Multi-Prolog: een bordgebaseerde parallelle logische programmeertaal

Koenraad O.M. De Bosschere

Promotor: Prof. Dr. ir. J. Van Campenhout

Proefschrift ingediend tot het behalen van de graad van
Doctor in de Toegepaste Wetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: Prof. Dr. ir. M. Vanwormhoudt
Faculteit van de Toegepaste Wetenschappen
Academiejaar 1992-1993

Woord vooraf

In het voorbije decennium is er heel wat gebeurd op het vlak van het logisch programmeren. Na een stille start in het begin van de zeventiger jaren met de ontwikkeling van de eerste Prologvertolker, kwam dit domein in de volle belangstelling door de aankondiging van het ambitieuze Japanse project dat tot doel had de computer van de vijfde generatie te bouwen. Deze computer zou op massaal parallelle wijze logische programma's uitvoeren.

Mede als gevolg van de belangstelling die door dit Japanse initiatief veroorzaakt werd, ontstonden er tal van parallelle logische programmeertalen. De best gekende zijn CONCURRENT PROLOG, PARLOG en GUARDED HORN CLAUSES. Deze geëngageerde-keuzetalen waren wel parallel, maar de prijs die hiervoor betaald moest worden was hoog: er waren sequentële en parallelle conjuncties en disjuncties nodig, het gedrag van sommige predicaten moest worden vastgelegd aan de hand van extra annotaties, logische programma's retourneerden slechts één oplossing, en dergelijke meer. Deze talen waren duidelijk in de eerste plaats ontworpen om massief parallel uitgevoerd te kunnen worden en de declaratieve en logische eigenschappen kwamen dan ook op de tweede plaats te staan. Deze parallelle logische programmeertalen hebben hun potentële gebruikers dan ook niet kunnen overtuigen de stap te wagen.

Terwijl de meeste ogen gericht waren op wat er zich op het zonet beschreven front aan het afspelen was, was er ook een aantal onderzoekers die er de voorkeur aan gaven PROLOG te parallelliseren in plaats van over te stappen naar een taal die in hun ogen minderwaardig was. Zij wilden de elegantie van de logica niet laten varen in ruil voor een potentieel betere prestatie. Talen zoals ANDORRA [HB88] en MUSE [AR90] zijn hiervan het resultaat.

Daarnaast trachtte een aantal onderzoekers het beste van de twee werelden te combineren: enerzijds de logische aanpak van PROLOG, en anderzijds de flexibiliteit om parallellisme te specificeren waar en wanneer men dat wil zoals in de geëngageerde-keuzetalen. Het resultaat was taakparallellisme. Taakparallellisme creëert taken die intern als een logisch programma uitgevoerd worden, maar die bij de onderlinge communicatie zich niet noodzakelijk als logisch programma gedragen.

Ook binnen het domein van de taakparallelle logische programmeertalen

tekenen er zich stromingen af. Sommigen zweren bij een gesloten kanaalmodel als methode van communiceren terwijl anderen dan weer een open manier van communiceren verkiezen. Er zijn er die proberen de logica van de taken over de grenzen van de taken heen te bewaren door indien nodig ook gespreid te gaan terugzoeken terwijl anderen verkiezen dit niet te doen om de taal efficiënt te kunnen implementeren.

In dit proefschrift wordt een nieuwe logische programmeertaal voorgesteld, gebaseerd op taakparallellisme en expliciete bordcommunicatie. De voorgestelde taal is krachtiger dan de vergelijkbare talen omdat ze (i) ofwel expressiever is, ofwel (ii) flexibeler in gebruik. Bovendien is er op geen enkel punt toegegeven aan de syntaxis van PROLOG waardoor het er een echte uitbreiding van is. De taal werd geïmplementeerd op een multiprocessor met gemeenschappelijk geheugen.

Dit proefschrift is praktisch van aard. Dit kadert in de traditie van de "toegepaste wetenschappen" die poogt om bestaande inzichten in de wetenschappen aan te wenden om bruikbare toepassingen te ontwikkelen. De weinige keren dat er dan toch theoretische beschouwingen gemaakt worden hebben zij enkel als doel bijkomende inzichten te verschaffen en zijn in geen geval een doel op zich.

Het is passend hier de gelegenheid te baat te nemen om een aantal feiten te herdenken, en een aantal mensen te bedanken die bijgedragen hebben tot het tot stand komen van dit werk.

Dit project startte eind 1988 in het kader van het nationaal stimuleringsprogramma voor fundamenteel onderzoek inzake artificiële intelligentie, uitgevoerd op initiatief van de Belgische Staat — Diensten van de Eerste Minister — Programmatie van het Wetenschapsbeleid (DPWB), dat tot doel had om een aantal pas afgestudeerden de kans te geven met het onderzoek in de artificiële intelligentie in aanraking te komen. In het kader van dit project had het toenmalig Laboratorium voor Elektronica en Meettechniek zich ertoe verbonden een parallelle logische programmeertaal te ontwikkelen. Dit was een voortzetting van eerder onderzoek naar de implementatie van parallelle procedurale programmeertalen [VCSD84, DB86, DB87, VCD87].

Een studie van de op dat ogenblik bestaande parallelle logische programmeertalen maakte duidelijk dat zij nagenoeg allemaal gebaseerd waren op een vorm van zogenaamd *en/of-parallellisme*, waarbij een *logische variabele* gebruikt wordt als communicatiekanaal [DB89c, DB89d]. Hoe eenvoudig het basisprincipe ook was, de implementatie bleek steeds heel wat moeilijker te zijn, niet in het minst omdat deze talen een haast ontelbaar aantal parallelle taken in het leven moeten roepen om een programma te kunnen uitvoeren.

Langzaam aan groeide onze overtuiging dat een alternatieve parallellisering op basis van een beperkt aantal grofkorrelige sequentële taken even waardevol kon zijn. Deze sequentiële taken zouden dan expliciet met elkaar moeten communiceren zonder hierbij de gemeenschappelijke variabelen te gebruiken. Einde 1989 was de idee voldoende rijp [DB89a, DB89b] om aan een implementatie te beginnen.

In 1989 kende het Nationaal Fonds voor Wetenschappelijk Onderzoek mij een mandaat van aspirant navorser toe op basis van het zonet vermeldde onderzoeksvoorstel. Dit mandaat van het NFWO zou mij in staat stellen om gedurende vier jaar vrij van elke administratieve beslommering mijn onderzoek uit te voeren. Dit mandaat is zonder overdrijving een zegen voor mijn onderzoek geweest.

De nodige componenten werden aangekocht. Hierbij wil ik ir. Jan Lagast en ir. Diederik Houtteman niet vergeten die zich ingezet hebben voor de aankoop van een multiprocessor met gemeenschappelijk geheugen en vier processoren. Tevens zou ik Mats Carlson van het Zweedse Instituut voor Computerwetenschappen (SICS) willen bedanken voor de steeds bereidwillige medewerking en het ter beschikking stellen van de implementatiedetails van SICSTUS PROLOG.

Halverwege 1990 kregen wij kortstondig het gezelschap van ir. Frank Gielen van de Koninklijke Militaire School te Brussel die in het kader van een ander project de ware-tijdskern MTOS-UX kwam implementeren op onze multiprocessor. De beschikbaarheid van deze kern maakte het later mogelijk om op korte tijd een parallel prototype te realiseren.

Vanaf 1990 kon ik rekenen op de medewerking van ir. Lieven Wulteputte die zich de eerstvolgende twee jaar intensief zou inzetten voor de realisatie van een prototype, eerst als student, later als medewerker aan het project. Zijn hulp bij de implementatie [DBW90, DBW91, DBW92] is werkelijk van onschatbare waarde geweest en bovendien wist hij op muzikale manier een aangename sfeer tijdens het werk te scheppen. Dank je wel, Lieven.

Mijn promotor, Prof. Jan Van Campenhout, ben ik ook heel wat dank verschuldigd. Hij was diegene die mij aanzette met dit onderzoek te beginnen. Hij was ook diegene die mij toeliet mijn eigenzinnige ideën over parallelle logische programmeertalen te realiseren en mij daarbij de nodige logistieke ondersteuning verleende. Bovendien was hij steeds een gewillige gesprekspartner om over mijn onderzoek te praten en mijn aandacht te vestigen op hiaten en mogelijke alternatieven. Hij ligt dan ook aan de basis van een aantal van de ideën die in dit proefschrift uitgewerkt worden.

Mijn dank gaat ook uit naar Prof. Saumya Debray van de universiteit van Arizona en zijn echtgenote Wendy, die zorgden voor een bijzonder aangename tijd toen ik samen mijn echtgenote voor een vier maanden durend studieverblijf aldaar verbleef. Van hem leerde ik hoe het wereldje van het logisch

programmeren min of meer in elkaar steekt.

Ik zou ook Dr. Jean-Marie Jacquet van het "Institut d'Informatique" van de Facultés Universitaires N.D. de la Paix te Namen willen bedanken voor de tijd en moeite die hij investeerde in het opstellen van de semantiek van de taal MULTI-PROLOG. Hij wist mijn schuchtere pogingen [DB90a, DB90b, DB91a] op te tillen tot een volwaardige semantische beschrijving [DBJ92].

Verder zou ik ook mijn collega's van de vakgroep willen bedanken voor hun collega-zijn. Twee van hen, ir. Peter Veelaert en ir. Ronny Blomme zou ik in het bijzonder willen bedanken voor het nalezen van dit proefschrift. Ik zou zeker ook de voorzitter Prof. Vanwormhoudt niet willen vergeten voor de inspirerende gesprekken en voor de middelen waarover ik kon beschikken.

Tot slot zou ook Katrien willen bedanken voor de morele steun bij mijn onderzoek en voor haar geduld van de laatste maanden. Jaja, weldra zal ik mijn werkkamer opruimen.

Gent, 27 september 1992.

Inhoud

Tabellen

Afbeeldingen

Programma's

Notaties

H_B	Herbrandbasis
H_U	Herbranduniversum
S_a	Verzameling van alle atomen
S_b	Verzameling van alle borden
S_d	Verzameling van alle doelen
S_g	Verzameling van alle gebeurtenissen
S_I	Verzameling van alle interpretaties
S_P	Verzameling van alle programma's
S_s	Verzameling van alle sporen
S_t	Verzameling van alle termen
S_T	Verzameling van alle taken
S_θ	Verzameling met alle substituties
δ	Leeg doel
ϵ	Lege substitutie
λ	Leeg spoor

Niet alle werken over PROLOG maken gebruik van dezelfde terminologie. Om verwarring bij het lezen van dit proefschrift zoveel mogelijk uit te sluiten geven we hier een lijstje met de terminologie zoals ze hier gebruikt wordt.

Constante Dit kan zowel een symbolische constante zoals `a`, `<` of `wilfried` zijn. Gewoonlijk begint een symbolische constante niet met een hoofdletter. Een numerische constante is in ons geval een getal zoals `3` of `-5.3`.

Variabele Dit is een symbolische constante die met een hoofdletter begint zoals `X` of `Kind`.

Structuur Dit is een object bestaande uit een constante en een aantal argumenten. Voorbeelden zijn `f(1,2,3)` en `vader(karel)`. De argumenten kunnen willekeurige termen zijn. Men kan abstractie maken van de concrete argumenten door de notatie `f/3` en `vader/1` te gebruiken.

Atoom Dit object heeft dezelfde syntaxis als een structuur, maar kan bovendien waar of vals zijn. Voorbeelden zijn `even(2)` en `is_vader(piet)`. Een atoom kan gebruikt worden als vraag, als kop van een bepaling en als doel.

Lijst Dit is een opeenvolging van termen. Een niet-lege lijst is een object dat bestaat uit een kop en een staart en genoteerd wordt als `[kop|staart]`. De kop en de staart kunnen willekeurige termen zijn, inclusief lijsten. Voorbeelden van lijsten zijn `[]` (de lege lijst, dit is in feite een symbolische constante), `[1]`, `[1,2,3,4]` (een lijst met vier elementen).

Term Dit kan een constante, een structuur, een lijst of een variabele zijn.

Feit Dit is een atoom gevolgd door een punt. Hiermee drukt men uit dat het atoom waar is. Een voorbeeld is `even(0).`

Regel Een regel bestaat uit een kop en een romp, gescheiden door het symbool `:-` en gevolgd door een punt. De kop is waar indien de romp waar is. Een voorbeeld van een regel is `even(X) :- 0 is X mod 2.`

Bepaling Dit kan een feit of een regel zijn.

Procedure Dit is een verzameling van bepalingen die logisch bij elkaar horen omdat de naam en het aantal argumenten van de kop overeenstemt.

Woordenlijst

Bij de realisatie van dit proefschrift werd er een ernstige inspanning gedaan om zoveel mogelijk de Nederlandse terminologie te gebruiken. In de informatica is dit een groot probleem omdat de Nederlandse terminologie verre van gestandaardiseerd is en het vaak moeilijk is om een geschikte Nederlandse term te vinden die precies de inhoud van een begrip weergeeft.

Een drietal boeken zijn bijzonder behulpzaam geweest bij het opsporen van de Nederlandse terminologie.

(i) *Kluwer's woordenboek informatica Nederlands-Engels Engels-Nederlands*. Dit is een degelijk boek [Bak85] dat zich vooral richt op de meer algemene terminologie. Het bevat de vertaling van een 12 000 termen.

(ii) *Woordenboek automatisering* van Henk Van Biemond. Dit boek [Bie88] behandelt begrippen uit de telecommunicatie- en automatiseringswereld.

(iii) *Computer Jargon* van Hein van Steenis [vS91]. Dit boekje verklaart een 2500 vaak gebruikte computertermen.

(iv) Voor de Prologterminologie werd hoofdzakelijk een beroep gedaan op PROLOG, *beschrijving van de standaard* [CM87], een vertaling van [CM81]. In dit boek werd een aanzienlijke inspanning geleverd om een originele Nederlandse terminologie te gebruiken.

(v) Verder werd er ook nog gesteund op het boek *Programmeren in Prolog* door Henk Schotel [Sch87], en op *Inleiding in Prolog* door F.G. Hilgevoord en H.K.U. Wind [HW88].

Om de lezer een houvast te bieden bij het lezen van dit proefschrift wordt hier een vertalende woordenlijst ingelast.

Nederlandse term	**Engelse term**
aaneenschakeling	concatenation
achterwaartse communicatie	back communication
actief wachtend	polling
afspraak	rendez-vous
bediener	server

bepaling	clause
beperkt en-parallellisme	restricted and parallelism
bereik	scope
berging	store
bewerkingen van elders	remote operations
billijk	fair
boodschap	message
bord	blackboard
bordvergrendeling	blackboard lock
buffer met beperkte capaciteit	bounded buffer
cliënt	client
cliënt bediener	client-server
communicatie tussen taken	interprocess communication
communicatiemakelaar	message broker
compilator	compiler
compileren	to compile
concretisering	instantiation
correct	sound
corrupte wijzer	dangling pointer
deelbord	subblackboard
doel	goal
doelparallellisme	goal parallelism
het doorgeven van boodschappen	message passing
doorschrijven	to write through
doorvoercapaciteit	throughput
en-parallellisme	and parallelism
engagement	commit
evaluatieprogramma	benchmark program
faling	failure
fitting	socket
gastheer	host computer
geëngageerde keuze	committed choice
gebeurtenis	event
geheugensanering	garbage collection
genereer-en-controleer	generate and test
gespreid terugzoeken	distributed backtracking
herhaal-faallus	repeat-fail loop
hernoemen	to rename
hoop	heap
kanaal	channel
kennisbron	knowledge source

keuzedoel	choice goal
keuzepunt	choice point
knip	cut
kop	head
leesbewerking	get operation
leesgebeurtenis	get event
leesreductie	get reduction
meeluisteren	to snoop
meerdradig uitvoeringspakket	multi-thread package
menger	merger
multiprocessor met gemeenschappelijk geheugen	shared memory multiprocessor
multiprocessor met gespreid geheugen	distributed memory multiprocessor
of-parallellisme	or parallelism
omroepen	to broadcast
ontluizen	to debug
preëmptie	preemption
overlast	overhead
overloop	overflow
partnersysteem	buddy system
patstelling	deadlock
poort	port
prestatie	performance
procedureoproep van elders	remote procedure call
programmatekst	source code
programmatuurbouwkunde	software engineering
referentiepunt	benchmark
rekenblad	spreadsheet program
reserveren	to allocate
romp	body
schrijfbewerking	put operation
schrijfgebeurtenis	put event
schrijfreductie	put reduction
schrijfwachter	write guard
semafoor	semaphore
slagen	to succeed
sonde	probe
spoor	trace
stapel	stack
stapeloverloop	stack overflow

strijdig	unsatisfiable
stroom	stream
stuursysteem	run-time system
systeemelement	resource
taak	process
taakcreatie	spawn
taakparallellisme	process parallelism
telsemafoor	counting semaphore
terugcopiëren	to copy back
terugzoeken	to backtrack
toepassingsdomein	domain of discourse
toestand	status
tussenaankomsttijd	interarrival time
tussengeheugen	cache
uitgestelde evaluatie	lazy evaluation
uitvoeringsfout	run-time error
vastlopen	to deadlock
veeltal	tuple
veeltallenruimte	tuple space
vermenging	interleaving
verplaatsing	relocation
vertrouweling	trust
het vervullen van randvoorwaarden	constraint solving
verwezenlijkbaar	satisfiable
vlak	flat
volledig	complete
volledig geconcretiseerd	ground
vraag	initial goal
vrijgeven	to deallocate
wachter	guard
ware tijd	real-time
ware-tijdskern	real-time kernel
weerlegging	refutation
wekkertaak	time-out process
werkverdeler	scheduler
wijzer	pointer

Samenvatting

Sometimes when I have finished a book
I give a summary of the whole of it.

— ROBERT WILLIAM DALE, *Nine Lectures on Preaching* (1878).

Het doel van dit proefschrift is te demonstreren dat taakparallellisme en expliciete bordcommunicatie gebruikt kunnen worden om de taal PROLOG te parallelliseren met behoud van haar gebruikelijke semantiek.

De taal PROLOG werd hiertoe uitgebreid met een bord en de bijbehorende predicaten. Deze laten toe verscheidene onafhankelijk uitvoerende sequentiële Prologtaken te creëren en deze taken onderling te laten communiceren. Deze communicatie gebeurt uitsluitend via het bord. Deze nieuwe taal wordt MULTI-PROLOG genoemd en er wordt aangetoond dat de communicatieprimitieven van MULTI-PROLOG zeker even krachtig zijn als wat bij andere taakparallelle logische programmeertalen gebruikelijk is.

De semantiek van de communicatieprimitieven werd bestudeerd in het Prologkader. Een operationele en declaratieve semantiek worden voorgesteld en daaruit blijkt dat ondanks het feit dat de bordcommunicatie een neveneffect is voor een individuele taak en dus zichtbaar is in de semantiek van die taak, dit niet langer het geval is voor het totale programma. De semantiek van een Multi-Prologvraag is volledig vergelijkbaar met deze van een Prologvraag omdat ook communicatiegebeurtenissen tussen taken uiteindelijk unificeren met elkaar en hierdoor als het ware ophouden een neveneffect te zijn.

Het bord is gemeenschappelijk aan alle taken en hierdoor bijzonder kritisch bij de implementatie. Dank zij een zorgvuldig ontwerp kon er vermeden worden dat het de fatale flessehals werd voor het totale systeem. Dit toont aan dat een gecentraliseerde communicatiestructuur niet a priori gedoemd is om minderwaardig te zijn in vergelijking met een gedecentraliseerde structuur. Om dit alles te staven werd een implementatie gemaakt van de basisprimitieven van MULTI-PROLOG.

Er wordt aangetoond dat het voorgestelde communicatieparadigma ook als programmeertaal goed bruikbaar is. Een verzameling van klassieke voorbeelden, gaande van de dinerende filosofen tot een vluchtreservatiesysteem kun-

nen probleemloos geïmplementeerd worden. Hierbij hoeven er absoluut geen kunstgrepen gebruikt te worden. De gebruiksgemak van het bord speelt hierbij een niet-onbelangrijke rol.

Het besluit is dat een taakparallelle PROLOG met expliciete bordcommunicatie zeker een volwaardig alternatief vormt voor het scala van bestaande parallelle logische programmeertalen en bovendien efficënt geïmplementeerd kan worden.

1 Inleiding

Het laatste wat men ontdekt als men een boek schrijft is
hoe men had moeten beginnen.

— BLAISE PASCAL

In dit inleidend hoofdstuk wordt het kader geschetst waarin het Multi-Prologonderzoek heeft plaatsgevonden. Er wordt gestart met een overzicht van bestaande programmeertalen, en de plaats die PROLOG *als logische programmeertaal hierin inneemt. Dan volgt een schets van de inzichten in het hedendaagse onderzoek naar parallellisme. In een derde sectie wordt er een overzicht van bestaande parallelle logische programmeertalen gegeven. Dit brengt ons bij* MULTI-PROLOG*, het centrale onderwerp van dit proefschrift. Vervolgens wordt mijn thesis geponeerd, gevolgd door een overzicht van de originele bijdragen die voortvloeiden uit dit onderzoek. Dit hoofdstuk wordt besloten met een overzicht van de structuur van dit proefschrift.*

1.1 Logisch Programmeren

Programmeertalen kunnen ingedeeld worden op basis van het paradigma dat er ten grondslag aan ligt [GJ90]. In deze indeling wordt de grootste groep gevormd door de *imperatieve* of *procedurale talen*. Het berekeningsmodel is gesteund op de theorie van de *Turing machines*. Tot deze groep behoren talen zoals C, PASCAL, FORTRAN, en zelfs ASSEMBLER. Ondanks het feit dat ze uiterlijk nogal wat verschillen vertonen, worden de stapelgebaseerde talen zoals FORTH, ASYST en POSTSCRIPT ook tot de imperatieve talen gerekend.

De groep van imperatieve talen is de oudste omdat het berekeningsmodel zo nauw aansluit bij de architectuur van de eerste computers en bij de structuur van algoritmen. Hun evolutie heeft echter niet stilgestaan. Uitgaande van FORTRAN in de jaren vijftig heeft men gaandeweg gegevensstructuren en

controlestructuren aan de talen toegevoegd. Tegenwoordig zijn de meeste van deze talen zodanig naar elkaar toegegroeid dat de keuze van een taal eerder gemaakt wordt op basis van argumenten die vreemd zijn aan de taal als dusdanig (beschikbaarheid van bibliotheken, krachtige ontwikkelomgeving, beschikbaarheid op een bepaald platform, verspreiding). Een stamboom van de meest courante imperatieve talen vindt men in [DVC90].

De *objectgeoriënteerde talen* zijn gebaseerd op het *framemodel*, ontwikkeld door Marvin Minsky [Min74]. In deze talen concentreert men zich hoofdzakelijk op het ontwerp van een (hiërarchische) gegevensstructuur ([Str88b, GL88]). Hierbij maakt men gebruik van abstracte gegevenstypes die een combinatie zijn van gegevens in de klassieke zin en bewerkingen. Een gegevensstructuur van dit type wordt een *object* genoemd. Een dergelijk object kan enkel gemanipuleerd worden door een beperkt aantal routines die vooraf expliciet gespecificeerd werden. Om het dupliceren van programmatekst zoveel mogelijk tegen te gaan kunnen objecten eigenschappen (gegevensdefinitie, routines) van elkaar overerven. Het gevolg hiervan is dat de programmatekst beter gestructureerd en dus beter te beheersen is.

De simulatietaal SIMULA wordt beschouwd als de voorloper van de hedendaagse object-georiënteerde programmeertalen waarvan de best gekende wellicht SMALLTALK is. In de tachtiger jaren vond een aantal van de principes van objectoriëntatie ingang in de procedurale talen. Tal van objectgeoriënteerde uitbreidingen zoals C++ [Str88b, Str88a], OBJECT C of TURBO-PASCAL V5.0 zijn hiervan het resultaat. Deze talen zijn niet zo zuiver objectgeoriënteerd als SMALLTALK, maar bieden wel het voordeel van de bijkomende structuur, en zijn bovendien goed bruikbaar. Het succes van de objectgeoriënteerde versies van C hangt nauw samen met de opkomst van X-WINDOW en MS-WINDOWS die ook objectgeoriënteerd zijn.

