemodellering volgens NIAM", In: "Proceedings Semantiek van Gegevensmodellen: Het Tijdperk na Codd", 1988, pp. 45-76.

REPORT DOCUMENTATION PAGE

(MOD-NL)

1. DEFENSE REPORT NUMBER (MOD-NL) TD 89-4578	2. RECIPIENT'S ACCESSION NUMBER	3. PERFORMING ORGANIZATION REPORT NUMBER FEL-89-A312
3. PROJECT/TASK/WORK UNIT NO. 21896	5. CONTRACT NUMBER	6. REPORT DATE DECEMBER 1989
7. NUMBER OF PAGES 61	8. NUMBER OF REFERENCES 36	9. TYPE OF REPORT AND DATES COVERED FINAL REPORT
10. TITLE AND SUBTITLE KWALITEIT VAN EXPERTSYSTEMEN: ALGORITMEN VOOR INTEGRITEITS-CONTROLE (QUALITY OF EXPERT SYSTEMS: ALGORITHMS FOR INTEGRITY CONTROL)		
11. AUTHOR(S) J.H.J. LENTING AND M. PERRE		
12. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) TNO PHYSICS AND ELECTRONICS LABORATORY, P.O. BOX 96863, THE HAGUE, THE NETHERLANDS UNIVERSITY OF LIMBURG, P.O. BOX 616, MAASTRICHT, THE NETHERLANDS RESEARCH INSTITUTE FOR KNOWLEDGE SYSTEMS, P.O. BOX 463, MAASTRICHT, THE NETHERLANDS		
13. SPONSORING/MONITORING AGENCY NAME(S) MOD-NL DIRECTOR DEFENCE RESEARCH AND DEVELOPMENT		
13. SUPPLEMENTARY NOTES THE PHYSICS AND ELECTRONICS LABORATORY IS PART OF THE NETHERLANDS ORGANIZATION FOR APPLIED SCIENTIFIC RESEARCH		
15. ABSTRACT (MAXIMUM 200 WORDS, 1033 POSITIONS) THIS REPORT IS THE RESULT OF THE THIRD PHASE OF THE TECHNOLOGY PROJECT "QUALITY OF EXPERT SYSTEMS". CARRIED OUT UNDER THE COMMISSION OF THE MINISTRY OF DEFENCE, DIRECTOR DEFENCE RESEARCH AND DEVELOPMENT. PARTICIPANTS IN THE PROJECT ARE TNO PHYSICS AND ELECTRONICS LABORATORY (FEL-TNO), UNIVERSITY OF LIMBURG (RL) AND THE RESEARCH INSTITUTE FOR KNOWLEDGE SYSTEMS (RIKS) BASED ON (FEL1989-138) AND (FEL-89-A267) THIS REPORT CONTAINS THE RESULTS OF AN INVESTIGATION INTO ALGORITHMS FOR INTEGRITY CONTROL OF KNOWLEDGEBASES, WITH SPECIFIC INTEREST IN THE COMPUTATIONAL EFFICIENCY. ALGORITHMS FOR PRESERVING CONSISTENCY AND INTEGRITY OF KNOWLEDGEBASES ARE BEING COMPARED. ANOTHER SUBJECT IN THIS REPORT IS THE WAY IN WHICH A MAPPING CAN BE MADE BETWEEN SPECIFICATIONS MADE WITH E(XTENDED)NIAM AND AN EXECUTABLE PROLOG PROGRAM, IN ORDER TO ESTABLISH A FORMAL DETERMINATION OF THE CONSISTENCY AND INTEGRITY OF KNOWLEDGEBASE SPECIFICATIONS.		
16. DESCRIPTORS ARTIFICIAL INTELLIGENCE DATABASES EXPERT SYSTEMS QUALITY CONTROL ALGORITHMS	IDENTIFIERS KNOWLEDGE BASES	
17a. SECURITY CLASSIFICATION (OF REPORT) UNCLASSIFIED	17b. SECURITY CLASSIFICATION (OF PAGE) UNCLASSIFIED	17c. SECURITY CLASSIFICATION (OF ABSTRACT) UNCLASSIFIED
18. DISTRIBUTION/AVAILABILITY STATEMENT UNLIMITED AVAILABLE		17d. SECURITY CLASSIFICATION (OF TITLES) UNCLASSIFIED