De *functionele* of *applicatieve talen*, een groep van talen geïnspireerd op de theorie van de primitief recursieve functies [Rev88, Bar85], beschouwen alle objecten uit de taal als functies, d.w.z. per definitie vrij van neveneffecten. De best gekende functionele taal is ongetwijfeld LISP. Functionele talen (vaak LISP dialecten) worden courant gebruikt als onderdeel van grote toepassingen. Denken we maar aan MATHEMATICA, REDUCE, AUTOLISP (bij AUTOCAD), EMACS, enz. In dit kader worden ze gebruikt om het programmapakket aan te passen aan persoonlijke noden en smaak. Een wellicht beter gekende maar tevens zwakkere vorm van functioneel programmeren zijn de rekenbladen. Deze bestaan essentieel uit een netwerk van elkaar oproepende functies, die gegroepeerd worden in een cellenmatrix [Bai87, Mye90].

De *logische programmeertalen* steunen op de wiskundige logica [DB89e]. Door zijn declaratieve natuur vereist een logische programmeertaal een bijzondere manier van programmeren. Net zoals dit gebruikelijk is in de wiskundige

logica, begint men met een aantal eigenschappen van het te beschrijven systeem op te sommen. Nadien wordt deze informatie gebruikt om vragen over het systeem te beantwoorden. Dit vereist een inferentiemotor om alle stukjes informatie samen te voegen tot een antwoord of 'bewijs'.

Kenmerkend voor de logische programmeertalen is dat de betekenis van een programma grotendeels onafhankelijk is van de gekozen inferentiemotor [New90]. Het is immers de logica die de betekenis van het programma vastlegt. Het zoeken naar een geschikt antwoord kan dan ook op verscheidene manieren gebeuren (diepte-eerst, breedte-eerst, heuristisch,...). De twee belangrijke kwaliteitseisen van een inferentiemotor zijn dat hij (i) precies dezelfde oplossingenverzameling genereert als voorgeschreven door de logica, en (ii) aan minimale eisen qua efficiëntie voldoet. Gezien de *partieel onbeslisbare* natuur van de predicatenlogica is het moeilijk om aan deze beide eisen te voldoen. Daarom legt men beperkingen op zoals het gebruik van Hornbepalingen in plaats van het gebruik van algemene logische uitdrukkingen, en laat men zelfs oogluikend toe dat sommige oplossingen soms niet gevonden worden door bepaalde inferentiemotoren [DB91b]. Dit is een gevolg van het compromis dat dient gemaakt te worden tussen enerzijds efficëntie en anderzijds volledigheid. De logische programmeertaal met de meeste bekendheid is PROLOG.

Naast de vier besproken families van talen is er ook nog een aantal in dit kader minder belangrijke talen zoals talen voor de verwerking van karakterrijen (SNOBOL, AWK, ICON), documentbeschrijvingstalen (TEX, POSTSCRIPT), gegevensbanktalen (SQL).

Naargelang van het toepassingsgebied kan de implementatievorm van een programmeertaal variëren. Een taal kan gecompileerd worden naar machinetaal, naar een intermediaire taal, of naar een andere hoge-niveautaal, of kan geïnterpreteerd of vertolkt worden. De geïnterpreteerde talen worden vaak interactief aangewend.

De vier hiervoor beschreven taalfamilies hebben elk hun sterke en zwakke kanten. Daarom is men reeds geruime tijd op zoek naar een manier om de vier paradigma's te verenigen in één enkele taal, die dan het beste van de vier werelden te bieden zou hebben. Men blijkt echter nog een heel eind van die ultieme doelstelling verwijderd te zijn. In afwachting hiervan doet men wel pogingen om sommige paradigma's te combineren (imperatief en objectgeoriënteerd blijkt een succes te zijn; cfr. C++). Pogingen om functionele en objectgeoriënteerde elementen toe te voegen aan het logisch programmeren zijn er ook reeds geweest [Kos87, GM, Eli92], maar met weinig navolging.

Dit proefschrift maakt gebruik van PROLOG, de op dit ogenblik meest verspreide logische programmeertaal. PROLOG is een samentrekking van *Programmation en logique*. De eerste implementatie van de taal PROLOG is verbonden met de naam van A. Colmerauer (1972). Zij was het levende bewijs dat logica

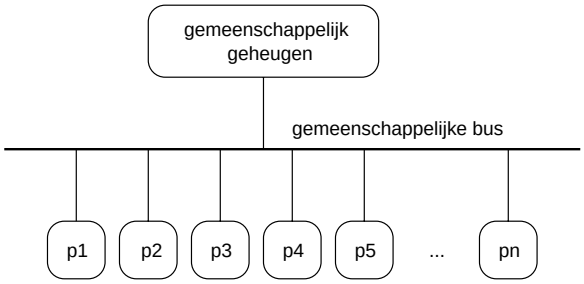
bruikbaar is als programmeertaal mits toepassing van de gepaste berekeningsregel, iets wat door Kowalski later kernachtig werd uitdrukt als: *algoritme* = *logica* + *controle* [Kow79]. De eerste implementaties maakten rechtstreeks gebruik van programmatekstvertolkers om een Prologprogramma uit te voeren. Dit maakte de uitvoering van een programma traag. Het is pas een vijftal jaar later in 1977 dat door D.H.D. Warren een eerste Prologcompilator ontworpen werd. De compilator genereerde code voor de *Warren Abstract Machine* (WAM) [War83]. De gegenereerde code moest echter nog steeds door een vertolker uitgevoerd worden. Later werden er compilators ontwikkeld die rechtstreeks machinetaal of C genereerden. Niettegenstaande dit is de WAM nog steeds actueel en worden de principes nog veel toegepast, zij het in een aangepaste vorm. Ten onrechte hoort men soms nog beweren dat de uitvoering van logische programma's traag is. De compilatietechniek en de stuursystemen zijn immers zodanig geoptimaliseerd dat de snelheid vergelijkbaar wordt met die van b.v. C-programma's [VR90, GDBD92].

Ondanks zijn vele unieke kenmerken is PROLOG er nooit echt in geslaagd door te breken als algemene programmeertaal bij het grote publiek. Misschien kan standaardisering in de toekomst het tij doen keren [pro91]. Het staat echter boven elke twijfel dat moderne Prologimplementaties alles hebben wat van een moderne taal mag verwacht worden.

Van de lezer van dit proefschrift wordt verwacht dat hij/zij een elementaire kennis van de sequentiële taal PROLOG heeft. Het onderscheid tussen een feit en een regel, en enkele noties over de berekeningsregel (bepalingen worden sequentieel afgelopen, conjuncties worden van links naar rechts geëvalueerd) zouden voldoende moeten zijn om het grootste deel van deze tekst te kunnen begrijpen.

Voor hen die zich eerst willen verdiepen in de taal kunnen de volgende boeken aangeprezen worden:

- W.F. Clocksin and C.S. Mellish. *Programming in Prolog* [CM81]: min of meer hét standaardwerk over PROLOG (1981). Er bestaat ook een vertaling in het Nederlands [CM87]. Als eerste kennismaking moet het echter onderdoen voor
- E. Shapiro and L. Sterling. *The Art of Prolog* [SS86]: een degelijk boek voor de beginner (1987).
- R. O'Keefe. *The Craft of Prolog* [O'K90]: Een excellent en recent boek voor de meer gevorderde Prologprogrammeur (1990).



Afbeelding 1.1 Multiprocessor met gemeenschappelijk geheugen.

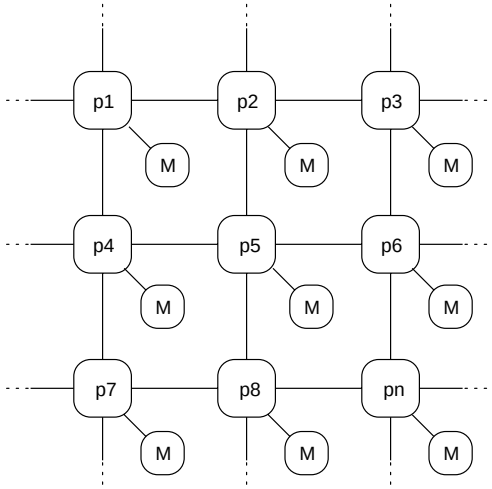
1.2 Parallellisme

Een groot deel van het hedendaagse onderzoek naar programmeertalen betreft het onderzoek naar parallellisme. Dit hangt ongetwijfeld samen met het feit dat multiprocessoren tegenwoordig voor een redelijke prijs verkrijgbaar zijn in tal van configuraties, maar dat de programmatuur voor deze multiprocessoren nauwelijks in staat is deze krachtige computersystemen op een verantwoorde manier te benutten. Het produceren van multiprocessorhardware is blijkbaar veel gemakkelijker dan het programmeren ervan.

Multiprocessoren kunnen op basis van de plaats en toegankelijkheid van hun geheugen ingedeeld worden in twee categoriën: deze met gemeenschappelijk geheugen, in hun eenvoudigste vorm geschetst in afbeelding 1.1, en deze met gespreid geheugen uit afbeelding 1.2. Beide hebben hun voor- en nadelen die hierna in het kort besproken worden [VC91].

In een multiprocessor met gemeenschappelijk geheugen zullen de processoren meestal niet rechtstreeks met elkaar communiceren, maar via het gemeenschappelijk geheugen. Dit houdt in dat een processor ergens in het gemeenschappelijk geheugen een bericht achterlaat dat later door een andere processor kan opgehaald worden. Het feit dat twee processoren dezelfde geheugencel simultaan kunnen nodig hebben vereist dat dit op een gecontroleerde manier gebeurt. De gemeenschappelijke bus zal ervoor zorgen dat op het niveau van de geheugentoegangen alle aanvragen sequentieel afgewerkt worden. Deze vorm van synchronisatie is onmisbaar, maar vaak ontoereikend op het programmaniveau. Daar moet men over de mogelijkheid kunnen beschikken om gedurende een bepaald tijdsinterval exclusieve toegang tot bepaalde geheugencellen te verwerven. Het beschermen van gemeenschappelijk gegevens tegen ongeoorloofde toegang is een belangrijk aspect van de programma's voor dit type van multiprocessor.

In een gespreide architectuur (zie afbeelding 1.2) is het geheugen niet lan-



Afbeelding 1.2 Multiprocessor met gespreid geheugen.

ger gemeenschappelijk, maar verdeeld over de processoren. Daar doet het dienst als lokaal geheugen. De processoren communiceren nu via speciale communicatiekanalen (de technologie die hierbij gebruikt wordt kan variëren van speciale verbindingen tussen twee punten tot complete communicatienetwerken die zichzelf automatisch configureren). In zijn meest eenvoudige vorm heeft een communicatiekanaal geen geheugen. Dit wil zeggen dat de zender van een boodschap zal moeten wachten tot wanneer de ontvanger in staat is de boodschap te ontvangen. De communicatie is met andere woorden *synchroon*. Voorbeelden van multiprocessoren met gespreid geheugen zijn de INMOS TRANSPUTER en de INTEL HYPERKUBUS. Het aantal fysische kanalen dat met een processor verbonden is wordt beperkt door de hardware. Bij een TRANSPUTER is dit vier, bij een processor uit de HYPERKUBUS is dit de dimensie van de HYPERKUBUS. Van zodra er in het netwerk twee processoren meer dan het aantal kanalen per processor zijn, zal er een route doorheen het netwerk moeten uitgestippeld worden en zullen naburige netwerkknopen moeten tussenkomen bij het tot stand komen van de communicatie. Dit veroorzaakt onvermijdelijk vertraging bij de communicatie.

De meest voorkomende gespreide systemen zijn ongetwijfeld de in een lokaal netwerk gekoppelde werkstations. Deze communiceren onderling door middel van boodschappen die via het lokale netwerk (op basis van b.v. Ethernet) verstuurd en beantwoord worden. Een lokaal netwerk als communicatienetwerk is echter traag omdat er per processor slechtséén fysisch communicatiekanaal is (de verbinding met het netwerk), en omdat het netwerk een

busstructuur heeft, met alle nadelige gevolgen van dien. Speciale communicatiekanalen kunnen een debiet tot meer dan 2 Mbyte/s tussen processoren halen, terwijl een lokaal netwerk zelden 10% hiervan haalt [Wis92].

Als we nu beide architecturen tegen elkaar afwegen kunnen we stellen dat het belangrijkste voordeel van een multiprocessor met gemeenschappelijk geheugen zijn eenvoud en zijn verwantschap met monoprocessoren is. De beschikbaarheid van een globale toestand vergemakkelijkt het ontwikkelen van toepassingen. Mede hierdoor kunnen de courant gebruikte programmeertalen meestal zonder veel aanpassingen gebruikt worden op dergelijke machines.

Het belangrijkste nadeel van deze architectuur is de afhankelijkheid van de globale bus. Deze bus beperkt doorgaans de maximale grootte van de multiprocessor, zowel elektrisch als voor wat betreft de capaciteit. Om dit nadeel weg te werken zijn sommige architecturen voorzien van speciale communicatiehardware, een hoeveelheid lokaal geheugen per processor, en steeds maar groter wordende tussengeheugens. Deze laatste gedragen zich ook als een soort van lokaal geheugen.

Het voornaamste voordeel van de multiprocessoren met gespreid geheugen is hun groeipotentieel. In tegenstelling met de multiprocessoren met gemeenschappelijk geheugen groeit de communicatiebandbreedte (het aantal kanalen) met het aantal processoren. Dit neemt echter niet weg dat de diameter van het communicatienetwerk (d.w.z. het aantal processoren dat in het slechtste geval moet inspringen om een communicatie tussen twee processoren tot stand te brengen) wel groeit met het aantal processoren. De communicatiebandbreedte groeit dus nog steeds minder sterk dan het aantal processoren. De grens van de bruikbaarheid ligt echter een stuk verder dan bij multiprocessoren met gemeenschappelijk geheugen. Bovendien kan men trachten optimaal gebruik te maken van de beschikbare bandbreedte door taken die intensief met elkaar communiceren toe te wijzen aan naburige processoren.

Het belangrijkste nadeel van multiprocessoren met gespreid geheugen is hun gebrek aan flexibiliteit. Tenzij men programma's in verscheidene knopen van het netwerk opslaat is het niet mogelijk taken op efficënte wijze tussen processoren te laten migreren. Bovendien moeten de taken in een gespreid programma zwak gekoppeld zijn en mogen ze geen gebruik maken van gemeenschappelijke gegevens. Dit maakt het ontwerp van dergelijke programma's moeilijker (geen gemeenschappelijke klok of toestand, nauwelijks observatie mogelijk zonder verstoring).

Samenvattend kan gesteld worden dat het maken van een toepassing voor een multiprocessor met gespreid geheugen moeilijker is omdat er met meer aspecten rekening moet gehouden worden. Indien er echter een geschikte decompositie gevonden wordt, zal het resulterende programma wellicht efficiënter uitgevoerd kunnen worden en bovendien gemakkelijker kunnen uit-

gebreid worden dan het geval zou zijn bij een multiprocessor met gemeenschappelijk geheugen.

Het maken van een toepassing voor een multiprocessor met gemeenschappelijk geheugen is een heel stuk eenvoudiger omdat de observeerbaarheid van de architectuur zoveel beter is. Toepassingen kunnen sneller en met minder moeite ontwikkeld worden, maar zijn wel beperkt in hun uitbreidingsmogelijkheden.

Sinds een aantal jaren worden er pogingen ondernomen om het programmeergemak van de multiprocessoren met gemeenschappelijk geheugen te combineren met de uitbreidingsmogelijkheden van de multiprocessoren met gespreid geheugen. Het resultaat hiervan is het zogenaamde *gespreide gemeenschappelijke geheugen*. Dit is een simulatie van gemeenschappelijk geheugen op een multiprocessor met gespreid geheugen [NL91]. Het grootste probleem bij de implementatie is het kunnen garanderen van de coherentie en een aanvaardbare efficiëntie. Dit probleem is sterk verwant met het probleem van de coherentie bij tussengeheugens. De voordelen van het gespreide gemeenschappelijke geheugen is het gemak van programmeren, de onafhankelijkheid van de onderliggende multiprocessorarchitectuur, de overdraagbaarheid en de schaalbaarheid. Het nadeel is het verlies aan prestatie in vergelijking met systemen die met fysisch gemeenschappelijk geheugen uitgerust zijn.

1.3 Parallelle Logische Programmeertalen

Zoals eerder aangehaald in deze inleiding staat de betekenis van een logisch programma los van zijn uitvoering ([New90]). Door het feit dat de taal geen particuliere manier van uitvoeren dicteert (zoals dit wel het geval is in sequentiële imperatieve programmeertalen: de volgorde is strikt vastgelegd), kan een logisch programma op verscheidene manieren uitgevoerd worden. In het bijzonder is men vrij te kiezen voor een sequentële of een parallelle uitvoering van het programma [DB89d].

Er volgt nu een overzicht van manieren waarop men in het verleden gepoogd heeft logische programmeertalen op parallelle manier uit te voeren. Een uitstekend overzicht van hedendaagse parallelle logische programmeertalen wordt geboden door [Con87, Sha89, KW92].

1.3.1 Unificatieparallellisme

Een aanzienlijk deel van de uitvoeringstijd van een logisch programma wordt gebruikt voor unificatie. Bij de unificatie van variabelen en constanten valt er niet veel parallellisme te bespeuren, maar bij de unificatie van structuren en lijsten is dit wel het geval. In het geval van een lijst zou men de kop en de staart van de lijst simultaan kunnen unificeren, en in het geval van een struc-

```
?- f(X,X) = f(a,b).
```

Programmafragment 1.1 Unificatieparallellisme moet rekening houden met afhankelijkheden via gemeenschappelijke variabelen. De unificaties `X=a` en `X=b` mogen niet zomaar parallel uitgevoerd worden zonder te synchroniseren op `X`.

```
p(f(a)).
p(f(b)).
p(f(X)) :- X=c.
```

Programmafragment 1.2 Bij of-parallellisme kan de vraag `?- p(f(b)).` beantwoord worden door de drie bepalingen van dit programmafragment simultaan te proberen.

tuur zou men de argumenten van de structuur simultaan kunnen unificeren. In de meeste gevallen wordt dit echter bemoeilijkt door gemeenschappelijke variabelen. Een variabele mag tijdens een unificatietaak slechts één waarde toegewezen krijgen. Er moet dus vermeden worden dat twee unificatietaken tegelijkertijd een waarde zouden toekennen aan een variabele (zie programmafragment 1.1). Dit vereist dus synchronisatie en vertraagt aanzienlijk de unificatieroutine. Unificatieparallellisme kan wellicht een versnelling leveren, maar dit zal altijd beperkt blijven [DKM84] omdat er relatief veel gecommuniceerd moet worden. Een statische analyse van een programma kan in sommige gevallen uitsluitsel geven over het al dan niet gebruiken van parallelle unificatie. Deze vorm van parallellisme wordt *fijnkorrelig* genoemd.

1.3.2 Of-parallellisme

Met de of-keuze wordt de keuze tussen de bepalingen van een procedure bedoeld. Om een doel te kunnen evalueren moet een bepaling geselecteerd worden. In PROLOG gebeurt dit door de lijst met bepalingen op een sequentiële manier af te lopen totdat de geschikte bepaling gevonden wordt. In plaats van deze lijst met bepalingen sequentieel af te lopen zou men echter ook alle bepalingen simultaan kunnen uitvoeren (zie programmafragment 1.2). De bepaling die als eerste met succes termineert zal dan de eerste oplossing genereren.

Deze vorm van parallellisme is echter moeilijk efficiënt te implementeren. Vooreerst moeten alle parallelle taken kunnen beschikken over een volwaardige verzameling van gegevensstructuren (omgeving, stapel, hoop, ...), en bovendien toegang hebben tot de gegevensstructuren van de oproeper. Er moet anderzijds over gewaakt worden dat de parallelle taken de gemeenschappelijke gegevensstructuren van oproepende bepaling niet kunnen aanpassen omdat dit conflicten kan veroorzaken met de andere parallelle taken. De gegevensstructuren van de oproeper moeten dus beschermd worden tegen overschrijvingen. Niettegenstaande deze moeilijkheden is er toch een aantal parallelle

```
kop :- wachter | vertrouweling.
```

Programmafragment 1.3 Bepaling met een wachter.

logische programmeertalen die of-parallellisme ondersteunen [AR90, HJ90].

De moeilijkheden bij de implementatie van of-parallellisme kunnen gedeeltelijk uit de weg gegaan worden door niet PROLOG, maar een zogenaamde geëngageerde-keuzetaal parallel uit te voeren. Deze talen genereren slechts één oplossing per doel. Een bepaling bestaat nu uit een wachter en een vertrouweling, gescheiden door het engagementssymbool (|) (zie programmafragment 1.3). De wachter groepeert de doelen die instaan voor de selectie van de bepaling. De overige doelen van de bepaling worden gegroepeerd in de zogenaamde vertrouweling. De wachter is in feite een uitbreiding van de kop. De unificatie met de kop en de evaluatie van de wachter zijn samen verantwoordelijk voor de selectie van een bepaling. De koppen en de wachters van alle bepalingen van een procedure kunnen simultaan geëvalueerd worden. De bepaling waarvan deze evaluatie als eerste slaagt, wordt geselecteerd, en de andere bepalingen worden verder genegeerd. De vertrouweling wordt pas geëvalueerd nadat een bepaling geselecteerd werd.

Een wachter wordt veilig genoemd indien er geen toewijzingen aan variabelen van de oproeper gebeuren. Dit impliceert dat de taak die de wachter evalueert de gegevensstructuren van de oproeper niet zal kunnen aanpassen. De taak die de geëngageerde keuze als eerste uitvoert krijgt de toelating de corresponderende vertrouweling uit te voeren. De taken die de wachters van de alternatieve bepalingen evalueren worden gestopt waardoor de vertrouweling dan naar believen de gegevensstructuren van de oproeper kan veranderen.

In dit verband wordt er ook van vlakke wachters gesproken. Een wachter is vlak indien hij enkel bestaat uit ingebouwde niet-recursieve doelen. Hierdoor wordt verhinderd dat er vernestelde wachters[1] moeten geëvalueerd worden. Door het feit dat de semantiek van de toegelaten ingebouwde predicaten gekend is, wordt het gemakkelijker om te optimaliseren en om te controleren of een wachter veilig is. Vlakheid alleen impliceert echter zeker geen veiligheid [New90].

De best gekende geëngageerde-keuzetalen zijn PARLOG [Gre87, Rin88], CONCURRENT PROLOG [HCF84, Sha86, AS88], GUARDED HORN CLAUSES [Ued85]), P-PROLOG [Yan87]). Deze talen worden onder andere gebruikt voor systeemtoepassingen en minder voor toepassingen uit de AI-sfeer. In de literatuur wordt vermelding gemaakt van verscheidene pogingen om een gedeelte van een besturingssysteem aan de hand van deze talen te implementeren [Sha87a].

[1] Deze kunnen veroorzaakt worden door een gebruikersgedefinieerd predicaat in een wachter op te roepen.

```
p(X,Y) :- p1(X), p2(Y), p3(Z).

?- p(V,V).
```

Programmafragment 1.4 Afhankelijkheden via gemeenschappelijke variabelen kunnen de volledig parallelle uitvoering van een conjunctie verhinderen.

1.3.3 En-parallellisme

Deze vorm van parallellisme is toegelaten op basis van het feit dat in de logica een conjunctie in willekeurige volgorde mag geëvalueerd worden, en dus ook parallel. In een logische programmeertaal stelt zich echter nu ook weer het probleem van de variabelen die gemeenschappelijk zijn (cfr. unificatieparallellisme). Ook nu zorgen zij ervoor dat men niet zomaar alle conjuncties kan parallelliseren. Bovendien kan een doel verscheidene oplossingen opleveren. Al deze oplossingen moeten uiteindelijk een kans krijgen. Dit impliceert dat er een zekere vorm van terugzoeken bewaard moet blijven tussen doelen die gemeenschappelijke variabelen bezitten.

Daarnaast is er ook nog het probleem van de predicaten die eisen dat sommige van de argumenten een waarde hebben (b.v. de relationele operator 'kleiner dan'). Zolang één van beide argumenten een variabele is, zal dit predicaat een uitvoeringsfout geven. Dit is zeker niet gewenst en daarom zal dit predicaat moeten kunnen blokkeren totdat zijn argumenten voldoende geconcretiseerd zijn. Dit is opnieuw een vorm van overlast. Samenvattend kan gesteld worden dat een en-parallel systeem in staat moet zijn (i) om de uitvoering van een taak stil te leggen totdat aan bepaalde voorwaarden voldaan is (b.v. totdat een variabele voldoende geconcretiseerd is), en (ii) om conjuncties met gemeenschappelijke variabelen sequentieel uit te voeren.

De detectie van gemeenschappelijke variabelen is echter niet triviaal. In programmafragment 1.4 zal `p3(Z)` onafhankelijk van `p1(X)` en `p2(Y)` kunnen uitgevoerd worden. Daarentegen kunnen `p1(X)` en `p2(Y)` wel degelijk afhankelijk zijn indien `p` opgeroepen wordt als `?- p(V,V)`.

Twee belangrijke en-parallelle stromingen zijn enerzijds het beperkte en-parallellisme waarbij enkel onafhankelijke factoren parallel uitgevoerd worden [Her86, Deg87], en stroomgebaseerd en-parallellisme waarbij de afhankelijke factoren gecatalogeerd worden als producenten en consumenten en waarbij de producenten vóór de consumenten uitgevoerd worden [Con87] en hierbij synchroniseren op de gemeenschappelijke variabelen.

1.3.4 En/of-parallellisme

Dit is een combinatie van de beide voorgaande. Doelen in een conjunctie worden parallel uitgevoerd en elk individueel doel zal aanleiding geven tot

een verzameling van taken die de verschillende bepalingen evalueren. Het is duidelijk dat deze aanpak een ware explosie van taken veroorzaakt. Deze aanpak exploiteert anderzijds wel het maximum aan parallellisme in de taal. Deze en de twee vorige vormen van parallellisme worden ook doelparallellisme genoemd [SP91] omdat zij zich situeren op het niveau van de doelen.

1.3.5 Taakparallellisme

Terwijl het doelparallellisme ontspruit aan de kenmerken van het resolutiemechanisme [DB90c] voor het evalueren van een logisch programma, wordt taakparallellisme volledig gecontroleerd door de programmeur. Hij beslist hoe en wanneer gebruik te maken van parallellisme. Dit gebeurt door expliciet sequentiële taken te creëren en naderhand deze taken expliciet met elkaar te doen communiceren. Deze aanpak maakt een precieze controle van het parallellisme mogelijk. Kenmerkend voor taakparallellisme is dat het eerder ingegeven wordt door het soort van toepassing dan door de taal. Taakparallellisme wordt *grofkorrelig* genoemd.

De programmeertaal die in dit proefschrift ontworpen en geëvalueerd wordt steunt volledig op het zogenaamde gebruikersgespecificeerd parallellisme of taakparallellisme. De keuze voor deze vorm van parallellisme werd ingegeven door de volgende bedenkingen.

- Het taakparallellisme heeft als belangrijk voordeel dat men zich geen zorgen hoeft te maken over de problemen die veroorzaakt worden door de gemeenschappelijke variabelen. Dit heeft als gevolg dat de inferentiemotor een stuk eenvoudiger zal zijn (het gebruik van cactusstapels, meervoudige omgevingen, opsporing van afhankelijkheden e.d.m. kunnen achterwege gelaten worden), en hierdoor aan efficiëntie zal winnen.

- Het taakparallellisme biedt een goede controle over het parallellisme, hetgeen het mogelijk maakt het parallellisme heel nauwkeurig te doseren op die plaatsen waar men het nodig acht. Toegegeven dat dit de zoveelste operationele component in een declaratieve programmeertaal is, maar anderzijds wordt men door middel van de taakinterpretatie [Sha86] van het doelparallellisme ook al met heel wat meer operationele aspecten geconfronteerd dan men eigenlijk zou wensen. Bovendien kan men dit ook positief benaderen door te stellen dat dit misschien een bijdrage kan leveren tot de integratie van het imperatief en logisch programmeren. Voorwaarde hierbij is natuurlijk dat de operationele component de logische component niet te veel verstoort.

- Doordat taakparallellisme gebruikersgespecificeerd is, bestaat er geen impliciet parallellisme in de taal en blijft het karakter van de onderliggende

taal bewaard. Door een bestaande vertrouwde sequentiële taal als substraat voor de parallelle uitbreiding te kiezen kan men vrij pijnloos overstappen van de sequentiële naar de parallelle versie van de programmeertaal. Het volstaat immers om een beperkt aantal nieuwe primitieven te leren gebruiken. Dit in tegenstelling tot de andere parallelle logische programmeertalen die in feite een nieuwe manier van denken en programmeren vergen.

In vergelijking met doelparallellisme is er nauwelijks onderzoek verricht naar taakparallellisme in logische programmeertalen. Erger nog, taakparallellisme wordt doorgaans als minderwaardig afgedaan. Slechts een beperkt aantal realisaties zag het daglicht. De voornaamste zijn CS-PROLOG [FK89], PMS-PROLOG [WJH92], DELTA-PROLOG [PMCA88], MB-PROLOG [Wis92], SHARED PROLOG [BC91].

Met dit pleidooi voor gebruikersgespecificeerd parallellisme willen wij echter geenszins suggereren dat deze vorm van parallellisme nu werkelijk alle problemen uit de weg ruimt. Het staat boven elke twijfel dat de exploitatie van doelparallellisme van in het begin krachtiger en theoretisch beter onderbouwd is omdat het mede ondersteund wordt door het refutatiealgoritme. Voorlopig blijkt het echter niet de meest renderende aanpak te zijn [Den85].

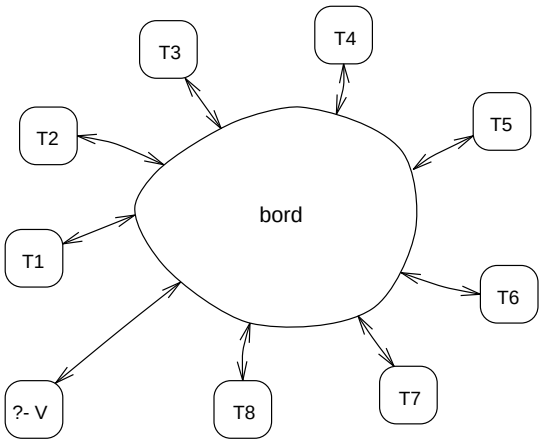
1.4 De Taal Multi-Prolog

MULTI-PROLOG is een parallelle logische programmeertaal die gebaseerd is op het gebruik van taakparallellisme en expliciete communicatie. Zoals de naam reeds doet vermoeden betreft het een uitbreiding van de sequentiële taal PROLOG.

Alle communicatie in MULTI-PROLOG is expliciet en *globaal*. De ontvanger van een boodschap wordt niet geëxpliciteerd en is zelf verantwoordelijk voor de identificatie van de boodschappen die voor hem bestemd zijn. Alle boodschappen worden centraal bijgehouden op een zogenaamd *bord*.

De generische structuur van een Multi-Prologprogramma tijdens de uitvoering wordt weergegeven in afbeelding 1.3. Het bord is de enige vorm van communicatie tussen de taken. De taken T1...T9 kunnen daardoor enkel met het bord communiceren.

Het gebruik van een bord als communicatiestructuur tussen taken kan het best vergeleken worden met het verloop van een werkvergadering waarbij een schrijfbord gebruikt wordt. De vergadering wordt geopend door de voorzitter die een probleem te berde brengt. Nadien begint de discussie tussen de deelnemers aan de vergadering. Als een deelnemer iets te zeggen heeft gaat hij naar het bord en verduidelijkt zijn standpunt door dit op het bord neer te schrijven. Andere deelnemers kunnen op deze informatie reageren door in-



Afbeelding 1.3 Structuur van een Multi-Prologprogramma in uitvoering.

formatie toe te voegen, stukken van het bord af te vegen, verduidelijking te vragen, en dergelijke meer. Een deelnemer zal enkel reageren op informatie waarover hij iets zinvols kan vertellen. Zoniet wacht hij geduldig totdat de discussie voor hem opnieuw interessant wordt. Van zodra de voorzitter een bevredigende oplossing voor zijn probleem heeft wordt de vergadering gesloten en verlaat de voorzitter het lokaal. Niets belet echter dat de andere deelnemers nog een tijdje over het probleem blijven doorbomen. Zij zullen de oplossing van de voorzitter echter niet meer kunnen beïnvloeden. Deze metafoor beschrijft redelijk nauwkeurig de uitvoering van een Multi-Prologprogramma.

De keuze voor een bord (d.w.z. een centrale communicatiestructuur) wordt als volgt gemotiveerd. De evolutie van de parallelle logische programmeertalen is in belangrijke mate beïnvloed geweest door het gespreid berekeningsmodel. De meeste systemen gebruiken kanalen om te communiceren. Men zou ze kunnen bestempelen als gespreide logische programmeertalen. Sommige gaan zelfs zo ver om gespreid terugzoeken te implementeren (DELTA-PROLOG [PMCA86, PMCA88], CS-PROLOG [FF92]). Merkwaardig genoeg werd het merendeel van deze logische programmeertalen eerst op een multiprocessor met gemeenschappelijke geheugen geïmplementeerd. De reden hiervoor zal ongetwijfeld met het gemak van implementatie te maken hebben [Lev86, LK88]. Deze talen maken echter geen gebruik van de grootste kracht van een multiprocessor met gemeenschappelijk geheugen, namelijk de mogelijkheid om een gegevensstructuur nagenoeg simultaan ter beschikking te stellen van verscheidene taken.

In MULTI-PROLOG wordt er een ernstige poging gedaan om een gemeen-

schappelijke gegevensstructuur op een gecontroleerde manier beschikbaar te stellen voor de taken. Het resultaat hiervan is het *bord*, een gegevensstructuur die door alle taken gebruikt kan worden om gegevens op te slaan en te raadplegen. Dit bord heeft een aantal interessante kenmerken.

- De gegevens op het bord zijn voor iedereen zichtbaar. Een bord geeft ons dus de mogelijkheid om een bericht niet enkel te verzenden naaréén taak, maar ook om een bericht om te roepen naar verscheidene taken. Een gegeven kan door eender welke taak gelezen worden, waardoor het bord ook gebruikt kan worden voor communicatie tussen taken. Het is echter ook mogelijk dat een taak een bericht leest, en meteen nadien het bericht van het bord afveegt (destructief lezen).
- Het bord is persistent. Dit betekent dat het gegevens kan opslaan voor een bepaalde tijd en dat de communicatie niet *synchroon* hoeft te zijn.
- Logisch gezien is het bord een lijst met gegevens in de volgorde van hun aankomst. Het houdt een soort van globale toestand van de communicatie bij. De beschikbaarheid van deze toestand is interessant bij de stapsgewijze ontwikkeling van programma's.

1.5 Mijn Thesis

Mijn thesis luidt als volgt:

"Taakparallellisme en expliciete bordcommunicatie zijn een *elegante, logisch verantwoorde, efficiënte* en *algemeen bruikbare uitbreiding* voor een sequentiële logische programmeertaal".

De uitbreiding is *elegant*.

- Het aantal benodigde primitieven is minimaal. Een primitief om een taak te creëren, een primitief om een gegeven op het bord te plaatsen en een klein aantal primitieven om gegevens van het bord te lezen.
- Alle synchronisatie gebeurt onzichtbaar door de bordprogrammatuur. De gebruiker kan hierdoor op een hoger niveau aan communicatie doen. Logisch gedraagt het bord zich als een lijst van gegevens in volgorde van aankomst.
- Het bord als een chronologische lijst van gegevens is vrij gemakkelijk te beheersen. Het gedraagt zich als de globale toestand van de communicatie, een eigenschap die nuttig is bij het ontluizen van programma's.

De uitbreiding is *logisch verantwoord*.

- Communicatie met een bord is duidelijk een neveneffect. Het is best vergelijkbaar met `assert` en `retract` in PROLOG. Niettegenstaande dit kan er een declaratieve semantiek opgesteld worden die een uitbreiding is van de klassieke semantiek voor Hornbepalingen. De semantiek van een Multi-Prologprogramma kan nog steeds beschreven worden als een deelverzameling van de Herbrandbasis. Men zou kunnen stellen dat, uitgaande van een gegeven deelverzameling van de Herbrandbasis, men kan kiezen voor hetzij een sequentiële (in b.v. PROLOG) hetzij een parallelle realisatie.

- Bordcommunicatie is een neveneffect, maar dan wel een neveneffect in de traditie van het logisch programmeren. Gegevens op het bord gedragen zich immers als logische variabelen. Zij worden éénmaal gecreëerd, kunnen verscheidene keren gebruikt worden, en worden uiteindelijk verwijderd. Evenmin als de logische variabelen, kunnen zij van waarde veranderen.

De uitbreiding is *efficiënt*.

- De kostprijs van een communicatie kan bijzonder laag gemaakt worden indien men het bord implementeert aan de hand van echt gemeenschappelijk geheugen. De kostprijs komt dan in de buurt van een paar logische inferenties. Dit is bijzonder snel in vergelijking met de andere bordgebaseerde talen die gebruik maken van lokale netwerken voor de communicatie tussen taken.

- De snelheid waarmee gecommuniceerd kan worden hangt in de eerste plaats af van de kwaliteit van de implementatie van het bord, en pas in de tweede plaats van de kwaliteit van het programma. Een goed geïmplementeerd bord staat borg voor efficiënt communicerende programma's.

De uitbreiding is *algemeen bruikbaar*.

- Het concept van taakparallellisme en expliciete bordcommunicatie is zeker niet beperkt tot een taal als PROLOG. Het kan met hetzelfde gemak toegepast worden op andere logische, en zelfs niet-logische programmeertalen. Het bewijs van de toepasbaarheid op niet-logische programmeertalen wordt geleverd door LINDA [Gel85].

- Het bordcommunicatieparadigma verschaft een aantal nieuwe inzichten en werpt een nieuw licht op het gebruik van logische programmeertalen.

De beschikbaarheid van een bord, waarvan het gebruik niet moet geschuwd worden omwille van 'purisme' zoals dit bij `assert` en `retract` wel het geval is, opent totaal nieuwe mogelijkheden.

- Het feit dat de originele taal slechts met een beperkt aantal primitieven werd uitgebreid creëert een lage drempel om deze taal te beginnen gebruiken. Het bord, d.w.z. de gemeenschappelijke toestand van alle communicaties tussen de taken wordt geapprecieerd door mensen met een meer proceduraal verleden.

- Doordat de sequentiële taken enkel via het bord met elkaar gekoppeld zijn kunnen ze relatief onafhankelijk van elkaar ontwikkeld worden. Dit is een belangrijk programmatuurbouwtechnisch aspect van de voorgestelde uitbreiding.

1.6 Originele Bijdragen

MULTI-PROLOG is vandaag lang niet de enige bordgebaseerde taal. Wel is het zo dat rond de tijd dat de Multi-Prologconcepten vorm kregen, er nog geen vermelding was van alternatieve implementaties. De oudst ons bekende implementatie is SHARED PROLOG [Cia89a, BC91] die rond dezelfde tijd ontwikkeld werd. Later ontstond een borduitbreiding voor SICSTUS PROLOG [AAF+91], en heel recent is er ook sprake van dat de taal BINPROLOG [Tar92] met een bord zou uitgerust worden. De originaliteit van mijn werk situeert zich voornamelijk op drie terreinen.

1. De *implementatie van een passief bord*. Omdat de andere bordgebaseerde systemen ontworpen zijn voor een architectuur met gespreid geheugen, werd hun bord ontworpen als een taak die communicatieaanvragen van de taken ontvangt en verwerkt. Het nadeel van dit ontwerp is dat het bord een taak is, en dat bij nagenoeg elke communicatie met het bord een interventie van de werkverdeler vereist is. In ons ontwerp is het bord een gegevensstructuur die gemanipuleerd wordt aan de hand van een aantal onderling gesynchroniseerde routines. Deze routines worden bij het bord geleverd. Het zijn dus eigenlijk de taken zelf die het bord manipuleren en het bord ondergaat deze manipulaties.

2. Het *bewijs dat deze vorm van parallellisme en communicatie niet strijdig is met het paradigma van het logisch programmeren*, maar er integendeel een nuttige uitbreiding van is. Dit komt vooral tegemoet aan de soms gehoorde kritiek dat neveneffecten in een logische programmeertaal uit den boze zijn en dat taakparallellisme niet verenigbaar zou zijn met het logisch programmeren.

3. Het *bewijs dat bordcommunicatie efficiënt kan geïmplementeerd worden* op een multiprocessor met gemeenschappelijk geheugen. Alle communicatie moet noodzakelijk via het bord afgehandeld worden en er moet voor gezorgd worden dat de integriteit van het bord bewaard blijft als verscheidene taken dit bord simultaan willen manipuleren. Dank zij een gespreid synchronisatieschema kan het bord echter zodanig geïmplementeerd worden dat het geen flessehals voor het totale systeem vormt. Door het bord rechtstreeks te implementeren aan de hand van gemeenschappelijk geheugen kan de communicatie veel sneller gemaakt worden dan in vergelijkbare bordgebaseerde PROLOGs die gespreid uitgevoerd worden op een netwerk van werkstations [ACD90, BC91, AAF+91, Tar92].

Zoals gebruikelijk werden de resultaten van dit onderzoek voorgesteld op internationale bijeenkomsten. De voornaamste worden hier kort vermeld.

(i) Het taalconcept werd voorgesteld op twee conferenties: *Parallel Computing'89* in Leiden [DB89b], en *Second International Conference on Software Engineering for Real-time Systems* [DB89a].

(ii) Het bordconcept werd voorgesteld op *Parallel Execution of Logic Programs*, een workshop in de rand van de *ICLP'91* conferentie [DB91a].

(iii) Een aantal prestatieresultaten van de prototype-implementatie werd voorgesteld op de *Second IASTED International Conference on Computer Applications in Industry* [WDB92].

(iv) De semantiek van bordcommunicatie werd voorgesteld op de conferentie *PARLE'92* [DBJ92].

Op dit ogenblik wordt er gewerkt aan een aantal bijdragen voor internationale tijdschriften waarin gerapporteerd wordt over de afgewerkte implementatie.

1.7 Overzicht van het Proefschrift

Het proefschrift bevat 4 hoofdstukken. Deze hoofdstukken kunnen beschouwd worden als het 'bewijs' van mijn thesis zoals voorgesteld in sectie 1.5.

Hoofdstuk 2 beschrijft en motiveert de beslissingen die ten grondslag liggen aan het ontwerp van de taal MULTI-PROLOG. De taal wordt voorgesteld, de semantiek informeel gespecificeerd, en een aantal sterke en zwakke punten van de taal wordt toegelicht. Het hoofdstuk wordt afgesloten met een aantal voorbeelden en een overzicht van verwante talen. Uit dit hoofdstuk volgt dat bordcommunicatie een *elegante* manier van communiceren is.

Hoofdstuk 3 gaat in op twee semantische beschrijvingen: de operationele semantiek die de concrete uitvoering van een Multi-Prologprogramma modelleert, en de declaratieve semantiek die zich bezighoudt met de logica van de

taal. Het vormt de *logische verantwoording* van de bordcommunicatie. Dit hoofdstuk staat nagenoeg volledig los van de rest en kan bij een eerste lezing gerust overgeslagen worden. Bovendien is het technisch en onderstelt het een hoeveelheid voorkennis over semantiek van logische programmeertalen. Het boek *Foundations of Logic Programming* door J.W. Lloyd [Llo87] is een goede voorbereiding op dit hoofdstuk.

Hoofdstuk 4 beschrijft het prototype dat ontwikkeld werd op een multiprocessor met gemeenschappelijk geheugen. Ruime aandacht wordt hierbij besteed aan de implementatie van het bord, het meest kritische deel van de implementatie. Het hoofdstuk wordt afgesloten met een aantal prestatiegegevens die gemeten werden op het prototype en met de resultaten die volgen uit een simulatie van het bordmodel. Hieruit blijkt dat bordcommunicatie *efficiënt* geïmplementeerd kan worden op een multiprocessor met gemeenschappelijk geheugen.

Hoofdstuk 5 geeft een illustratie van de expressieve kracht van de taal. Hiertoe wordt een aantal 'populaire' voorbeelden uit de literatuur over parallelle logische programmeertalen geïmplementeerd in MULTI-PROLOG, en gecommentarieerd. Dit vormt het bewijs van de *algemene bruikbaarheid* van MULTI-PROLOG.

Het besluit geeft een overzicht van de belangrijkste resultaten die volgen uit dit werk.

2 De Taal Multi-Prolog

When someone says, "I want a programming language
in which I need only say what I wish done,"
give him a lollipop.

— ALAN PERLIS, *Epigrams on Programming* (1982)

In dit hoofdstuk wordt het ontwerp van de taal MULTI-PROLOG *beschreven. Na een beschrijving van de basisideeën, wordt de kerntaal voorgesteld. Deze kerntaal bevat enkel de elementaire communicatieprimitieven van de taal. De voorstelling van de kerntaal wordt gevolgd door voorbeelden die een concreter beeld van de taal geven. Deze voorbeelden vormen een goede inleiding voor een overzicht van de sterke en zwakke punten van* MULTI-PROLOG. *Dit is dan de aanleiding om de kerntaal uit te breiden met een aantal bijkomende primitieven. Dit hoofdstuk wordt afgesloten met een overzicht van de verwante talen.*

2.1 De Basisidee

MULTI-PROLOG is een bordgebaseerde parallelle logische programmeertaal. In de inleiding werd de taal gecatalogeerd bij de talen met taakparallellisme. Dit uit zich in de structuur van de taal.

(i) Doordat het parallellisme en de communicatie gebruikersgespecificeerd zijn, zullen er speciale voorzieningen in de taal moeten zijn die de gebruiker in staat stellen het parallellisme en de communicatie op een precieze manier uit te drukken. In MULTI-PROLOG werd er gekozen voor een reeks van speciale predicaten.

(ii) Alles wat met parallellisme en communicatie te maken heeft zal afgehandeld worden door deze speciale predicaten. Dit heeft als gevolg dat deze predicaten als doel vanuit een louter sequentële omgeving zullen opgeroepen worden. Zoals de naam MULTI-PROLOG reeds doet vermoeden zal als

sequentieel substraat de taal PROLOG gekozen worden. Het had evenwel eender welke andere taal kunnen zijn (zelfs een parallelle).

(iii) Alle communicatie is expliciet en verloopt via het bord. Om te verhinderen dat er alternatieve communicatiepaden zouden ontstaan moet er vermeden worden dat bij de communicatie gemeenschappelijke variabelen tussen de taken gecreëerd worden. Dit kan vermeden worden door vrije variabelen bij de communicatie te hernoemen. Hierdoor kunnen ze wel gecommuniceerd worden, maar de gecommuniceerde variabelen zullen geen verband meer hebben met de originele variabelen.

2.2 De Kerntaal

Het ontwerp van de taal, uitgaande van de zonet besproken basisideeën gaf aanleiding tot de toevoeging van de volgende vier elementen aan de taal PROLOG:

(i) een gemeenschappelijk bord (is steeds impliciet aanwezig ondersteld);

(ii) een predicaat om een nieuwe taak te creëren (taakcreatie);

(iii) een predicaat om een gegeven[1] op het bord te plaatsen (schrijfbewerking);

(iv) een predicaat om een gegeven van het bord af te halen (leesbewerking).

Deze elementen worden nu één na één besproken.

2.2.1 Het Bord

Het bord is essentieel een gemeenschappelijke gegevensstructuur die door de taken gemanipuleerd kan worden. Gezien alle taken dit simultaan kunnen proberen doen, moet er gesynchroniseerd worden. Een volledige en gedetailleerde beschrijving van de interne werking en implementatie van het bord wordt gegeven in hoofdstuk 4 over de implementatie (zie blz. 98). Als eerste kennismaking kan vermeld worden dat het bord een complex gestructureerd object is dat

(i) er conceptueel uitziet als een lijst van gegevens in chronologische volgorde,

(ii) de inkomende aanvragen in volgorde van aankomst verwerkt,

(iii) over een intern geheugenbeheersysteem beschikt om gegevens op efficiënte manier te kunnen opslaan en de gebruikte geheugenruimte terug vrij te geven,

(iv) een aantal primitieven ter beschikking stelt om de inhoud van het bord op een gecontroleerde en gesynchroniseerde manier te manipuleren.

[1] Onder een gegeven wordt een algemene Prologterm verstaan. Dit kan dus een constante, een lijst, een structuur of een variabele zijn.

```
?- p(1,X)&, X=a, q(X)&, r(X).
```

Programmafragment 2.1 Voorbeeld van een taakcreatie.

Het bord is altijd aanwezig tijdens de uitvoering van een Multi-Prologprogramma. Het hoeft niet expliciet gecreëerd te worden. Initieel is het leeg.

2.2.2 De Taakcreatie

Elk Prologdoel kan gebruikt worden als argument voor de taakcreatie. Wanneer er variabelen voorkomen in dit doel, dan zullen die variabelen in de gecreëerde taak geen verband meer houden met de originele veranderlijken, dit in analogie met de communicatie van variabelen. De reden hiervoor is dezelfde: taken mogen geen gemeenschappelijke variabelen bezitten met andere taken. Alle communicatie moet langs het bord gebeuren. Naar analogie met de creatie van UNIX-taken, wordt een taak in MULTI-PROLOG opgestart door het plaatsen van het suffix & na een doel zoals geïllustreerd in programmafragment 2.1[2]. De eerste taak wordt gecreëerd als `p(1,X')` met een nieuwe veranderlijke `X'` die niets gemeen heeft met de originele `X`[3]. De tweede taak wordt gecreëerd als `q(a)`, en het resterende doel `r(a)` zal gewoon sequentieel uitgevoerd worden. Het resultaat van `r(a)` zal in dit voorbeeld beslissend zijn voor het al dan niet slagen van de vraag omdat de taakcreatie altijd slaagt, zelfs indien het predicaat niet bestaat. In dat geval wordt er een taak gecreëerd, maar van zodra de uitvoering van de taak start, zal ze falen. Dit neemt echter niet weg dat de taak wel met succes gecreëerd werd.

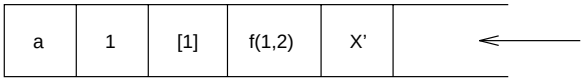
In normale omstandigheden zal de taakcreatie nooit falen. Enkel indien het maximale aantal toegelaten taken overschreden wordt zal ze falen. Dit maximale aantal taken is implementatieafhankelijk, en heeft niets met MULTI-PROLOG als dusdanig te maken. Een succesvolle taakcreatie betekent dat de taak met succes gecreëerd werd. Het wil echter niet zeggen dat de taak ook reeds uitgevoerd wordt. Om uitgevoerd te kunnen worden moet ze een processor toegewezen krijgen. Het toewijzen van een processor is de taak van de werkverdeler. De strategie van de werkverdeler is niet gedefinieerd in de taal MULTI-PROLOG. Deze kan dus vrij variëren van implementatie tot implementatie. Dit punt wordt verder uitgediept in het hoofdstuk over de implementatie (zie blz. 120).

Taken die gecreëerd werden tijdens de uitvoering van een programma worden *achtergrondtaken* genoemd. De vraag waarmee een Multi-Prologprogramma opgeroepen wordt, wordt geëvalueerd door de enige *voorgrondtaak* in het systeem. Het creëren van achtergrondtaken is niet louter voor de voorgrond-

[2] De precieze definitie van de operator & wordt later in dit hoofdstuk gegeven.
[3] Dus niet `p(1,X)`, noch `p(1,a)` worden als taak opgestart.

```
?- !a, !1, ![1], !f(1,2), !X.
```

Programmafragment 2.2 Voorbeeld van schrijfbewerkingen.



Afbeelding 2.1 Het bord als chronologische lijst.

taak weggelegd, maar is mogelijk voor elke taak. Populair uitgedrukt trachten de achtergrondtaken niets te 'bewijzen'. Hun enige taak is de voorgrondtaak te helpen bij het maken van een 'bewijs' door op gepaste wijze te reageren op de gegevens die op het bord gezet worden. Om een antwoord te kunnen opleveren moet de voorgrondtaak termineren. Achtergrondtaken hoeven niet te termineren. De voornaamste vereiste is dat ze op het gepaste ogenblik kunnen reageren. Een Multi-Prologprogramma termineert van zodra de voorgrondtaak termineert en een antwoord teruggeeft. De achtergrondtaken kunnen dan nog steeds aan het uitvoeren zijn, maar zijn niet nuttig meer omdat ze de voorgrondtaak niet meer kunnen beïnvloeden.

2.2.3 De Schrijfbewerking

Het schrijfpredicaat wordt gebruikt om informatie op het bord te plaatsen. In MULTI-PROLOG bestaat deze informatie uit Prologtermen (constanten, lijsten, structuren en variabelen). Zoals eerder vermeld zullen vrije variabelen hernoemd worden, alvorens ze op het bord geplaatst worden[4]. Er mogen immers geen wijzers vanuit het bord naar een taak lopen[5]. Naar analogie met CSP [BHR84] wordt het uitroepingsteken als symbool voor het schrijfpredicaat gekozen. In programmafragment 2.2 worden achtereenvolgens de constanten `a` en `1`, de lijst `[1]`, de structuur `f(1,2)` en een variabele `X` op het bord gezet. Het bord zal er uiteindelijk uitzien als in afbeelding 2.1. De volgorde van aankomst is gerespecteerd, en de variabele `X` werd vervangen door `X'`.

Het schrijfpredicaat faalt enkel indien het niet in staat is een term op het bord te plaatsen door een gebrek aan geheugenruimte. Door het schrijfpredicaat in een zogenaamde herhaal-faallus te plaatsen kan men ervoor zorgen dat in dit geval de te communiceren term niet verloren gaat, maar op het bord geplaatst wordt van zodra er plaats beschikbaar komt.

[4] Er is dus geen zogenaamde achterwaartse communicatie mogelijk.

[5] Een dwingende reden hiervoor is dat het bord in principe alle taken kan overleven. Men kan niet toelaten dat een term op het bord nog een verwijzing zou bevatten die naar de gegevensstructuren van een niet meer bestaande taak.

In de regel is het bord echter groot genoeg gekozen opdat elk correct werkend redelijk programma zou kunnen uitgevoerd worden. Het feit dat het bord te klein is kan wijzen op ofwel een fout in het programma (een taak die op hol geslagen is), of een zwak ontwerp van het programma (indien de mogelijkheid bestaat dat bepaalde producenten gegevens produceren tegen een veel hoger tempo dan dat de consumenten ze kunnen gebruiken, kan er beter een rem gezet worden op het tempo waartegen ze gegenereerd worden). Dit is trouwens een gevaar dat inherent is aan alle niet-synchrone communicatie. In tegenstelling met synchrone communicatie is er immers geen mechanisme dat de producenten van informatie in toom houdt [Wis92].

2.2.4 De Leesbewerking

Het leespredicaat wordt gebruikt om een gegeven op het bord te *consulteren* of te *verwijderen*. Bordtermen worden geselecteerd door middel van unificatie met het argument van het leespredicaat. Het argument kan een willekeurige Prologterm zijn. Het leespredicaat overloopt de inhoud van het bord, te beginnen met de oudste term. Van zodra een term gevonden wordt die unificeert, wordt die term gedupliceerd in de gegevensstructuren van de ontvangende taak[6]. Dit dupliceren is noodzakelijk om het bordgeheugen naderhand vrij te kunnen geven. Zoniet zou het bord in een minimum van tijd opgevuld zijn met vervallen gegevens die het geheugenbeheer ervan zouden bemoeilijken.

Om het programmeren in MULTI-PROLOG te vergemakkelijken werd er niet in één leespredicaat voorzien, maar in een reeks van acht varianten. Deze worden besproken aan de hand van drie vragen.

(i) Wat gebeurt er indien er *geen* geschikte bordterm gevonden wordt? In dit geval worden er twee mogelijkheden geboden. Het leespredicaat kan ofwel (i) blokkeren totdat een geschikte term op het bord verschijnt, ofwel (ii) falen (blokkerend tegenover niet-blokkerend).

(ii) Wat gebeurt er indien er *wel* een geschikte bordterm gevonden wordt? Twee mogelijke opties zijn: ofwel (i) wordt de term verwijderd van het bord, ofwel (ii) laat men de term gewoon staan om door de andere taken gebruikt te worden (destructief tegenover niet-destructief lezen).

(iii) Wat gebeurt er indien nadat een geschikte bordterm gevonden werd terugzoeken optreedt? Ook hier zijn er twee mogelijkheden: ofwel (i) wordt er verder gezocht op het bord, ofwel (ii) faalt het leespredicaat (terugzoekend tegenover niet-terugzoekend).

Deze drie zopas vermelde aspecten zijn orthogonaal en kunnen onafhankelijk van elkaar gecombineerd worden. De voorbeelden uit de volgende sectie zullen

[6] In de praktijk volstaat het natuurlijk om die stukken te dupliceren die nog niet aanwezig zijn.

Tabel 2.1 Overzicht van de leesdoelen.

symbool	betekenis
??!	niet-terugzoekend, niet-blokkerend, niet-destructief
??	niet-terugzoekend, niet-blokkerend, destructief
?!	niet-terugzoekend, blokkerend, niet-destructief
?	niet-terugzoekend, blokkerend, destructief
??!*	terugzoekend, niet-blokkerend, niet-destructief
??*	terugzoekend, niet-blokkerend, destructief
?!*	terugzoekend, blokkerend, niet-destructief
?*	terugzoekend, blokkerend, destructief

dit verduidelijken. De naamgeving van deze acht predicaten wordt samengevat in tabel 2.1. Een woordje over de gebruikte symboliek van de operatoren kan het één en het ander verduidelijken. De keuze van een vraagteken werd ingegeven door analogie met CSP [BHR84]. De keuze van ?? voor de niet-blokkerende variant is min of meer arbitrair. De niet-destructieve varianten hebben een uitroepteken (symbool voor de schrijfbewerking) na de vraagtekens. Dit wijst erop dat het bord onaangeroerd blijft (leesbewerking gevolgd door schrijfbewerking[7]). De *asterisk* wijst op het feit dat het betreffende leespredicaat via terugzoeken bijkomende oplossingen kan genereren. Deze notatie is ontleend aan de theorie van de reguliere uitdrukkingen waar een asterisk ook voor een arbitrair aantal gebruikt wordt. Er moet wel vermeld worden dat het terugzoeken nooit een eerder gelezen term terug op het bord zal plaatsen, dit in tegenstelling tot de parallelle logische programmeertalen die gespreid terugzoeken ondersteunen. Eenmaal een term van het bord verwijderd is, is deze voor altijd verwijderd, tenzij hij expliciet teruggeplaatst zou worden.

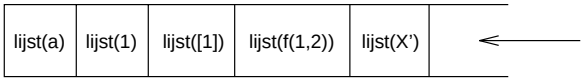
Het willen vasthouden aan een symbolische schrijfwijze voor de acht leespredicaten lijkt misschien wat ver gezocht, maar het vinden van een compacte en leesbare voorstelling voor elk van de acht varianten is zeker geen sinecure. De alternatieven, met name de volledige naam van de leesbewerking of één algemene leesbewerking met een reeks van bijkomende argumenten, zijn zo mogelijk nog verwarrender.

In de verwante bordgebaseerde parallelle logische programmeertalen zal men slechts vier van de acht hier voorgestelde varianten van het leespredicaat terugvinden omdat zij geen terugzoekende variant ondersteunen. Het terugzoeken voegt echter een extra dimensie toe die niet geïmplementeerd kan

[7] Merk evenwel op dat de niet-destructieve varianten zeker niet op deze manier kunnen geïmplementeerd worden. Het lezen en terugschrijven van een bordterm zou de volgorde van de termen op het bord immers wijzigen. Het schrijfpredicaat plaatst een term altijd aan het einde van de bordlijst.

```
schrijf_lijst([]).
schrijf_lijst([X|L]) :- !lijst(X), schrijf_lijst(L).
```

Programmafragment 2.3 `Schrijf_lijst`.



Afbeelding 2.2 De inhoud van het bord na de evaluatie van de Multi-Prologvraag `?- schrijflijst([a,1,[1],f(1,2),X]).`

worden aan de hand van de vier resterende varianten[8].

Terugzoeken maakt het mogelijk bij de niet-destructieve varianten de volledige lijst van bordtermen te overlopen. De niet-terugzoekende varianten zullen steeds de eerste bordterm die unificeert teruggeven. De andere termen zijn onbereikbaar, tenzij die eerste term eerst verwijderd wordt. Een voorbeeld op blz. 29 zal dit duidelijk maken. Anderzijds is het wel zo dat de niet-terugzoekende varianten eenvoudig kunnen geïmplementeerd worden aan de hand van de terugzoekende varianten en de knip.

Sommige parallelle logische programmeertalen met expliciete communicatie voorzien in een voorwaardelijke leesbewerking [PMCA88]. Deze maakt het gebruik van terugzoeken soms overbodig. Voor een discussie over de wenselijkheid van de invoering van een voorwaardelijke leesbewerking verwijzen we naar het einde van dit hoofdstuk.

2.3 Enkele Voorbeelden

In deze sectie wordt een aantal korte voorbeelden uitgewerkt. Deze moeten de lezer een zekere vertrouwdheid geven met de taal en met het paradigma van de bordcommunicatie. We beginnen met een paar kleine voorbeelden die geen gebruik maken van parallellisme maar wel het gebruik van het bord illustreren.

Programmafragment 2.3 plaatst de elementen van een lijst op het bord. Element `X` uit de lijst verschijnt op het bord in de vorm `lijst(X)`. Het resultaat op het bord staat weergegeven in afbeelding 2.2.

[8] Met terugzoekende variant wordt geen variant bedoeld die gespreid terugzoeken ondersteunt, maar terugzoekend zoals eerder in deze sectie voorgesteld. De reden voor het niet-aanbieden van een terugzoekende variant kan zijn dat in deze talen meer belang gehecht wordt aan de destructieve leesbewerkingen dan aan de niet-destructieve. Terugzoekende varianten zijn vooral nuttig voor de niet-destructieve varianten. Voor de destructieve leesbewerkingen maakt het functioneel weinig verschil uit of een bewerking al dan niet terugzoekend is.

```
lees_lijst([X|L]) :- ??lijst(X), !, lees_lijst(L).
lees_lijst([]).
```

Programmafragment 2.4 `Lees_lijst.`

```
maak_bord_leeg1 :- ??X, !, maak_bord_leeg1.
maak_bord_leeg1.

maak_bord_leeg2 :- ??*X, fail.
maak_bord_leeg2.
```

Programmafragment 2.5 Het wissen van het bord.

Programmafragment 2.4 verwijdert de termen `lijst/1` van het bord en plaatst ze sequentieel in een Prologlijst. Aan de hand van `??lijst(X)` worden de termen destructief, niet-blokkerend en niet-terugzoekend van het bord gehaald. Van zodra alle elementen van het bord verwijderd zijn, faalt de eerste bepaling en wordt de lijst afgesloten.

In programmafragment 2.5 worden twee procedures gegeven om het bord leeg te maken. De eerste procedure verwijdert één na één de termen van het bord op een recursieve manier. De tweede procedure is essentieel dezelfde als de eerste, maar dan op niet-recursieve manier en met een terugzoekende variant van de leesbewerking.

In een volgend voorbeeld wordt een implementatie van de predicaten `bagof` en `setof` besproken. De predicaten `bagof` en `setof` maken geen gebruik van de mogelijkheden om parallellisme uit te buiten. Ze werden geïmplementeerd als louter sequentiële routines (programmafragment 2.6).

De oplossingen worden aan de hand van een herhaal-faallus gegenereerd en op het bord gezet. Hiertoe worden ze geëncapsuleerd in de structuur `oplossing/1`. Bij `setof` wordt een oplossing met `schrijf_eenmaal/1` enkel aan het bord toegevoegd indien ze nog niet aanwezig was. Van zodra alle oplossingen gegenereerd zijn, worden ze van het bord afgehaald met `lees_lijst`. In het hoofdstuk over de implementatie staan de prestaties voor dit programma vermeld (zie blz. 126). Het blijkt nu dat de snelheid waarmee deze twee programmaatjes de oplossingen voor een gegeven doel genereren groter is dan de `setof` en `bagof` van SICSTUS PROLOG. Dit pleit voor MULTI-PROLOG gezien het op het bord zetten van een term niet enkel het copiëren van de term omhelst, maar ook het synchroniseren met de andere taken. Het feit dat dit een sequentieel programma is, en er dus geen andere taken aanwezig zijn belet niet dat alle synchronisatieprimitieven toch uitgevoerd worden alsof er verscheidene taken aanwezig zouden zijn.

In programmafragment 2.7 wordt er gebruik gemaakt van een terugzoekende leesbewerking om na te gaan of er een priemgetal op het bord staat. De

```
/* bagof */
bagof(X,G,_) :- call(G), !oplossing(X), fail.
bagof(_,_,L) :- lees_lijst(L).

/* setof */
setof(X,G,_) :- call(G), schrijf_eenmaal(X), fail.
setof(_,_,S) :- lees_lijst(L), sort(L,S).

schrijf_eenmaal(X) :- ??!oplossing(X), !.
schrijf_eenmaal(X) :- !oplossing(X).

lees_lijst([X|L]) :- ??oplossing(X), !, lees_lijst(L).
lees_lijst([]).
```

Programmafragment 2.6 `bagof` en `setof`.

```
priemgetal_op_bord_1 :- ??!*X, is_priemgetal(X).

priemgetal_op_bord_2 :- genereer_priem(X), ??!X.
```

Programmafragment 2.7 Voorbeeld met terugzoekende leesbewerking.

```
start :- genereer(X), controleer(X), fail.
start.

controleer(X) :- geldig(X), write(X).
```

Programmafragment 2.8 Sequentiële genereer-en-controleer.

tweede oplossing (niet-terugzoekend) controleert alle priemgetallenéén na één. Het probleem met de tweede oplossing is dat indien er geen priemgetal op het bord staat, dit algoritme niet termineert. Het zal steeds nieuwe priemgetallen blijven genereren. De eerste oplossing illustreert duidelijk het gebruik en het nut van een terugzoekende leesbewerking. Zonder het terugzoeken zou het onmogelijk zijn de volledige lijst te doorlopen zonder ze tegelijkertijd te moeten consumeren.

Nu volgen een paar voorbeelden die het gebruik van taken illustreren. We beperken ons hier in eerste instantie tot de creatie van parallelle taken die tijdens hun uitvoering niet communiceren.

Er wordt begonnen met een triviaal genereer-en-controleerprobleem waarin het bord niet echt gebruikt wordt. We vertrekken van het sequentiële programmafragment 2.8. Het doel `genereer(X)` genereert opeenvolgende waarden voor `X` die één na één gecontroleerd worden door `controleer(X)`. Enkel de correcte waarden worden door `controleer/1` uitgeschreven. De `fail` zorgt ervoor dat alle waarden voor `X` gegenereerd worden. Dit programmafragment kan op haast triviale wijze geparallelliseerd worden. Het resultaat is programmafragment 2.9. Per gegenereerde waarde voor `X` wordt er een afzonderlijke

```
start :- genereer(X), controleer(X)&, fail.
start.

controleer(X) :- geldig(X), write(X).
```

Programmafragment 2.9 Parallelle genereer-en-controleer.

```
dupliceer :- ?!*term(X), !term1(X), fail.

?- dupliceer&.
```

Programmafragment 2.10 Dupliceertaak voor termen.

taak opgestart om deze waarde te controleren en het resultaat mee te delen. De `fail` zorgt ervoor dat de waarden voor `X` de één na de ander gegenereerd worden. De voordelen van deze manier van parallelliseren zijn dat

(i) de ingreep in het sequentiële programmafragment nagenoeg triviaal is,

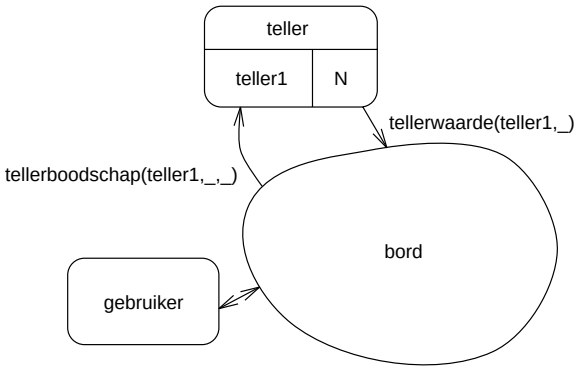
(ii) bij voldoend aantal te controleren waarden automatisch een goede belastingsverdeling tot stand komt. De gecreëerde taken kunnen immers op een willekeurige processor uitgevoerd worden. Ze worden in de volgorde van creatie toegewezen aan de eerstvolgende vrije processor.

Programmafragment 2.10 is een taak die bordtermen dupliceert. De taak leest de termen `term/1`, en dupliceert ze als `term1/1`. De herhaal-faallus zorgt ervoor dat alle termen `term/1` sequentieel verwerkt worden, en dat er op het einde van de lijst geblokkeerd wordt om te wachten op de aankomst van een nieuwe term. Deze taak zal dus enkel actief worden als een nieuwe term toegevoegd wordt, dan de term dupliceren en terug inslapen. Zonder een terugzoekende leesbewerking zou dit gedrag heel wat moeilijker te programmeren zijn.

Programmafragment 2.11 is een voorbeeld van een MULTI-PROLOG tellerobject. Dit programma maakt zowel gebruikt van parallellisme als van communicatie. Een tellerobject is een taak die intern de naam en de waarde van een teller bijhoudt. De waarde van de teller wordt aangepast aan de hand van boodschappen die via het bord gecommuniceerd worden. Alle boodschappen worden centraal ingelezen in `teller/2` en doorgespeeld naar `teller/5`. De procedure `teller/2` is in feite generisch en hoeft niet aangepast te worden indien aan het object meer functies zouden toegekend worden. In dat geval volstaat het om meer `teller/5` bepalingen bij te voegen. Dit is een gestructureerde manier van uitbreiding van een object. Een tellerobject '`teller1`' met initiële waarde 0 wordt gecreëerd met `teller(teller1, 0)&`. Een bord met één dergelijke tellertaak ziet er dan uit als in afbeelding 2.3. Er kan een boodschap naar dit object gestuurd worden door middel van het bordcommunicatiedoel `!tellerboodschap(teller1,Functie,Argument)`. De

```
teller(Naam, N) :-
  ?tellerboodschap(Naam, Functie, Arg),
  teller(Functie, Naam, N, NieuweN, Arg),
  teller(Naam, NieuweN).

teller(plus_een, _, N, NieuweN, _) :- NieuweN is N + 1.
teller(min_een, _, N, NieuweN, _) :- NieuweN is N - 1.
teller(plus, _, N, NieuweN, Arg) :- NieuweN is N + Arg.
teller(min, _, N, NieuweN, Arg) :- NieuweN is N - Arg.
teller(waarde,Naam,N,N,_) :- !tellerwaarde(Naam, N).
teller(vernietig,_,_,_,_) :- fail.
```

Programmafragment 2.11 Een tellerobject.



Afbeelding 2.3 Bord met één tellerobject.

`Functie` en zijn `Argument` bepalen wat het tellerobject precies zal doen. Merk op dat men op interactieve manier een dialoog kan voeren met deze taak 'in de achtergrond'. De interactie wordt gemakkelijker indien men een aantal hulppredicaten definieert zoals in programmafragment 2.12[9]. Alvorens een tellerobject te creëren kan het goed zijn zich er eerst van te verzekeren dat er geen boodschappen voor dat object meer op het bord staan. Dit kan aan de hand van het `wis_boodschappen/1` predicaat. Indien er `tellerboodschap/3`-termen op het bord zouden staan op het ogenblik dat een nieuw tellerobject gecreëerd wordt, zal dit object meteen beginnen met die boodschappen te verwerken. Pas dan zullen de nieuwe boodschappen aan bod komen. Een sessie van dit programma kan dan als in afbeelding 2.4 verlopen.

Een tellerobject is een duidelijk voorbeeld van hoe op een objectgeoriënteerde manier in MULTI-PROLOG geprogrammeerd kan worden. De generische structuur van een object is gegeven in afbeelding 2.5. De aspecten van de

[9] Het `op/3`-predicaat wordt hier gebruikt om zogenaamde operatoren te definiëren.

```
:- op(900, xfx, '+=').
:- op(900, xfx, '-=').
:- op(900, fx, '++').
:- op(900, fx, '--').
:- op(900, fx, toon).

nieuw(Naam) :- wis_boodschappen(Naam), teller(Naam,0)&.
Naam++ :- !tellerboodschap(Naam,plus_een,_).
Naam-- :- !tellerboodschap(Naam,min_een,_).
Naam += N :- !tellerboodschap(Naam,plus,N).
Naam -= N :- !tellerboodschap(Naam,min,N).
gedaan(Naam) :- !tellerboodschap(Naam,vernietig,_).
toon Naam :- !tellerboodschap(Naam,waarde,_),
             ?tellerwaarde(Naam,N), write(N).

wis_boodschappen(Naam) :-
   ??*tellerboodschap(Naam,_,_), fail.
wis_boodschappen(_).
```

Programmafragment 2.12 Hulppredicaten voor het tellerobject.

```
?- nieuw(teller).
yes
?- teller++.
yes
?- toon teller.
1
yes
?- teller += 4.
yes
?- toon teller.
5
yes
?- gedaan(teller).
yes
```

Afbeelding 2.4 Interactie met een tellerobject.

```
object(Toestand) :-
   ?Boodschap,
   methode(Boodschap, Toestand, NieuweToestand),
   object(NieuweToestand).

methode(boodschap1, OudeToestand, NieuweToestand) :- ...
methode(boodschap2, OudeToestand, NieuweToestand) :- ...
methode(boodschap3, OudeToestand, NieuweToestand) :- ...
```

Afbeelding 2.5 Een generisch object in MULTI-PROLOG.

objectoriëntatie in MULTI-PROLOG worden nu kort besproken.

(i) Een object wordt voorgesteld door een achtergrondtaak in MULTI-PROLOG. Het houdt zich in leven door zichzelf recursief op te roepen. Zijn lokale toestand wordt bijgehouden aan de hand van de argumenten van het doel dat gebruikt werd om de taak te creëren.

(ii) Een object ontstaat op het ogenblik dat het als taak gecreëerd wordt.

(iii) Een object sterft af van zodra het zichzelf niet langer recursief oproept. Dit kan na verloop van tijd, of als gevolg van een externe stimulus.

(iv) Een object communiceert aan de hand van termen op het bord die als boodschappen geïnterpreteerd worden.

(v) Een object wordt geactiveerd door het plaatsen van een geschikte boodschap op het bord. Zolang er geen geschikte boodschappen op het bord staan zal de taak zich in slapende toestand bevinden.

Het grootste gemis is het ontbreken van overerving. Een taak kan geen eigenschappen overerven van een andere taak en moet daardoor expliciet geprogrammeerd worden. Wel kunnen verscheidene identische taken gecreëerd worden. Dit verhindert echter niet dat objecten een extra dimensie aan de structuur van een Multi-Prologprogramma kunnen toevoegen.

2.4 Bespreking

De voorbeelden uit de vorige sectie geven een idee van de manier waarop er in MULTI-PROLOG geprogrammeerd wordt. Zij werpen ook een licht op de voordelen van de taal.

Kennis van Prolog volstaat om Multi-Prolog te kunnen begrijpen. Een inleiding tot de nieuwe communicatieprimitieven volstaat om de taal te kunnen gebruiken. Dit maakt de drempel om deze taal te beginnen gebruiken bijzonder laag. Alles is expliciet waardoor men zich als programmeur niet hoeft te bekommeren om onverwachte effecten als gevolg van niet-gewenste beïnvloeding via gemeenschappelijke variabelen. Deze kunnen gewoonweg niet optreden.

Multi-Prologtaken zijn zwak gekoppeld via het bord. Hierdoor kunnen taken individueel als sequentiële Prologprogramma's ontwikkeld worden. De interactie met het bord kan afzonderlijk bestudeerd worden door initieel een aantal termen op het bord te plaatsen, de taken te laten uitvoeren en na te gaan wat voor effect hun uitvoering op de inhoud van het bord gehad heeft. Pas nadat een taak het gewenste gedrag vertoont, zal het aan het rest van het programma toegevoegd worden (zie het voorbeeld van het tellerobject). Het

feit dat taken afzonderlijk kunnen ontworpen en uitgetest worden vormt een wezenlijk programmatuurbouwtechnisch voordeel op de systemen die gebruik maken van doelparallellisme. Bij deze laatste moet de totale toepassing als geheel ontworpen en uitgetest worden gezien de vrij hechte koppeling tussen de verschillende taken door middel van de gemeenschappelijke logische variabelen.

Door deze zwakke koppeling wordt het zelfs mogelijk te communiceren met taken die in een andere programmeertaal ontwikkeld werden. Van zodra de bordcommunicatieprimitieven beschikbaar zijn in een taal, kan een programma in deze taal in principe communiceren met het bord.

Multi-Prolog stimuleert een objectgeoriënteerde programmeerstijl. Alhoewel MULTI-PROLOG zeker niet wil pretenderen een objectgeoriënteerde logische programmeertaal te zijn, vertoont het toch zekere gemeenschappelijke kenmerken. De termen op het bord kunnen geïnterpreteerd worden als boodschappen die naar de *taken* of *objecten* gestuurd worden. Deze *boodschappen* kunnen de objecten aanzetten om hun (interne) toestand te wijzigen (zie tellerobject uit de voorbeelden). Hier houdt de gelijkenis met objectgeoriënteerde talen echter op, gezien er geen echte vorm van overerving —een toch wel essentiëel kenmerk— beschikbaar is. Wel kunnen de Multi-Prologobjecten gemakkelijk uitgebreid worden met nieuwe *methoden* door de objecten op een generische manier voor de stellen.

Bordcommunicatie in Multi-Prolog is efficiënt. Doordat het bord bijna rechtstreeks geïmplementeerd kan worden via gemeenschappelijk geheugen, kan men de communicatie in MULTI-PROLOG ook snel maken. Dit brengt met zich mee dat het bord principieel ook gebruikt kan worden voor andere taken dan louter de communicatie tussen taken. Mogelijke toepassingen zijn het ontluizen van programma's en het tijdelijk opslaan van informatie.

Als Multi-Prologcompilator volstaat een sequentiële Prologcompilator. De compilator hoeft niet te weten dat het programma parallel zal uitgevoerd worden omdat alle parallellisme gevangen zit in de communicatieprimitieven. De standaard inferentiemotor van PROLOG hoeft dus niet uitgebreid te worden hetgeen wel het geval is bij de parallelle logische programmeertalen met doelparallellisme waar de unificatieroutines uitgebreid moeten worden om rekening te houden met de gemeenschappelijke variabelen. Ook dit aspect draagt bij tot de efficiëntie.

Bordcommunicatie geeft aanleiding tot een nieuwe manier van logisch programmeren. Het gebruik van het bord is geen taboe, maar een noodzaak om efficiënte parallelle programma's te ontwikkelen. In tegenstelling tot de si-

tuatie in PROLOG waar het gebruik van `assert` en `retract` afgeraden wordt, wordt het gebruik van de bordprimitieven in MULTI-PROLOG aangemoedigd. Dit geeft soms aanleiding tot een totaal andere manier van denken. Zo kan men b.v. in plaats van gegevens in een lijst op te slaan, ze individueel op het bord opslaan. Hierdoor kunnen ze gemakkelijker door verscheidene taken simultaan gebruikt worden. Globale informatie kan beter op het bord opgeslagen worden in plaats van als extra argument gedurende de volledige uitvoering van een programma gecommuniceerd te worden.

De zonet opgesomde punten laten ons toe te stellen dat vanuit het standpunt van programmatuurbouwkunde en ontluizing MULTI-PROLOG krachtiger is dan de parallelle logische programmeertalen die gebaseerd zijn op doelparallellisme.

2.5 Uitbreidingen van de Kerntaal

In deze sectie wordt een aantal uitbreidingen van de kerntaal besproken. Deze zijn uitsluitend bedoeld om de uitdrukkingskracht van de taal te verhogen, maar ze zijn niet echt noodzakelijk. Deze bewering wordt enerzijds ondersteund door LINDA [ACG86] die slechts in een gedeelte van de functionaliteit van de kerntaal voorziet en toch als volwaardig beschouwd wordt en anderzijds door de voorbeelden uit hoofdstuk 5 die louter gebruik maken van de kerntaal en hierdoor schijnbaar niet gehinderd worden. Drie uitbreidingen zullen behandeld worden: extra voorwaarden bij de schrijfbewerkingen en de leesbewerkingen, modules en meervoudige borden.

2.5.1 Voorwaardelijke Schrijfbewerking

In de kerntaal termineert een schrijfbewerking steeds met succes indien er genoeg geheugenruimte op het bord ter beschikking is. De semantiek van de schrijfbewerking kan echter uitgebreid worden zodat er op een zinvolle manier gefaald kan worden. Een zinvolle uitbreiding is het toevoegen van een voorwaarde aan de schrijfbewerking. De syntaxis is dan `!X:V` waarbij `X` de te communiceren term is, en `V` de voorwaarde[10]. Opdat een schrijfbewerking met succes zou kunnen uitgevoerd worden, moet er in de eerste plaats aan de voorwaarde voldaan zijn. Het is dus in feite een *preconditie*.

Om een onderscheid te kunnen maken tussen faling door toedoen van de preconditie en faling door *bordoverloop* kan men ervoor zorgen dat een schrijfbewerking niet faalt indien er onvoldoende geheugen rest. Zij kan ofwel blokkeren totdat er geheugen vrijkomt of ze kan een uitvoeringsfout veroorzaken, vergelijkbaar met een stapeloverloop in de meeste systemen. In dit geval kan

[10] In CSP-termen wordt dit een schrijfwachter genoemd [Hoa85].

```
vervang(X,Y) :- !Y:?X.
```

Programmafragment 2.13 Vervangen van een term op het bord.

```
schrijf_eenmaal(X) :- ??!X, !.
schrijf_eenmaal(X) :- !X.
```

Programmafragment 2.14 Naïeve en foutieve implementatie van `schrijf_eenmaal`.

```
schrijf_eenmaal(X) :- ! X : not(??!X), !.
schrijf_eenmaal(_).
```

Programmafragment 2.15 Correcte `schrijf_eenmaal`.

men erop vertrouwen dat faling van de communicatie steeds het gevolg is van een faling van de voorwaarde.

De voorwaardelijke schrijfbewerking is slechts een echte uitbreiding indien er bordbewerkingen in de voorwaarde voorkomen. Zoniet kan men de voorwaarde gewoon vóór de bordbewerking uitvoeren zoals in `V, !X`. Het opnemen van bordprimitieven in de voorwaarde geeft aanleiding tot een reeks van interessante bewerkingen. Wij bespreken er hier twee van, namelijk het vervangen van een term op het bord en de `schrijf_eenmaal`.

Het vervangen van een term op het bord kan heel eenvoudig op de manier zoals weergegeven in programmafragment 2.13[11]. Deze bewerking vervangt de term `X` door de term `Y` op een atomaire[12] manier en is daarom verschillend van `?X, !Y`.

De `schrijf_eenmaal` zal zijn argument enkel op het bord zetten indien het er nog niet op staat. Een naïeve (en trouwens foute) implementatie staat weergegeven in programmafragment 2.14. Een probleem met deze implementatie is dat ze geen correcte synchronisatie kan garanderen. Twee taken kunnen immers simultaan deze procedure uitvoeren. Zij zullen beide tot de bevinding komen dat de term `X` nog niet op het bord staat, en hem beide creëren, waardoor hij tweemaal op het bord zal staan.

De implementatie van deze bewerking aan de hand van de voorwaardelijke schrijfbewerking is echter nagenoeg triviaal zoals blijkt uit programmafragment 2.15. Deze implementatie zal eerst de bordtermen aflopen om na te gaan of `X` reeds bestaat, en indien dit het geval is zal de eerste bepaling falen en de tweede slagen. Indien de term nog niet bestaat zal hij door de eerste bepaling aan het bord toegevoegd worden. Dit alles gebeurt atomair.

[11] De schrijfbewerking zal in dit geval wel blokkeren i.p.v. te falen indien `X` niet op het bord voorkomt.

[12] Met *atomair* wordt hier bedoeld dat een bewerking zonder onderbreking uitgevoerd wordt. In dit kader betekent dit dat het deel van het bord dat voor de bewerking van belang is, stabiel blijft tijdens de uitvoering van de bewerking.

```
setof(X,G,_) :- call(G),
                !oplossing(X):not(??!oplossing(X)),
                fail.
setof(_,_,S) :- lees_lijst(L), sort(L,S).

lees_lijst([X|L]) :-
   ??oplossing(X), !, lees_lijst(L).
lees_lijst([]).
```

Programmafragment 2.16 Setof geïmplementeerd met `schrijf_eenmaal`.

Het `schrijf_eenmaal`-primitief kan ondermeer gebruikt worden om de programmatekst voor `setof` een stuk doorzichtiger maken, en ook de mogelijkheid bieden om `setof` te parallelliseren, hetgeen in de kerntaal bijzonder moeilijk is omdat men niet kan verhinderen dat twee taken op nagenoeg precies hetzelfde ogenblik dezelfde term op het bord plaatsen.

Het programma `setof` (zie blz. 28) kan dan vereenvoudigd worden tot programmafragment 2.16.

2.5.2 Voorwaardelijke Leesbewerking

In de kerntaal kunnen er slechts syntactische voorwaarden opgelegd worden aan het soort van term dat gelezen wordt. Het is bijvoorbeeld moeilijk om uit te drukken dat enkel `f(1)` of `f(2)` mogen gelezen worden omdat er geen patroon kan gevonden worden dat enkel maar met deze twee termen kan unificeren.

Een voorwaardelijke leesbewerking zou deze extra voorwaarde op een elegante manier kunnen uitdrukken. De syntaxis van dit primitief lijkt op die van de voorwaardelijke schrijfbewerking: `?X:V`. De semantiek is ook analoog: een term wordt pas gelezen indien er niet enkel aan de unificatie voldaan is maar ook aan de voorwaarde `V`. Opnieuw gedraagt `V` zich dus als een soort van preconditie. De leesbewerking wordt pas echt uitgevoerd nadat aan de voorwaarde `V` voldaan is. In de plaats van `?` kunnen ook de zeven andere varianten van de leesbewerking gebruikt worden. In deze discussie beperken wij ons echter tot de `?`-variant.

In tegenstelling met de voorwaardelijke schrijfbewerking die in een aantal gevallen gesimuleerd kan worden in de kerntaal is dit hier niet meer het geval. Om dit duidelijk te maken worden er nu twee gebrekkige simulaties van een voorwaardelijke leesbewerking besproken. Een eerste poging leest de termen op het bord op niet-destructieve manier en verwijdert enkel die term die ook aan de voorwaarde voldoet. Het programmafragment 2.17 werkt niet goed omdat tussen het testen van de voorwaarde `V` en het verwijderen van de term `??X` een andere taak de term reeds kan geconsumeerd hebben. Uiteindelijk zal `??X` falen en zal er wel gezocht worden naar een ander alternatief, maar inmiddels zullen eventuele neveneffecten in `V` toch uitgevoerd

```
?X:V :- ?!*X, V, !, ??X.
```

Programmafragment 2.17 Gebrekkige implementatie van de voorwaardelijke leesbewerking.

```
?X:V :- ?*X, verificatie(X,V), !.

verificatie(_,V) :- V, !.
verificatie(X,_) :- !X, fail.
```

Programmafragment 2.18 Tweede gebrekkige implementatie van de voorwaardelijke leesbewerking.

```
start(X) :- ?f(X):filter(X).

filter(1).
filter(2).
```

Programmafragment 2.19 Disjunctie van gebeurtenissen.

```
p :- ?f(1) : ?f(2)
```

Programmafragment 2.20 Conjunctie van gebeurtenissen.

zijn. Programmafragment 2.18 verwijdert eerst de term, controleert dan de voorwaarde en plaatst de term terug indien er niet aan de voorwaarde voldaan is. Ook deze oplossing werkt niet goed omdat (i) termen die niet geselecteerd worden toch voor een korte tijd van het bord kunnen verdwijnen en hierdoor aanleiding kunnen geven tot faling van de niet-blokkerende leesbewerkingen in de andere taken en (ii) niet-geselecteerde termen niet op dezelfde plaats op het bord teruggeplaatst worden (`!X` plaatst termen altijd aan het einde van de lijst). De voorwaardelijke leesbewerking kan gewoonweg niet geïmplementeerd worden aan de hand van de bestaande primitieven tenzij het een sequentieel programma betreft, of tenzij de lijsten ten hoogste één element bevatten.

Het voorbeeld uit de inleiding (het lezen van `f(1)` of `f(2)`[13]) kan nu elegant uitgedrukt worden. Het programmafragment 2.19 spreekt voor zichzelf en behoeft geen verdere commentaar.

Ook hier ontstaan er tal van interessante combinaties indien men bordbewerkingen in de voorwaarde toelaat. Programmafragment 2.20 leest twee termen simultaan van het bord[14]. Gezien een voorwaardelijke leesbewerking atomair uitgevoerd wordt, moeten beide termen werkelijk op het bord voorkomen alvorens deze bewerking met goed gevolg kan uitgevoerd worden.

Het is interessant om te vermelden dat een voorwaardelijke leesbewerking

[13] Dit zal voortaan de disjunctie van twee gebeurtenissen genoemd worden.
[14] Dit zal voortaan de conjunctie van twee gebeurtenissen genoemd worden.

```
verzend(X) :- doeltaak(X,T), !boodschap(T,X).

doeltaak(f(1),taak1).
doeltaak(f(2),taak1).
doeltaak(f(_),taak2).

taak1 :- ?boodschap(taak1,f(X)), taak1.
taak2 :- ?boodschap(taak2,f(X)), taak2.
```

Programmafragment 2.21 Alternatief met een controleterm voor een disjunctie van gebeurtenissen.

als een zwakke vorm van terugzoeken kan beschouwd worden tussen de eigenlijke leesbewerking en de voorwaarde. De uitvoering van `?X:V` kan dan als volgt geïnterpreteerd worden: een term `X` wordt gelezen en de voorwaarde `V` wordt gecontroleerd. Indien er niet aan de voorwaarde voldaan is, treedt terugzoeken op waarbij de term `X` terug op de originele plaats op het bord gezet wordt. De volgende term wordt dan gelezen, de voorwaarde wordt gecontroleerd, enz.

Het ontbreken van de voorwaardelijke lees- en schrijfbewerkingen in de kerntaal mag zeker niet de indruk wekken dat deze taal hierdoor onbruikbaar zou zijn. De realiteit is dat deze primitieven in veel gevallen een programma kunnen vereenvoudigen, maar dat het ontbreken ervan niet verhindert dat een bepaalde functionaliteit gerealiseerd wordt. Soms moet men gewoon voldoende originaliteit aan de dag kunnen leggen om een gegeven probleem op een alternatieve manier op te lossen.

Een alternatieve methode om b.v. een disjunctie van gebeurtenissen te implementeren is om de voorwaarde niet door de ontvanger, maar door de zender te laten berekenen, en het resultaat van die voorwaarde samen met de term te communiceren. Onderstel dat de termen `f(1)` en `f(2)` slechts door `taak1` mogen gelezen worden, en dat alle andere termen `f/1` door `taak2` moeten gelezen worden. De realisatie hiervan kan er dan uitzien zoals in programmafragment 2.21. De bijkomende term `T` in de boodschap wordt de controleterm [PM91] genoemd en geeft aan voor welke taak een bepaald gegeven bestemd is. Het nadeel van deze oplossing is dat de zender op voorhand de ontvangers moet kennen en hierdoor minder algemeen is.

2.5.3 Modules

Doordat MULTI-PROLOG gebaseerd is op de taal PROLOG, erft het ook de eigenschappen van deze taal, inclusief de minder goede. Eén van die minder goede kenmerken van PROLOG is het gemis aan een moduleconcept. Het voordeel hiervan is dat alle bepalingen globaal zijn en door iedereen en op elk ogenblik gebruikt kunnen worden. Het nadeel is dat er slechts één symbolentabel be-

staat en dat er bij grote projecten gemakkelijk conflicten in de naamgeving van predicaten kan ontstaan. Bovendien maakt het ontbreken van een modulesysteem het bijzonder moeilijk om met een aantal mensen simultaan aan een programma te werken en wordt de ontwikkeling vertraagd omdat de bestanden uiteindelijk te groot worden.

De meeste hedendaagse Prologomgevingen voorzien dan ook in een modulesysteem dat deze ongemakken wegwerkt ([CWA+ 91]). Een module bestaat uit een verzameling van bepalingen met een lokale symbolentabel voor de namen van predicaten waardoor zij niet zichtbaar zijn voor de buitenwereld. Om een predicaat extern zichtbaar te maken moet het expliciet geëxporteerd worden. Modules die een predicaat van een andere module willen gebruiken moeten dit expliciet importeren. De symbolentabel voor de constanten blijft echter globaal over alle modules.

De interactie tussen MULTI-PROLOG en het modulesysteem van SICSTUS PROLOG [CWA+ 91] verloopt zonder problemen dank zij het feit dat de symbolentabel voor de constanten globaal blijft bestaan. Dit wil zeggen dat er voor de constanten slechts één symbolentabel bestaat en dat het adres van een constante in die symbolentabel dus gebruikt kan blijven worden als identificatie van die constante op het bord.

2.5.4 Lokale Borden

Niet enkel de programmatekst, maar ook de bordinhoud kan in modules opgedeeld worden. Het valt immers te verwachten dat in grote toepassingen er zich ook naamconflicten op het bordniveau zullen voordoen. De mogelijkheid om lokale borden te creëren en aan te wenden zou een aantal van deze conflicten uit de weg kunnen ruimen.

De aanwezigheid van verscheidene borden vereist dat zij op deéén of andere manier moeten kunnen geselecteerd worden. Een voor de hand liggende manier is om de bordprimitieven te laten voorafgaan door de naam van het bord. De lijst van primitieven staat in tabel 2.2. Om een zo groot mogelijke flexibiliteit te bereiken wordt er de voorkeur aan gegeven om een bord als een object van eerste orde te beschouwen zodat het kan gecommuniceerd worden. Om geen extra type aan de taal PROLOG te moeten toevoegen kiezen wij voor een symbolische constante als identificatie van het bord. Een dergelijk bord moet expliciet gecreëerd worden met `createbb/1` en verwijderd worden met `killbb/1`. Het predicaat `cleanbb/1` kan gebruikt worden om een bord leeg te maken. Indien `createbb/1` gebruikt wordt met een vrije variabele als argument, wordt er een anoniem bord aangemaakt (d.w.z. dat er een unieke symbolische constante gegenereerd wordt). Dit is nuttig indien men slechts tijdelijk een bord nodig heeft en men een naamconflict wil vermijden. Eenzelfde bord kan geen tweemaal aangemaakt worden. De invoering van meervoudige

Tabel 2.2 Bordcommunicatiedoelen.

onvoorwaardelijke primitieven	voorwaardelijke primitieven
`bord ! T`	`bord ! T : V`
`bord ? T`	`bord ? T : V`
`bord ?? T`	`bord ?? T : V`
`bord ?! T`	`bord ?! T : V`
`bord ??! T`	`bord ??! T : V`
`bord ?* T`	`bord ?* T : V`
`bord ??* T`	`bord ??* T : V`
`bord ?!* T`	`bord ?!* T : V`
`bord ??!* T`	`bord ??!* T : V`

borden maakt het basisbord in principe overbodig. Het zou trouwens expliciet gecreëerd kunnen worden. Het probleem met het creëren van borden is echter dat ofwel de naam van het bord gecommuniceerd moet worden, of dat er vaste afspraken moeten bestaan over de naamgeving van borden. Om op dit niveau verwarring zoveel mogelijk tegen te gaan werd het moederbord behouden en kreeg het de naam `blackboard`. Verder kan het steeds weggelaten worden als argument van een bordbewerking en is het gecreëerd nog voordat de eigenlijke uitvoering van een programma begint. Doordat er geen functioneel verschil is tussen het moederbord en de dynamische borden kan dit eerste ook vernietigd worden. Het nut hiervan wordt meteen duidelijk.

Doordat een bord nu een beperkte levensduur kan hebben moeten er voorzieningen getroffen worden voor enerzijds de opvang van bordbewerkingen op niet-bestaande borden en anderzijds voor de opvang van taken die staan te wachten op een term van een bord dat vernietigd wordt. Een bordbewerking op een niet-bestaand bord zal een uitvoeringsfout veroorzaken. Het laten falen van een bordbewerking zou aanleiding kunnen geven tot verkeerde interpretaties bij bewerkingen die ook om andere redenen kunnen falen. Het laten blokkeren totdat het bord gecreëerd wordt zou ook zinvol kunnen zijn, maar werd niet overwogen omdat er genoeg andere manieren zijn om een taak te laten wachten en we deze vorm van redundantie zoveel mogelijk willen vermijden.

Bij het vernietigen van een bord worden ook alle taken die staan te wachten op informatie van dit bord vernietigd. Dit is in overeenstemming met het gedrag van een bordbewerking die op een niet meer bestaand bord uitgevoerd wordt. Dit gedrag is bovendien nuttig omdat er hierdoor een manier ontstaat om een taak te stoppen. In de kerntaal moeten er steeds expliciete voorzie-

ningen genomen worden om een taak te doen stoppen[15]. Bij de vernietiging van een bord worden de wachtende taken automatisch door het bord vernietigd. Doordat taken die willen communiceren met een onbestaand (of niet meer bestaand) bord een uitvoeringsfout veroorzaken (en hierdoor vernietigd worden), zullen alle taken die met dit bord communiceren *na verloop van tijd* vernietigd zijn. Nu wordt het ook duidelijk waarom het nuttig kan zijn het moederbord te kunnen vernietigen. Op die manier kan men ervoor zorgen dat alle taken afsterven. Dit verhindert echter niet dat niet-communicerende, niet-terminerende taken voor altijd blijven verderlopen. Zij kunnen echter aanzien worden als onbelangrijk, en wellicht foutief. Een niet-terminerende vraag kan nooit een antwoord opleveren en is dus niet echt nuttig, en een niet-communicerende achtergrondtaak kan geen positieve bijdrage leveren tot de uiteindelijke oplossing en is dus ook niet erg nuttig. Evaluatieprogramma's kunnen soms gebruik maken van dergelijke taken om de belasting kunstmatig te verhogen, maar dit kan beschouwd worden als oneigenlijk gebruik van een taak.

Het nut van de lokale borden wordt nu geïllustreerd aan de hand van de programma's `setof` en `bagof`. In programmafragment 2.16 worden alle oplossingen op het bord geplaatst na eerst ingepakt geweest te zijn in de structuur `oplossing/1`. Deze werkwijze verhindert echter dat dit programmafragment simultaan door twee taken uitgevoerd wordt omdat de oplossingen van de twee taken vermengd zouden kunnen worden. Het programmafragment 2.22 creëert nu eerst een lokaal bord om de resultaten tijdelijk op te slaan. Nu kan het `setof` predicaat door verscheidene taken simultaan gebruikt worden zonder dat er vermenging kan optreden.

Het gebruik van lokale borden is echter niet nieuw, maar werd ook reeds voorgesteld in LINDA 3 [Gel89]. Het gedrag van de lokale borden is in de beide systemen echter niet gelijklopend. Voor een bespreking van de verschillen wordt er verwezen naar blz. 52.

2.6 Operatordefinities

Om de syntaxis van de bordbewerkingen uit dit hoofdstuk te ondersteunen in PROLOG worden de operatordefinities uit tabel 2.3 ingevoerd. Zij laten toe op symbolische en compacte wijze met het bord te communiceren. De syntactische constructies die met deze operatoren kunnen gemaakt worden zien er dan uit als in afbeelding 2.6. De precedentie van de &-operator wordt iets hoger gekozen dan de precedenties van de schrijf- en leesbewerkingen om dergelijke bewerkingen gemakkelijk als taak in de achtergrond te kunnen creëren. Een andere keuze zou minder zinvol geweest zijn.

[15] Een taak kan enkel stoppen door zichzelf niet langer recursief te blijven oproepen.

```
/* bagof */
bagof(X,G,L) :-
   createbb(Bord), bagof(X,G,L,Bord), killbb(Bord).
bagof(X,G,_,Bord) :-
   call(G), Bord!oplossing(X), fail.
bagof(_,_,L,Bord) :-
   lees_lijst(L,Bord).

/* setof */
setof(X,G,L) :-
   createbb(Bord), setof(X,G,L,Bord), killbb(Bord).
setof(X,G,_,Bord) :-
   call(G), Bord!oplossing(X):not(??!oplossing(X)), fail.
setof(_,_,S,Bord) :-
   lees_lijst(L,Bord), sort(L,S).

lees_lijst([X|L],Bord) :-
   Bord??oplossing(X), !, lees_lijst(L,Bord).
lees_lijst([],_).
```

Programmafragment 2.22 `bagof` en `setof` met een lokaal bord.

Tabel 2.3 Operatordefinities voor de borddoelen.

```
?- op(950,fx,'!').          ?- op(950,xfx,'!').
?- op(950,fx,'?*').         ?- op(950,xfx,'?*').
?- op(950,fx,'?').          ?- op(950,xfx,'?').
?- op(950,fx,'??*').        ?- op(950,xfx,'??*').
?- op(950,fx,'??').         ?- op(950,xfx,'??').
?- op(950,fx,'?!*').        ?- op(950,xfx,'?!*').
?- op(950,fx,'?!').         ?- op(950,xfx,'?!').
?- op(950,fx,'??!*').       ?- op(950,xfx,'??!*').
?- op(950,fx,'??!').        ?- op(950,xfx,'??!').
?- op(955,xfx,':').
?- op(960,xf,'&').
```

Uitdrukking	Interne voorstelling
`?a`	`'?'(a)`
`?[1,2,3,4]`	`'?'([1,2,3,4])`
`g(1,2,3)&`	`'&'(g(1,2,3))`
`?X:f(X)`	`':'('?'(X),f(X))`
`!f(X):?f(Y)`	`':'('!'(f(X)),'?'(f(Y)))`
`b?a`	`'?'(b,a)`
`b?X:f(X)`	`':'('?'(b,X),f(X))`
`!f(X)&`	`'&'('!'(f(X)))`

Afbeelding 2.6 Voorbeelden van communicatieprimitieven.

Tabel 2.4 Kanaalgebaseerde tegenover bordgebaseerde communicatie.

	zenden	ontvangen
kanaalgebaseerd	`K!G`	`K?G`
bordgebaseerd	`!K(G)`	`?K(G)`
bordgebaseerd	`kanaal(K,G)`	`kanaal(K,G)`

2.7 Verwante Talen

In deze sectie wordt een overzicht gegeven van een aantal andere parallelle logische programmeertalen die gebruik maken van taakparallellisme of expliciete communicatie. Voor wat de communicatie betreft worden er twee methoden gebruikt. Een eerste groep van talen is kanaalgebaseerd, de tweede groep is bordgebaseerd.

2.7.1 Kanaalgebaseerde Talen

Deze groep van talen is eigenlijk het minst verwant met MULTI-PROLOG. Zij wordt niettemin bij de verwante talen geplaatst omdat de manier van communiceren toch een zekere gelijkenis vertoont. Tabel 2.4 maakt duidelijk dat kanaalgebaseerde communicatie gemakkelijk kan vertaald worden in bordcommunicatie [Pin91b]. Twee bordgebaseerde alternatieven zijn mogelijk. Het eerste alternatief `K(G)` is te verkiezen omdat het efficiënter is. De te communiceren term is met name kleiner en maakt het mogelijk dat er gëindexeerd wordt op de naam van de structuur.

In de praktijk is er maar één verschil tussen deze simulatie van kanalen aan de hand van een bord en echte kanalen. De gesimuleerde kanalen kunnen door alle taken gebruikt worden terwijl echte kanalen slechts door die taken die de identificatie van het kanaal bezitten gebruikt kunnen worden. Dit kan echter ook gesimuleerd worden door de naam van het kanaal niet zelf te kiezen, maar automatisch te laten genereren en de naam van het kanaal dan tussen de taken onderling door te geven zoals in [Pin91b] beschreven wordt. Op die manier kunnen kanaalgebaseerde systemen volledig aan de hand van een bord geïmplementeerd worden. Het omgekeerde is echter heel wat moeilijker.

Delta-Prolog

DELTA-PROLOG [CMCP92] is wellicht één van de oudste kanaalgebaseerde parallelle logische programmeertalen. Zij is gebaseerd op de theorie van de gespreide logica van Monteiro [Mon84, Mon86]. Deze theorie breidt de logica van Hornbepalingen uit met mogelijkheden om op expliciete wijze parallellisme en communicatie te specificeren.

DELTA-PROLOG verrijkt PROLOG met het gebeurtenisconcept en en-parallellisme. Een gebeurtenis wordt gebruikt om een term tussen twee taken te communiceren. Er zijn twee gebeurtenisdoelen: één om te verzenden `G!K` en één om te ontvangen `G?K`. Het symbool `K` representeert een unieke naam die als kanaal dienst doet. Twee gebeurtenisdoelen `G1!K1` en `G2?K2` zijn complementair indien `G1` en `G2` unificeren en `K1` en `K2` dezelfde naam zijn. De communicatie is synchroon. Dit wil zeggen dat zowel de zender als de ontvanger moeten wachten totdat de communicatie heeft plaatsgevonden. Op het ogenblik dat er communicatie optreedt, kan er wel gecommuniceerd worden in twee richtingen (bidirectioneel). Dit wil zeggen dat er zowel aan de zenderkant als aan de ontvangerkant kan geünificeerd worden[16]. Nadat de communicatie heeft plaatsgevonden kunnen de gemeenschappelijke variabelen echter niet meer als communicatiemedium gebruikt worden (er kan dus geen achterwaartse communicatie plaatsvinden). Indien er faling optreedt tijdens of na de communicatie, vindt er gespreid terugzoeken plaats waarbij ook de communicatie ongedaan zal gemaakt worden. Dit vereist dat ook de taak waarmee de communicatie plaatsvond moet terugkeren naar het communicatiepunt en hierbij ook alle taken waarmee inmiddels gecommuniceerd werd op hun stappen moeten terugkeren.

Het en-parallellisme wordt uitgedrukt aan de hand van een gesplitst doel, voorgesteld door het symbool `//` als parallelle conjunctie tussen twee doelen. Soms zal er een vorm van terugzoeken nodig zijn om beide doelen te doen slagen omdat zij onafhankelijk van elkaar geëvalueerd worden en de substituties pas na (en dus niet tijdens) de evaluatie met elkaar vergeleken worden[17].

Een aantal uitbreidingen veralgemeent de taal [PMCA86, CFL88, PMCA88].

(i) `G!K:V` is een voorwaardelijke synchrone schrijfbewerking. Het gegeven `G` wordt over kanaal `K` verstuurd indien er aan de voorwaarde `V` voldaan is.

(ii) `G?K:V` is een voorwaardelijke synchrone leesbewerking. Deze voorwaardelijke leesbewerking is complementair met een voorwaardelijke schrijfbewerking indien de kanalen dezelfde zijn, de gegevens unificeren en er voldaan is aan de beide voorwaarden.

(iii) `G!!K` is een asynchrone schrijfbewerking.

(iv) `G??K` is de 'asynchrone' leesbewerking, d.w.z., het complement van `G!!K`. In tegenstelling met wat haar naam doet vermoeden blokkeert deze bewer-

[16] Dit is een elegant mechanisme dat jammer genoeg enkel bij synchrone communicatie kan gebruikt worden. Bovendien is het moeilijk efficiënt te implementeren op een multiprocessor met gespreid geheugen door het ontbreken van gemeenschappelijke variabelen.

[17] Tijdens de evaluatie kan er niet vergeleken worden omdat twee taken op twee verschillende processoren kunnen uitgevoerd worden en tijdens de uitvoering geen gemeenschappelijke variabelen bezitten.

king echter wel indien er geen gegeven voorhanden is.

(v) `G!!!K` is een synchrone schrijfbewerking zonder gespreid terugzoeken.

(vi) `G???K` is een synchrone leesbewerking zonder gespreid terugzoeken.

(vii) `::` is een keuzedoel om niet op één, maar op verscheidene gebeurtenissen terzelfder tijd te wachten. Een keuzedoel wordt gebruikt als volgt: `G1,E1 :: G2,E2` waarbij `G1` en `G2` gebeurtenisdoelen zijn en `E1` en `E2` uitdrukkingen zijn die geëvalueerd worden van zodra de corresponderende gebeurtenis heeft plaatsgevonden. Een keuzedoel evalueert met succes van zodra één van de alternatieven met succes evalueert.

DELTA-PROLOG is een elegante taal met voldoende uitdrukkingskracht om een scala van problemen mee aan te pakken. Zij heeft niettemin ook een aantal problemen. Vooreerst is er het probleem van de granulariteit. DELTA-PROLOG is een 'gespreide' logische programmeertaal. De taken worden enparallel uitgevoerd en niets verhindert dat er voor een enkelvoudige unificatie een taak opgestart wordt. Het gespreid terugzoeken maakt dit alles bijzonder complex, zowel om te implementeren als om te gebruiken.

Ten tweede kan communicatie met gespreid terugzoeken bijzonder krachtig lijken, maar is het zeker niet gemakkelijk in het gebruik. Het precies begrijpen van wat er in een sequentieel programma bij het terugzoeken gebeurt is al niet altijd even eenvoudig, laat staan wat er gebeurt als dit door verscheidene taken tegelijkertijd gedaan wordt. Bovendien kunnen taken hierdoor niet zomaar onafhankelijk van de rest van de toepassing ontwikkeld worden. Het trachten te bewijzen dat er geen patstellingen kunnen optreden in een programma dat gebruik maakt van gespreid terugzoeken vergt een aanzienlijke ervaring. Hierdoor kan wellicht uitgelegd worden waarom er ook gebeurtenisdoelen zonder gespreid terugzoeken aan de taal toegevoegd werden.

PMS-Prolog

De taal PROCESSES-MODULES-STREAMS PROLOG werd ontwikkeld door M.J. Wise aan de Universiteit van Sidney [WJH92]. Zoals de naam doet vermoeden betreft het een parallelle PROLOG gebaseerd op taakparallellisme en expliciete communicatie via stromen. Het moduleconcept geeft aan de programma's een bijkomende structuur. De taal werd speciaal ontworpen voor een multiprocessor met gespreid geheugen.

Het moduleconcept uit PMS-PROLOG lijkt goed op dat van MODULA-2 [Wir82]. Modules kunnen, net zoals in MODULA-2, vernesteld gedefinieerd worden. Een module is volledig afgeschermd van haar omgeving. De predicaten die in de exportlijst voorkomen zijn de enige die door de buitenwereld gebruikt kunnen worden. Exporteren alleen volstaat echter niet. De te gebruiken predicaten moeten door de gebruiker ook nog eens geïmporteerd worden. Bovendien kan

een module in beperkte mate geparameteriseerd worden door de parameters in de naam van de module te vermelden en ze bij het importeren een bepaalde waarde te geven. Deze moduleparameters zijn een soort van vrije variabelen en zijn overal in het bereik van de module zichtbaar zonder dat ze expliciet in de kop van een bepaling moeten voorkomen. Dit moduleconcept is een heel stuk beter uitgewerkt dan wat er in meer traditionele PROLOGs bestaat.

Taken in PMS-PROLOG zijn speciale modules die een predicaat exporteren dat dezelfde naam draagt als de module zelf. Een taak kan op dezelfde manier als een module geparameteriseerd worden en wordt gecreëerd aan de hand van het `fork/1`-predicaat. De taak die `fork/1` uitvoert wacht totdat de gecreëerde taak termineert. Het creëren van één enkele taak is dan een procedure-oproep en geeft geen aanleiding tot het ontstaan van parallellisme. Om toch simultaan uitvoerende taken te kunnen creëren, is het mogelijk niet één, maar een lijst met taken te creëren. Deze taken zullen dan wel echt parallel uitgevoerd worden. Opnieuw zal het `fork/1`-doel slechts termineren indien alle gecreëerde taken getermineerd zijn.

Taken kunnen geen variabelen gemeenschappelijk hebben. Ze communiceren uitsluitend door middel van expliciete `transmit/2`- en `receive/2`-predicaten. Deze predicaten maken gebruik van unidirectionele kanaalstructuren. De namen van de kanalen moeten expliciet tussen de taken gecommuniceerd worden, samen met de toegangsrechten (enkel lezen of enkel schrijven). Variabelen moeten bij het communiceren hernoemd worden om het ontstaan van alternatieve communicatiekanalen te vermijden. Vermeldenswaard is wel dat bij het `receive`-predicaat een lijst met kanalen gespecificeerd kan worden. In dat geval zal er niet-deterministisch gewacht worden op input vanéén van deze kanalen.

De manier om taken te creëren met `fork/1` doet in zekere zin nogal artificieel aan en doet meer denken aan een procedureoproep dan aan een taakcreatie. Een andere nadeel is dat er nogal afgeweken wordt van de traditionele Prologsyntaxis bij de invoering van modules. Wel moet er toegegeven worden dat het moduleconcept hierdoor elegant en krachtiger werd.

In de taal MB-PROLOG [Wis92], werd het model van de taal PMS-PROLOG nog verder uitgebreid met zogenaamde communicatiemakelaars. Dit zijn bemiddelaars bij het tot stand komen van een communicatie tussen een zender en een ontvanger. Zij zijn een veralgemening voor de kanaalnamen in PMS-PROLOG. Een makelaar gaat als volgt te werk. Nadat de zender en de ontvanger beide aan de makelaar te kennen hebben gegeven dat ze met elkaar willen communiceren, geeft de makelaar het fysisch adres van de ontvanger door aan de zender. Op dat ogenblik vindt de communicatie rechtstreeks plaats van de zender naar de ontvanger, zonder tussenkomst van de makelaar.

Een communicatiemakelaar introduceert een bijkomend niveau in de com-

municatie. Taken hoeven elkaar niet meer te kennen; ze hoeven enkel de naam van de makelaar die instaat voor hun onderlinge communicatie te kennen. De makelaar kan het verloop van de communicatie op allerlei manieren beïnvloeden zonder dat de taken hiervan iets merken. Zo kunnen er verscheidene zenders voor één ontvanger zijn, en omgekeerd. De zenders moeten een blind vertrouwen hebben in het adres dat hen door een makelaar ter beschikking gesteld wordt. Doordat de communicatie nu deel uitmaakt van een afzonderlijke laag krijgen de taken een grotere vrijheid om zich te verplaatsen in het netwerk van processoren.

Bordcommunicatie staat dichter bij communicatiemakelaars dan bij de klassieke kanaalgebaseerde systemen omdat ook bij bordcommunicatie de identiteit van de ontvanger niet door de zender gekend moet zijn. Het nadeel van communicatiemakelaars is wel dat de taken die met elkaar willen communiceren nog steeds de naam moeten kennen van de makelaar die zal instaan voor deze communicatie. In MB-PROLOG kan dit op elegante wijze door de naam van de makelaar te communiceren als parameter bij de creatie van de taak. Op die manier moet de naam ten minste niet via de bepalingen doorgegeven worden.

CS-Prolog

COMMUNICATING SEQUENTIAL PROLOG [FF92] werd ontwikkeld door Ivan Futó en Janus Szeredi aan het Computer Research Institute in Budapest. Het is een gespreide PROLOG met behoud van de eigenschappen van het declaratief logisch programmeren. Dit wil zeggen dat ondanks de gespreide uitvoering toch de volledige zoekruimte zal doorzocht worden naar oplossingen. De taal is gebaseerd op grofkorrelig parallellisme (beperkt en-parallellisme) en expliciete communicatie. Om de zoekruimte volledig te kunnen doorzoeken wordt er gebruik gemaakt van gespreid terugzoeken. De taal vormt een uitbreiding van PROLOG gezien de syntaxis van PROLOG bewaard blijft en alle voorzieningen die nodig zijn om de taal te parallelliseren toegevoegd werden aan de hand van ingebouwde predicaten. Een programma dat geen gebruik maakt van deze ingebouwde predicaten is een Prologprogramma.

Het unieke aan CS-PROLOG is dat het parallellisme gecreëerd wordt op basis van *hypothesen*. Een hypothese is een doel dat afzonderlijk bewezen wordt. De taak die de hypothese aanneemt hoeft niet te wachten op het bewijs, maar gaat ervan uit dat de hypothese waar is. Een programma termineert van zodra alle taken getermineerd zijn, dus ook de hypothesen. Indien een hypothese niet kan bewezen worden zal de taak die de hypothese aangenomen heeft terugzoeken tot vóór het punt waar de hypothese aangenomen werd. Naderhand kan er dan een nieuwe hypothese aangenomen worden. Alle hypothesen moeten volledig geconcretiseerd zijn.

Een CS-Prologprogramma wordt geïmplementeerd aan de hand van een aantal PIMs (PROLOG Inferentiemachines). Dit zijn PROLOGs die volledig onafhankelijk van elkaar uitvoeren (doordat alle objecten die gecommuniceerd worden volledig geconcretiseerd moeten zijn, kunnen er geen communicatiekanalen op basis van de logische variabele ontstaan). De communicatie tussen de PIMs gebeurt aan de hand van expliciete communicatieprimitieven. De communicatie is asynchroon; m.a.w. het verzenden van een gegeven zal nooit blokkeren. De boodschappen worden niet centraal maar gespreid door de PIMs bijgehouden op basis van de naam van de gecommuniceerde structuren.

We beëindigen dit overzicht van CS-PROLOG met een overzicht van de primitieven.

- `new(Doel,Naam,Processor)` creëert een taak met de naam `Naam` op `Processor`. Deze taak zal `Doel` evalueren. Van zodra `Doel` als taak gecreëerd is kan het door de oproeper als hypothese gebruikt worden. Indien `Doel` door de net gecreëerde taak niet kan bewezen worden, dan zal de taak die `new/3` uitgevoerd heeft gedwongen worden terug te keren tot op het punt dat `new/3` uitgevoerd werd.
- `new_unb(Doel,Naam,Processor)` is een niet-terugzoekende variant van `new/3`. Zelfs als `Doel` niet kan bewezen worden, zal er niet teruggezocht worden.
- `delete_process(Naam)` en `delete_process_unb(Naam)` vernietigen taken.
- `active_process(Naam)` concretiseert `Naam` met de naam van de taak die dit predicaat evalueert.
- `max_processes(MP)` concretiseert `MP` met het totaal aantal processoren.
- `send(Boodschap,Taken)` en `send_unb(Boodschap,Taken)` zenden `Boodschap` uit naar alle taken uit de lijst `Taken`.
- `wait_for(Boodschap)` en `wait_for_dnd(Boodschap)` wachten deterministisch resp. niet-deterministisch op `Boodschap`.

Alle doelen die als hypothese voorgesteld worden en alle termen die gecommuniceerd worden moeten volledig geconcretiseerd zijn. Het concept van hypothese in CS-PROLOG is een elegante en declaratieve manier om parallelle taken te creëren. De taal lijdt evenwel aan dezelfde nadelen als DELTA-PROLOG. Het feit dat de communicatie en de hypothesen aanleiding kunnen geven tot gespreid terugzoeken legt een hypotheek op de efficëntie van de implementatie en maakt het soms moeilijk om het gedrag van bepaalde programma's te begrijpen.

Bovendien kan men ook bezwaren hebben tegen het feit dat in de taal het nummer van de processor zichtbaar is bij de creatie van een taak. Een dergelijk harde binding van een taal aan de hardware is niet steeds gewenst. Wat gebeurt er wanneer men een programma dat ontwikkeld werd voor een multiprocessor met 10 processoren wil laten uitvoeren op een multiprocessor met slechts 5 processoren?

Het feit dat enkel volledig geconcretiseerde termen gecommuniceerd kunnen worden legt ook beperkingen op aan de uitdrukkingskracht van de taal. Soms is het nuttig termen te communiceren waarvan sommige delen verband houden met elkaar zoals X+X. Deze term drukt de som uit van twee identieke termen. Deze informatie is nuttig indien deze term bijvoorbeeld zou moeten afgeleid worden.

CC-talen

De concurrent constraint (CC) talen [Sar89] maken ook gebruik van een soort van gemeenschappelijke gegevensstructuur die de *berging* genoemd wordt. Deze berging wordt gebruikt om randvoorwaarden op te slaan. De berging is echter geen bord in de zin van MULTI-PROLOG en wel om twee redenen.

1. De berging is monotoon. Dit wil zeggen dat er enkel informatie kan aan toegevoegd worden. Er kan geen informatie uit de berging verwijderd worden.

2. De berging is actief en wordt gebruikt om de randvoorwaarden te vervullen. Het toevoegen van een randvoorwaarde aan de berging betekent meer dan louter het opslaan ervan. Ze moet ook gecombineerd worden met de informatie die zich reeds in de berging bevindt. Een bord bevat daarentegen enkel volledig onafhankelijke termen.

Het bord van MULTI-PROLOG is niet monotoon, noch actief en is daarom moeilijk te vergelijken met de berging van de CC-talen.

Communicerende Prologeenheden

Deze programmeertaal [MN86] wil in de eerste plaats objectgeoriënteerd zijn, en pas in de tweede plaats parallel. Het hoofddoel is het opsplitsen van een Prologprogramma in een aantal communicerende Prologobjecten. Het programmafragment dat een dergelijk *object* beschrijft wordt een *eenheid* genoemd. Een eenheid heeft een aantal ingangen. Dit zijn predicaten die buiten de eenheid zichtbaar zijn. Deze predicaten kunnen echter niet rechtstreeks geëvalueerd worden, maar moeten via het `send/3` predicaat opgeroepen worden. De syntaxis is

```
send(Eenheid, Doel, Resultaat).
```

Het `send/3`-predicaat zendt `Doel` naar het object `Eenheid` en wacht totdat via `Resultaat` een waarde teruggegeven wordt. Het `Resultaat` kan waar of vals zijn. De berekende waarden worden geïnstantieerd in het `Doel`. Als er verscheidene oplossingen zijn zal dit ook meegedeeld worden. Het `send/3`-predicaat is synchroon en faalt nooit.

Een eenheid voert een programma uit aan de hand van een *metavertolker*. Deze is verantwoordelijk om de doelen die niet lokaal gedefinieerd zijn om te zetten naar `send/3`-doelen die naar andere eenheden gestuurd worden. Deze metavertolker maakt het mogelijk dat de communicatie losgekoppeld wordt van het programma. Afhankelijk van de omstandigheden waarin een programma uitgevoerd wordt kan het gebruik maken van verschillende eenheden.

Indien een eenheid uit louter bepalingen zonder neveneffecten bestaat, kan deze eenheid door verscheidene taken simultaan gebruikt worden. Twee taken kunnen synchroniseren door middel van synchronisatiebepalingen van de vorm

```
ingang(...), aanvaard(...) :- romp(...)
```

Enkel indien er een twee externe aanvragen komen, één voor `ingang`, en één voor `aanvaard`, dan zal de evaluatie van `romp` doorgaan. Via unificatie zullen de resultaten aan de twee externe aanvragers teruggestuurd worden. Dit mechanisme is vergelijkbaar met de manier van communiceren in gegeneraliseerde Hornbepalingen [JM91b, JM91a].

2.7.2 Bordgebaseerde Talen

Deze talen vertonen de grootste gelijkenis met MULTI-PROLOG. Zij maken allemaal gebruik van een globale gegevensstructuur voor de communicatie tussen de taken. Behoudens de expertsystemen houden de meeste van deze talen op de één of andere manier verband met het Lindacommunicatieparadigma.

Expertsystemen

Expertsystemen zijn de eerste toepassingen geweest die gebruik maakten van een bord om hun uitvoering te ondersteunen [LE77, EM88, JDB89]. Organisatorisch bestaat een bordgebaseerd expertsysteem uit drie componenten.

De kennisbronnen. De kennis die vervat zit in het expertsysteem wordt opgedeeld in onafhankelijke modules die kennisbronnen genoemd worden. Het zijn de kennisbronnen die de inhoud van het bord zullen manipuleren.

Het bord. Dit is de globale gegevensstructuur die de toestand van het expertsysteem bijhoudt. Het bord is de enige manier om informatie tussen kennisbronnen uit te wisselen.

De controle. Dit onderdeel van het bord staat in voor het activeren van de kennisbronnen als gevolg van de inhoud van het bord.

Deze drie componenten zijn niets anders dan de drie ingredënten van elk regelgebaseerd systeem, met name (i) de kennisbank, (ii) het werkgeheugen en (iii) de vertolker die instaat voor de selectie van de regels, het oplossen van conflicten en de uitvoering van de regels.

Multi-expertsystemen zijn wellicht de eerste toepassingen geweest die het bord niet enkel als werkgeheugen, maar ook als methode van communiceren gebruikten [FL77]. De kennisbronnen worden in dit geval parallelle taken. Ook de controle kon geparallelliseerd worden. De voornaamste voordelen van het gebruik van een bord in multi-expertsystemen zijn enerzijds de toegenomen verwerkingskracht door het gebruik van verscheidene processoren en anderzijds de mogelijkheid om kennisbronnen van totaal verschillende origine te combineren [Cor89]. In hoofdstuk 5 wordt er verder ingegaan op de prestaties van bordgebaseerde expertsystemen.

Linda

In het begin van de tachtiger jaren werd aan de universiteit van Yale het Linda-communicatieparadigma ontwikkeld [CG86a, Gel85, CGL86, ACG86]. Het was een eenvoudig alternatief voor de drie reeds bestaande modellen voor parallelle uitvoering, met name gemeenschappelijke variabelen, het doorgeven van boodschappen, en bewerkingen van elders.

Het Lindamodel bestaat uit een gemeenschappelijke veeltallenruimte die door alle taken gebruikt kan worden. De manipulaties van de veeltallenruimte zijn essentieel beperkt tot het toevoegen en het verwijderen van een veeltal. De beschikbare primitieven worden hierna opgesomd.

- `out()` wordt gebruikt om een veeltal in de veeltallenruimte op te nemen. Zo zal `out("P",5,vals)` het veeltal `("P",5,vals)` in de veeltallenruimte plaatsen.
- `in()` wordt gebruikt om een veeltal uit de veeltallenruimte te verwijderen. De argumenten van `in` bestaan uit waarden en formele parameters. Een waarde moet exact overeenstemmen met het corresponderende element van het veeltal terwijl een formele parameter met eender welke waarde van hetzelfde type kan overeenstemmen. Na de uitvoering van `in` zullen de formele parameters een waarde gekregen hebben. Zo zal de uitvoering van `in("P",int i,vals)` het veeltal `("P",5,vals)` uit de veeltallenruimte halen en de waarde `5` aan `i` toekennen.
- `read()` heeft een vergelijkbare werking als `in`, maar verwijdert het geselecteerde veeltal niet uit de veeltallenruimte. Het kent echter wel waarden

toe aan de formele parameters.

- `eval()` is vergelijkbaar met `out`, maar creëert een taak die eerst de componenten van het veeltal evalueert en pas nadat de evaluatie afgelopen is het veeltal in de veeltallenruimte zet. De oproep `eval("som",1+2)` zal een taak creëren om de uitdrukking `1+2` te evalueren en uiteindelijk het veeltal `("som",3)` aan de veeltallenruimte toe te voegen.

LINDA is een eenvoudig en elegant communicatieparadigma. Zijn belangrijkste kenmerken [Gel85] worden hierna besproken.

(i) *Ruimtelijke ontkoppeling.* Hiermee wordt bedoeld dat er geen enkele band meer bestaat tussen een gegeven in de veeltallenruimte en de taak die dat gegeven gegenereerd heeft. Zij hebben niets gemeenschappelijks meer. Een veeltal wordt gecopieerd en begint hierdoor aan een eigen bestaan.

(ii) *Tijdsontkoppeling.* Een veeltal wordt door een taak in de veeltallenruimte achtergelaten en hoeft niet meteen door een andere taak geconsumeerd te worden. Door de asynchrone communicatie kan een veeltal een bepaalde tijd in de veeltallenruimte verblijven.

(iii) *Gespreid gebruik van gegevens.* Een gegeven kan op verscheidene manieren in een veeltallenruimte opgeslagen worden. Door een gegeven niet als één veeltal, maar in stukjes op te slaan, kan men het ook in stukjes consumeren (eventueel door verscheidene taken). De veeltallenruimte garandeert dat alle bewerkingen atomair zullen plaatsvinden. In tegenstelling met de andere parallelle programmeertalen hoeven er hiervoor geen extra voorzieningen getroffen te worden.

Zoals hier beschreven is LINDA echter nogal beperkt om praktisch bruikbaar te zijn. Later werden er dan ook allerlei uitbreidingen gesuggereerd om de bruikbaarheid te verhogen zoals niet-blokkerende primitieven en meervoudige veeltallenruimten [Gel89].

In Linda 3 [Gel89] worden lokale veeltallenruimten als veeltal opgenomen in een andere veeltallenruimte. Een veeltallenruimte kan slechts gebruikt worden indien ze deel uitmaakt van een andere veeltallenruimte. Het verwijderen van een lokale veeltallenruimte uit de veeltallenruimte die ze bevat heeft de stopzetting van de taken die met deze lokale veeltallenruimte geassocieerd zijn als gevolg. In tegenstelling met de situatie in MULTI-PROLOG worden de taken echter niet vernietigd. Van zodra de veeltallenruimte opnieuw opgenomen wordt in een andere veeltallenruimte (en hierdoor actief wordt), zullen de taken gewoon verdergezet worden.

Deze aanpak is elegant, maar heeft toch ook zijn negatieve kanten. Hij is elegant omdat er niet echt bijkomende primitieven nodig zijn. Het enige bijkomende primitief is `tsc` dat een nieuwe veeltallenruimte aanmaakt. De

bewerking `out` plaatst de veeltallenruimte in een andere veeltallenruimte en maakt ze actief. De bewerking `in(?t)` doet alle taken die met de veeltallenruimte `t` verbonden zijn wachten. Een later uitgevoerde `out(t)` zal de taken gewoon doen verderlopen. Het nadeel van deze aanpak is echter dat (i) het 'doen stoppen' van taken nogal vaag is, en (ii) het gedrag van `in` en `out` nu afhangt van het type van veeltal waarop ze inwerken. Dit is nogal verwarrend voor een paradigma dat er prat op gaat het eenvoudigste te zijn.

Sicstus 2.1

De versie 2.1 van SICSTUS-PROLOG [AAF+91] bevat een module met Lindaprimitieven. Deze module staat echter volledig los van de taal zelf. Het is louter een uitbreiding aan de hand van een aantal ingebouwde predicaten.

- `out(t)` plaatst een term `t` op het bord.
- `in(t)` leest een term `t` van het bord op destructieve wijze, en blokkeert indien er geen geschikte term kan gevonden worden.
- `rd(t)` leest een term `t` van het bord zonder de term ook fysisch van het bord af te halen, en blokkeert indien er geen geschikte term kan gevonden worden.
- `in_noblock(t)` verwijdert een term `t` van het bord en faalt indien dit niet mogelijk is.
- `rd_noblock(t)` consulteert een term `t` op het bord en faalt indien er geen beschikbaar is.
- `in(l,e)` verwijdert één van de termen uit lijst `l` van het bord. Het argument `e` bevat de term die uiteindelijk verwijderd werd.

Uit de eerder summiere handleiding [AAF+91] is niet meteen duidelijk of er termen met variabelen kunnen gecommuniceerd worden. Wellicht is dit echter wel mogelijk. Of deze variabelen in deze termen dan nog enig verband hebben met de originele variabelen is niet duidelijk. Wellicht echter niet, om dezelfde reden als in MULTI-PROLOG[18].

Een programma kan van verscheidene borden gebruik maken. Een bord wordt geïmplementeerd als een bedienertaak. Een taak die wil communiceren met een bord moet zich vooraf kenbaar maken bij dat bord, en wordt een cliënt van het bord genoemd. Een cliënt kan slechts met één bord terzelfder tijd gekoppeld zijn.

[18] Om te vermijden dat er gemeenschappelijke variabelen tussen de individuele taken zouden ontstaan.

In vergelijking met MULTI-PROLOG is deze implementatie misschien wel algemener omdat ze kan uitgevoerd worden op een breed gamma van machines en gebruik maakt van een lokaal netwerk voor de communicatie tussen de taken, maar aan de andere kant mist ze een aantal elementen.

- Ze wordt een implementatie van LINDA genoemd, maar ze is slechts een gedeeltelijke implementatie. Zo werd het `eval`-predicaat niet geïmplementeerd.
- Bordcommunicatie is een vorm van communicatie op hoog niveau, maar toch wordt de implementatie zichtbaar. Een cliënt moet de fitting en de poort van de communicatie op het UNIX-niveau kennen om te kunnen communiceren. Een abstract bordgegevenstype was volgens ons een meer geslaagde keuze geweest. Bovendien wordt hiermee ook een hypotheek gelegd op implementaties die een alternatieve communicatiemethode willen kiezen.
- Er werd wel in een disjunctie van communicatieprimitieven voorzien, maar niet in een conjunctie. Het is met andere woorden niet mogelijk om twee termen simultaan van het bord af te halen.
- Er werd noch voorzien in terugzoekende communicatieprimitieven, noch in enige voorwaardelijke variant. Dit legt toch wel een beperking op de uitdrukkingskracht van de taal.

Prolog-Linda

PROLOG-LINDA [CWA+91, pro91] is een implementatie van LINDA in de logische programmeertaal μPROLOG [Nai85]. Er bestaan eigenlijk twee versies: PROLOG-1-LINDA voor een monoprocessor, en PROLOG-N-LINDA voor een multiprocessor. Deze implementaties van PROLOG-LINDA zijn origineel omdat ze gebruik maken van een deductieve veeltallenruimte. Dit wil zeggen dat er geen termen, maar bepalingen in de veeltallenruimte opgeslagen worden die naderhand door het programma kunnen gebruikt worden. De primitieven worden nu één na één besproken.

- `out/1` slaat een feit of een regel op in de veeltallenruimte. Dit werd geïmplementeerd aan de hand van het ingebouwd `assert`-predicaat.
- `in/1` verwijdert een feit of een regel uit de veeltallenruimte. Indien geen geschikte bepaling kan gevonden worden, dan zal de taak die deze bewerking uitvoert blokkeren.
- `rd/1` consulteert de bepalingen in de veeltallenruimte met de standaard berekeningsregel voor PROLOG. Hierdoor ontstaat een deductieve veeltallenruimte.

```
start(X) :-
   out((keuze(1) :- f(1))),
   out((keuze(2) :- f(2))),
   rd(keuze(X)),
   in(f(X)),
   in((keuze(1) :- f(1))),
   in((keuze(2) :- f(2))).
```

Programmafragment 2.23 Simultaan wachten op twee veeltallen in PROLOG-LINDA.

```
out((even(N) :- N<0, !, fail)).
out(even(0)).
out((even(N) :- N2 is N-2, even(N2)).
```

Programmafragment 2.24 Even getallen in de veeltallenruimte.

- `inp/1` is de niet-blokkerende variant van `in/1`.
- `rdp/1` is de niet-blokkerende variant van `rd/1`.
- `eval/2` creëert een nieuwe taak, uitgaande van een programmabestand en een initiële vraag. De uitvoering van deze taak zal slechts een bepaling in de veeltallenruimte achterlaten indien er bij de evaluatie van de initiële vraag expliciet een `out/1` uitgevoerd wordt.

De deductieve veeltallenruimte biedt een elegante oplossing voor een aantal moeilijk te programmeren problemen in LINDA. Twee ervan worden hier besproken. Het eerste betreft het wachten op twee veeltallen uit de veeltallenruimte. Een manier om te wachten op zowel `f(1)` als `f(2)` is weergegeven in programmafragment 2.23. Deze oplossing is eleganter dan de oplossing met de controleterm (zie blz. 39), maar veroorzaakt wel flink wat overlast omdat er een heus programma in de veeltallenruimte moet geplaatst worden alvorens een term te kunnen lezen. Bovendien kan dit programmafragment problemen geven indien het simultaan door twee taken zou uitgevoerd worden. De uitvoering van `rd(keuze(X))` en `in(f(X))` worden immers voor zover ons bekend niet atomair uitgevoerd.

Het programmafragment 2.24 slaat alle even getallen op in de veeltallenruimte. Dit is moeilijk te realiseren in een klassieke veeltallenruimte omdat dit zou vereisen dat inderdaad alle even getallen in de veeltallenruimte zouden moeten opgenomen worden en dit is natuurlijk onmogelijk. Een alternatieve methode is wel om de even getallen op aanvraag door een taak te laten genereren op het bord. Dit wordt geïllustreerd in programmafragment 5.8 in hoofdstuk 5.

Hoe krachtig een deductieve veeltallenruimte ook moge zijn, van de kant van de implementatie zijn er toch wel een aantal ernstige bemerkingen.

1. Het opslaan van bepalingen in plaats van termen veroorzaakt een aanzienlijke overlast omdat de veeltallenruimte nu niet enkel moet gesynchroniseerd worden ten opzicht van de andere Lindaprimitieven, maar ook ten opzichte van de Prologvertolker. Er kan immers niet zomaar een bepaling die op hetzelfde ogenblik door een taak gebruikt wordt uit de veeltallenruimte weggehaald worden.

2. De uitvoeringstijd van de `rd/1` en `rdp/1` primitieven wordt moeilijk te voorspellen omdat dit eigenlijk μPrologprogramma's geworden zijn. Het is zelfs mogelijk dat een dergelijk programma nooit termineert en daardoor de veeltallenruimte voor altijd vergrendeld zal houden.

Dit is meteen de voornaamste kritiek op LINDA-PROLOG. De communicatie bevat te veel logische aspecten waardoor ze moeilijk efficënt geïmplementeerd kan worden. Een correcte en betrouwbare implementatie van de communicatieprimitieven vereist dat de volledige veeltallenruimte moet vergrendeld worden voor een tijd die op voorhand niet gekend is, maar zal afhangen van het doel dat in de veeltallenruimte geëvalueerd wordt.

Prolog-D-Linda

De programmeertaal PROLOG DISTRIBUTED LINDA [SP91] is een toepassing van het Lindacommunicatieparadigma op SICSTUS PROLOG [SP91]. De implementatie is vergelijkbaar met die van LINDA-PROLOG, maar dan toegepast op SICSTUS-PROLOG. Ze bestaat uit één zogenaamde controletaak die instaat voor communicatie met de buitenwereld en voor het opstarten van de toepassing. De veeltallenruimte is gespreid over een aantal bedienertaken. Het programma wordt gespreid uitgevoerd door een aantal clënttaken. Het `eval`-primitief wordt gebruikt om nieuwe cliënttaken in het leven te roepen. Deze `eval`-primitieven worden allemaal verwerkt door de zogenaamde `eval`-bedienertaak. De informatie over de verdeling van de veeltallenruimte wordt bijgehouden door de controletaak en door ieder van de clënttaken.

De gegevens worden opgeslagen als Prologbepalingen in de Prologgegevensbank. Zowel feiten als regels kunnen op die manier opgeslagen worden, hetgeen de veeltallenruimte deductief maakt. Twee toepassingen zijn bekend: een genetisch algoritme en een systeem voor automatische deductie.

De voor- en nadelen van deze taal zijn nagenoeg dezelfde als deze van LINDA-PROLOG.

BinProlog

BINPROLOG [Tar92] is een kleine en snelle Prologcompilator die gebaseerd is op de transformatie van PROLOG tot binaire bepalingen, ontwikkeld aan de

```
wachter invoer | doel succes_uitvoer ; faling_uitvoer
```

Afbeelding 2.7 De structuur van een SHARED PROLOG patroon.

universiteit van Moncton in Canada door Paul Tarau. BINPROLOG is vrij van neveneffecten zoals `assert` en `retract` maar gebruikt de Lindaprimitieven als vervanging ervan en om parallellisme te ondersteunen. Voor het overige is over de implementatietoestand van deze primitieven weinig geweten behalve dan het feit dat ook zij gebruik (zullen) maken van een lokaal netwerk om te communiceren.

Shared Prolog

SHARED PROLOG [Cia89a, ACCD89, Cia89b, Cia89c, CC91] is een bordgebaseerde taal die nogal afwijkt van wat in de andere Lindagebaseerde systemen gangbaar is. De taal werd ontwikkeld aan de universiteit van Pisa door A. Brogi en P. Ciancarini. Een programma in SHARED PROLOG bestaat uit een bord en een verzameling van theorieën. Een theorie bestaat uit een aantal *patronen* en een Prologprogramma. Afbeelding 2.7 toont de structuur van een patroon. Het bestaat uit de volgende componenten

(i) een `wachter` die waar moet zijn opdat het patroon zou kunnen geactiveerd worden;

(ii) een lijst van bordtermen `invoer` die atomair moet kunnen geconsumeerd worden opdat een patroon zou kunnen geactiveerd worden.

(iii) een engagementssymbool (`|`);

(iv) het `doel` of de romp van het patroon, d.w.z. het stukje Prologprogramma dat zal uitgevoerd worden van zodra het patroon geactiveerd wordt;

(v) een verzameling van termen `succes_uitvoer` die op het bord gezet wordt indien de evaluatie van `doel` slaagt;

(vi) een verzameling van termen `faling_uitvoer` die op het bord gezet wordt indien de evaluatie van `doel` faalt.

De zogenaamde *preactivering* (alles wat voor de geëngageerde keuze komt) wordt atomair uitgevoerd. Het programmafragment 2.25 beschrijft een tellerobject dat twee bewerkingen kan ondergaan (`plus_een` en `min_een`). De naam en de waarde van de teller wordt op het bord bijgehouden als term `teller(Naam,N)`. Van zodra een bewerking (b.v. `plus_een/1`) en een teller (`teller/2`) van het bord kunnen afgehaald worden, wordt de betreffende bewerking uitgevoerd, en wordt de nieuwe waarde van de teller op het bord geplaatst. Het simultaan lezen van twee termen stelt absoluut geen probleem.

```
teller :-
  {plus_een(Naam), teller(Naam,N)}
  |
  plus_een(N,NieuweN)
  {teller(Naam,NieuweN)} ; { }
*
  {min_een(Naam), teller(Naam,N)}
  |
  min_een(N,NieuweN)
  {teller(Naam,NieuweN)} ; { }
with
  plus_een(N,NieuweN) :- NieuweN = N+1.
  min_een(N,NieuweN) :- NieuweN = N-1.
```

Programmafragment 2.25 Een tellerobject in SHARED PROLOG.

Ook het simultaan wachten op twee leesbewerkingen stelt geen probleem omdat dit kan geïmplementeerd worden als twee patronen. Alle patronen worden immers of-parallel uitgevoerd. Het patroon waarvan de preactivering als eerste slaagt, zal de evaluatie van de andere patronen stoppen en mutueel exclusief de corresponderende *postactivering* uitvoeren.

Het tellerobject kan wel een flessehals vormen indien het veel tellers moet bijhouden. De oplossing in MULTI-PROLOG waar één taak per teller gebruikt wordt is heel wat flexibeler. Het creëren van één taak per teller is moeilijk in SHARED PROLOG omdat de taken statisch gecreëerd moeten worden.

Als besluit kan gesteld worden dat het taalontwerp van SHARED PROLOG eenvoudig en toch bijzonder krachtig is. Het patroonconcept is zowat in staat om alle voorwaarden op het bord op een elegante manier uit te drukken.

In vergelijking met MULTI-PROLOG heeft SHARED PROLOG echter een viertal zwakke punten: (i) er is geen mogelijkheid om op een dynamische manier bijkomende taken of patronen te creëren; (ii) de uitvoering van de preactivering zal altijd relatief traag zijn omdat ze vrij complex is; (iii) de syntaxis wijkt nogal af van die van PROLOG, en (iv) de communicatie met het bord is beperkt tot de preactivering en het einde van de postactivering. Op andere plaatsen kan er niet met het bord gecommuniceerd worden.

Polis Prolog Deze bordgebaseerde PROLOG [Cia91] houdt het midden tussen SHARED PROLOG en LINDA met meervoudige veeltallenruimten. Enerzijds breidt ze de taal SHARED PROLOG uit door de invoering van meervoudige borden, en anderzijds beperkt ze de mogelijkheden van het Lindaparadigma door de plaatsen waar er gecommuniceerd kan worden te beperken tot het begin en het einde van een programmaveeltal (zoals in SHARED PROLOG ook reeds het geval is). Verder gelden dezelfde opmerkingen als voor SHARED PROLOG.

```
p(X,Y) :- q(X), q(Y).
q(a).
q(b).
```

Programmafragment 2.26 Demonstratieprogramma voor het `eval`-predicaat.

```
:- op(950,xf,'!&').

!&X :- eval(X)&.
eval(X) :- call(X), !X, fail.
```

Programmafragment 2.27 Simulatie van het `eval`-predicaat.

2.8 Bespreking

Uit het overzicht van de verwante talen blijkt dat nagenoeg alle concepten die daar aanwezig zijn ook in MULTI-PROLOG beschikbaar zijn. De enige uitzondering is wellicht het `eval`-primitief van `Linda`. Dit primitief wordt in de eerste plaats gebruikt om taken te creëren, maar heeft finaal ook een effect op het bord. MULTI-PROLOG heeft een primitief om een taak te creëren, maar dit heeft niet hetzelfde effect op het bord.

Aan het `eval`-primitief zou in MULTI-PROLOG de volgende betekenis kunnen gegeven worden. Een doel wordt omgezet in een taak; deze taak wordt vervolgens uitgevoerd, en als resultaat wordt een geconcretiseerde versie van het doel op het bord geplaatst. Indien het doel niet verwezenlijkt kan worden, dan wordt er geen term op het bord geplaatst.

Dit bordprimitief krijgt als syntaxis `!&doel` mee. Een voorbeeld in programmafragment 2.26 kan het een en het ander verduidelijken. De vraag `?- !&p(X,Y).` zal na verloop van tijd aanleiding geven tot het verschijnen van de termen `p(a,a)`, `p(a,b)`, `p(b,a)` en `p(b,b)` op het bord. Deze oplossingen kunnen naderhand van het bord afgehaald worden aan de hand van `?p(X,Y)`. Dit primitief kan geïmplementeerd worden zoals in programmafragment 2.27. Het voornaamste nadeel van dit primitief is dat men niet kan weten wanneer deze taak werkelijk beëindigd is. Als er geen oplossing op het bord gevonden wordt kan men niet weten of er nog een oplossing komt, of of de taak reeds gestopt is. Dit beperkt het nut van dit primitief. Enkel indien men op voorhand het aantal oplossingen kent kan men blokkerend wachten totdat alle oplossingen op het bord gegenereerd werden.

Het primitief simuleert in dat geval echter wel een soort van doelparallellisme zoals getoond wordt in programmafragment 2.28. Indien zowel `p/1` als `q/1` minstens één oplossing genereren, dan zal het programma gebaseerd op `eval` er zeker ook één genereren. De vraag `start/2` kan op zijn beurt opnieuw via `!&start(X,Y)` verwezenlijkt worden waardoor een hele boom van taken ontstaat.

```
start(X,Y) :- !p(X)&, !q(Y)&, ?p(X), ?q(Y).
```

Programmafragment 2.28 Een voorbeeld van het gebruik van `eval`.

Samenvatting

Dit hoofdstuk bespreekt informeel het ontwerp van de taal MULTI-PROLOG. De twee vertrekpunten, namelijk taakparallellisme en expliciete bordcommunicatie worden toegelicht, en later uitwerkt. Dit geeft aanleiding tot de zogenaamde kerntaal. De taakcreatie, de schrijfbewerking en de leesbewerking worden voorgesteld en gedetailleerd besproken. Een aantal voorbeelden geeft een idee van hoe ze in de praktijk kunnen gebruikt worden.

Vervolgens wordt de kerntaal uitgebreid met een viertal nieuwe elementen, met name een voorwaardelijke schrijfbewerking, een voorwaardelijke leesbewerking, modules en meervoudige borden. Een overzicht van verwante talen toont aan dat deze uitgebreide taal nagenoeg alle functionaliteit van deze talen ter beschikking stelt.

3 Semantiek

Semantics is what we have in our heads;
as soon as we write it down it is not semantics anymore.

— K. SAMUELSON

Het doel van dit hoofdstuk is om op een meer formele manier de bord-communicatie, en haar effect op de taal PROLOG *te bestuderen. In de inleiding wordt er geschetst op welke manier de semantiek van een logische programmeertaal traditioneel bestudeerd wordt. Nadien wordt er aangegeven welke problemen er ontstaan door de invoering van bord-communicatieprimitieven. Nadat een aantal basisbegrippen verklaard werden, worden twee semantische beschrijvingen gegeven: (i) een operationele semantiek die de uitvoering van een Multi-Prologprogramma tracht te modelleren, en (ii) een declaratieve semantiek die de logisch-declaratieve betekenis van het programma tracht te vatten.*

3.1 Inleiding

Semantiek wordt soms bestempeld als de tegenhanger van syntaxis. In tegenstelling met de syntaxis die zich bezig houdt met de vorm, houdt de semantiek zich bezig met de betekenis van een uitdrukking. Deze betekenis komt tot uiting in de vorm van de resultaten bij de evaluatie. Leslie Lamport beschrijft in [Lam84] het doel van semantiek als volgt:

> Het doel van semantiek is om aan elk syntactisch correct programma een wiskundige betekenis te hechten die het effect van de uitvoering ervan beschrijft.

De semantiek moet een formele basis verschaffen, maar hoeft niet noodzakelijk ook een praktisch bruikbare methode te zijn om b.v. eigenschappen van een programma af te leiden. De semantiek mag met andere woorden gerust gebruik maken van oneindige verzamelingen en dergelijke meer.

In dit proefschrift zal de semantiek hoofdzakelijk gebruikt worden om aan te tonen dat de betekenis van een Multi-Prologvraag in feite niet verschillend is van de betekenis van een Prologvraag gezien zij op dezelfde manier kan uitgedrukt worden.

3.2 De Semantiek van Hornbepalingen

Deze sectie verklaart een aantal belangrijke basisbegrippen uit de semantiek van verzamelingen van Hornuitdrukkingen. Deze sectie werd hier met opzet ingelast om de lezer vertrouwd te maken met de terminologie en de begrippen die gangbaar zijn bij de studie van semantiek. Het verschaffen van inzicht is hierbij belangrijker dan de wiskundige rigoureusheid. Deze sectie is vooral gebaseerd op het boek van Lloyd [Llo87].

Definitie 1 (Taal) *Een* taal *bestaat uit de verzameling van uitdrukkingen die opgebouwd kunnen worden aan de hand van een alfabet.*

In het geval van Hornbepalingen bestaat het alfabet uit een aantal leestekens, logische voegwoorden ($\wedge$, $\vee$, $\neg$, $\leftarrow$), namen van constanten, functies, predicaten en variabelen.

In een *logische* taal zijn we vooral geïnteresseerd in het al dan niet waar zijn van een uitdrukking. Van een logische uitdrukking zoals $c \vee d$ kan de waarheid niet zomaar achterhaald worden zonder eerst een betekenis te hechten aan de symbolen die in de uitdrukking voorkomen. Men gebruikt een *interpretatie* om een betekenis het geven aan deze symbolen.

Definitie 2 (Toepassingsdomein) *Een* toepassingsdomein *is een verzameling T van mogelijke waarden voor de variabelen.*

Definitie 3 (Interpretatie) *Een* interpretatie *bestaat uit*

- *een niet-lege verzameling T die het toepassingsdomein genoemd wordt,*
- *een functie die de constanten*[1] *uit een uitdrukking afbeeldt op elementen uit het toepassingsdomein T,*
- *een functie die functies uit een uitdrukking afbeeldt op functies over het toepassingsdomein T.*
- *een functie die predicaten uit een uitdrukking afbeeldt op relaties over het toepassingsdomein T.*

Een interpretatie geeft een betekenis aan alle symbolen in een logische uitdrukking, behalve dan de variabelen. In dit kader is het niet nodig een waarde

[1] Sommige auteurs beschouwen de constanten als functies zonder argumenten.

toe te kennen aan de variabelen omdat deze universeel gekwantificeerd zijn en dus niet vrij zijn.

Een uitdrukking kan enkel geëvalueerd worden voor een bepaalde interpretatie omdat dit de enige manier is om aan de symbolen uit een uitdrukking een betekenis te hechten. Het is meteen duidelijk dat er verscheidene interpretaties per toepassingsdomein kunnen gemaakt worden en dat een uitdrukking kan geïnterpreteerd worden voor verscheidene toepassingsdomeinen. Voor sommige interpretaties zal een uitdrukking waar zijn, voor andere dan weer vals. Een interpretatie waarvoor een uitdrukking waar is wordt een model voor die uitdrukking genoemd.

Definitie 4 (Model) *Een model voor een uitdrukking is een interpretatie waarvoor de uitdrukking waar is. Een model is een model voor een verzameling van uitdrukkingen indien het een model is voor elk van de uitdrukkingen uit de verzameling.*

Definitie 5 (Verwezenlijkbaarheid) *Een verzameling van uitdrukkingen wordt* verwezenlijkbaar *genoemd indien er minstens één model voor bestaat.*

Definitie 6 (Geldigheid) *Een verzameling van uitdrukkingen wordt* geldig *genoemd indien eender welke interpretatie er een model voor is.*

Een voorbeeld van een dergelijke verzameling is $\{p \vee \neg p\}$.

Definitie 7 (Strijdigheid) *Een verzameling van uitdrukkingen wordt* strijdig *genoemd indien er geen model voor bestaat.*

Een voorbeeld van een dergelijke verzameling is $\{p \wedge \neg p\}$.

Definitie 8 (Logisch gevolg) *Een uitdrukking U is een* logisch gevolg *van een verzameling van uitdrukkingen V indien elk model voor V ook een model is voor U.*

Het bewijs dat een uitdrukking een logisch gevolg is, is met deze definitie verre van triviaal. Het aantal mogelijke modellen is immers meestal oneindig groot. Daarom wordt er uitgekeken naar een alternatieve methode. Een eerste stap in de goede richting is de volgende stelling.

Stelling 1 (Bewijs door weerlegging) *Een uitdrukking U is een logisch gevolg van een verzameling van uitdrukkingen V als en slechts als de verzameling $V \cup \{\neg U\}$ strijdig is.*

Bewijs Het bewijs is nagenoeg triviaal. Indien U een logisch gevolg is van V, dan moet elk model voor V ook een model zijn voor U en kan het dus geen

model zijn voor $\neg U$. De verzameling $V \cup \{\neg U\}$ moet dus noodzakelijk strijdig zijn.

Anderzijds, indien U geen logisch gevolg is van V, dan moet $\neg U$ een logisch gevolg van zijn van V, en zal $V \cup \{\neg U\}$ dus niet strijdig zijn omdat elk model voor V meteen ook een model zal zijn voor $\neg U$. □

Het bewijs dat een uitdrukking een logisch gevolg is van een verzameling van uitdrukkingen is ook met deze stelling nog steeds een onbegonnen werk. Het bewijs dat een verzameling van uitdrukkingen strijdig is vereist immers dat er nagegaan wordt dat er geen model bestaat. Dit wil zeggen dat men voor alle mogelijke interpretaties moet bewijzen dat ze geen model zijn.

Gelukkig is er de stelling van Herbrand die stelt dat men zich bij het bewijs van de strijdigheid mag beperken tot de zogenaamde Herbrandinterpretaties. We beginnen met de definitie van twee verzamelingen.

Definitie 9 (Herbranduniversum) *Het* Herbranduniversum H_U *is de verzameling van alle volledig geconcretiseerde termen die in een taal voorkomen.*

Definitie 10 (Herbrandbasis) *De* Herbrandbasis H_B *is de verzameling van alle volledig geconcretiseerde atomen die in een taal voorkomen.*

Definitie 11 (Herbrandinterpretatie) *Een* Herbrandinterpretatie *wordt gedefinieerd als volgt:*

- *als toepassingsdomein wordt het Herbranduniversum* H_U *gekozen,*
- *de constanten worden afgebeeld op de corresponderende constanten in* H_U*,*
- *de functies* f *worden afgebeeld op de corresponderende functies:* $(H_U)^n \rightarrow H_U$ *die* $(t_1, \ldots, t_n)$ *afbeelden op* $f(t_1, \ldots, t_n)$*,*
- *de predicaten worden afgebeeld op een relatie over* H_U*.*

Een Herbrandinterpretatie kan beschouwd worden als een 'abstracte' interpretatie. Hierbij krijgen de symbolen uit een taal geen concrete waarden (zoals het getal 1 of de constante 'maandag'), maar representeren zij zichzelf. Deze interpretatie laat een aantal vereenvoudigingen toe.

Stelling 2 *Een Herbrandinterpretatie is een deelverzameling van de Herbrandbasis* H_B*.*

Bewijs Een Herbrandinterpretatie bestaat uit drie vaste componenten enéén variabele component. De vaste componenten zijn het toepassingsdomein H_U, de afbeelding van de constanten en de afbeelding van de functies. De enige variabele component is de afbeelding van de predicaten.

De afbeelding van de predicaten op een relatie over H_U drukt uit welke atomen waar zijn en welke atomen vals zijn. Doordat de Herbrandbasis alle

atomen bevat, zal de verzameling van atomen die waar zijn een deelverzameling van de Herbrandbasis zijn. Alle atomen die dan niet tot deze deelverzameling behoren worden als vals beschouwd.

Anderzijds kan, gegeven een deelverzameling van de Herbrandbasis, steeds een relatie aangemaakt worden waar de atomen uit deze deelverzameling deel van uitmaken. □

Verder geldt dat een Herbrandinterpretatie alle kenmerken van een gewone interpretatie heeft. Herbrandmodellen e.d.m. worden op de klassieke manier gedefinieerd.

Belangrijk is wel de stelling van Herbrand die stelt dat van zodra er een model voor een uitdrukking bestaat er ook een Herbrandmodel moet bestaan. Dit is niet echt verwonderlijk omdat alle atomen die in het model waar zijn automatisch in de Herbrandbasis voorkomen. De deelverzameling van deze atomen levert dan een Herbrandmodel op.

Eerder in deze sectie werd een model gedefinieerd als een interpretatie waarvoor een uitdrukking waar is. De verzameling van modellen is een deelverzameling van de verzameling van interpretaties. Het aantal mogelijke modellen is echter doorgaans nog steeds oneindig. Dit komt omdat het model informatie kan bevatten die niet relevant is voor de uitdrukking waarop het slaat. Deze overtollige informatie kan in principe weggelaten worden. Het *minimaal model* wordt gedefinieerd als de doorsnede van alle Herbrandmodellen en bevat dus enkel die informatie die gemeenschappelijk is aan alle modellen. Het blijkt dat dit minimale model precies overeenstemt met de verzameling van logische gevolgen die uit een verzameling van uitdrukkingen V voortvloeien. Immers, een logisch gevolg moet waar zijn voor alle modellen van V, en dus ook voor hun doorsnede. Elk logisch gevolg behoort dus tot het minimale Herbrandmodel.

We stappen nu over naar logische programma's door de verzameling van uitdrukkingen te beperken tot een verzameling van bepalingen en deze verzameling een programma te noemen. Een vraag wordt genoteerd als $\neg U$.

De declaratieve semantiek van een programma P wordt dan gedefinieerd als alle atomen die een logisch gevolg zijn van P, d.w.z. het minimale Herbrandmodel.

Voor de studie van een programma P met een vraag $\neg U$ waarin er veranderlijken voorkomen, moet men gebruik maken van substituties. Een atoom kan immers slechts een logisch gevolg zijn van een programma indien het volledig geconcretiseerd is (althans volgens de zonet uiteengezette theorie). Een substitutie is een verzameling van koppels bestaande uit een variabele en een term. Een dergelijk koppel drukt de waarde van een veranderlijke uit. In dit kader zijn we vooral geïnteresseerd in een speciaal soort van substitutie die een *unificator* genoemd wordt.

Definitie 12 (Unificator) *Gegeven een eindige verzameling V van termen en atomen wordt de substitutie θ een* unificator *voor V genoemd indien er geldt dat $V\theta$ een singleton is.*

Uit deze definitie volgt dat er niet één maar oneindig veel unificatoren voor een gegeven verzameling bestaan. Een meest algemene unificator, afgekort *mgu*[2] is een unificator θ waarvoor geldt dat er voor alle andere unificatoren σ voor V een substitutie τ kan gevonden worden zodat $\sigma = \theta\tau$. Vaak spreekt men over de meest algemene unificator omdat alle meest algemene unificatoren door de variabelen te hernoemen identisch kunnen gemaakt worden. Een voorbeeldje kan het één en het ander verduidelijken. Een mgu van de verzameling

$$\{f(X, 1), f(Y, Y)\}$$

wordt gegeven door $\{X/Y, Y/1\}$ omdat het resultaat van de toepassing van deze mgu op de verzameling gelijk is aan $\{f(1, 1)\}$. Daarentegen heeft de verzameling

$$\{f(X, 1), f(Y, 2)\}$$

geen unificator omdat er geen enkele substitutie kan gevonden worden die de beide termen uit de verzameling identisch maakt.

De evaluatie van een vraag met variabelen zal niet enkel het antwoord waar of vals opleveren, maar moet bovendien ook een substitutie opleveren die een waarde geeft aan de veranderlijken die in de vraag voorkomen. Dit brengt ons bij de volgende definitie.

Definitie 13 (Correcte antwoordsubstitutie) *Gegeven een programma P en een vraag $\neg U$, dan geldt dat een* correcte antwoordsubstitutie *een substitutie θ is waarvoor $U\theta$ een logisch gevolg is van P.*

Een correcte antwoordsubstitutie is de manier om op declaratieve manier de betekenis van een programma weer te geven.

De operationele semantiek is een formalisering van het algoritme om een substitutie te *berekenen*.

Definitie 14 (Afleiding) *Gegeven een vraag $\neg U : \neg(a_1 \wedge \ldots \wedge a_i \wedge \ldots \wedge a_n)$ en een bepaling $B : p \leftarrow d_1 \wedge \ldots \wedge d_m$ geldt dat $\neg U'$ kan afgeleid worden uit $\neg U$ en B als volgt:*

- *a_i is het geselecteerde atoom,*
- *θ is de mgu(a_i,p), en*
- *$\neg U'$ is de vraag $\neg(a_1 \wedge \ldots \wedge a_{i-1} \wedge d_1 \wedge \ldots \wedge d_m \wedge a_{i+1} \wedge \ldots \wedge a_n)\theta$.*

[2] Dit is de afkorting van het Engelse *most general unifier*.

Een SLD-afleiding[3] bestaat uit een rij van vragen $\neg U, \neg U_1, \ldots$ zodat U_{i+1} kan afgeleid worden uit U_i en een programma P volgens definitie 14. Een *SLD-weerlegging* voor $P \cup \{\neg U\}$ is een eindige SLD-afleiding $\neg U, \neg U_1, \ldots, \neg U_n$ waarvoor geldt dat $U_n = \delta$.[4] Een SLD-weerlegging levert een substitutie op die de samenstelling is van de substituties die door de individuele afleidingen gegenereerd worden.

Definitie 15 (berekende antwoordsubstitutie) *Gegeven een programma P en een vraag $\neg U$ geldt er dat de* berekende antwoordsubstitutie *de restrictie is van de substitutie van de SLD-weerlegging voor $P \cup \{\neg U\}$ tot de variabelen in U.*

Het sluitstuk van deze studie is dan het bewijs van correctheid en volledigheid. Correctheid vereist dat een berekende oplossing ook een correcte oplossing is. In de zonet beschreven termen betekent dit dat elke berekende antwoordsubstitutie ook een correcte antwoordsubstitutie is. Volledigheid betekent dat alle oplossingen ook effectief berekend kunnen worden. Technisch betekent dit dat er voor elke correcte antwoordsubstitutie ook een berekende antwoordsubstitutie bestaat.

De bewijzen van correctheid en volledigheid worden hier niet gegeven maar kunnen in de literatuur teruggevonden worden [Llo84].

Onder de declaratieve en operationele semantiek van een vraag (m.b.t. een programma P) wordt de verzameling van alle correcte resp. berekende antwoordsubstituties verstaan. Deze verzameling is equivalent met de verzameling van atomen die een logisch gevolg zijn van het programma resp. een SLD-weerlegging bezitten.

3.3 Een Multi-Prologvoorbeeld

De klassieke aanpak zoals beschreven in de vorige sectie zal niet gebruikt kunnen worden in de semantiek voor MULTI-PROLOG omwille van de neveneffecten. De bordcommunicatieprimitieven veroorzaken immers neveneffecten die een niet geringe invloed hebben op het gedrag van het programma. Dit is echter volstrekt normaal gezien zij precies gebruikt worden om het gedrag van het programma te beïnvloeden. Zoniet zou de communicatie tussen de taken niet veel zin hebben. Programmafragment 3.1 wordt gebruikt om het één en het ander te verduidelijken.

Het gedrag van dit programma zal nu in twee omstandigheden bestudeerd worden: ofwel is het bord leeg (initële situatie), ofwel bevat het bord reeds een

[3] SLD staat voor het Engelse Linear Resolution with Selection function for Definitie clauses. Op de betekenis hiervan gaan wij niet dieper in.

[4] Het lege doel wordt δ genoteerd.

```
p :- r(X), !X.
q :- ?X, r(X).

r(a).
r(b).
```

Programmafragment 3.1 Voorbeeld van een Multi-Prologprogramma.

hoeveelheid informatie. Het gedrag zal in de twee gevallen grondig verschillen. De feiten `r(a)` en `r(b)` zijn er enkel om het programma niet te triviaal te maken. Zij zullen verder niet besproken worden. Het gedrag van een viertal vragen zal nu gedetailleerd besproken worden. Merk terloops op dat `?` de blokkerende, destructieve, niet-terugzoekende variant van het leespredicaat is.

Geval 1: het bord is leeg.

- De vraag `?- p.` kan met succes geëvalueerd worden. Na de evaluatie zal wel een residu (de term `a`) op het bord overblijven.
- De evaluatie van de vraag `?- q.` blokkeert totdat er een term op het bord verschijnt. Gezien er geen andere taken bestaan zal deze evaluatie nooit termineren.
- De vraag `?- p,q.` zal met succes geëvalueerd worden.
- De evaluatie van de vraag `?- q,p.` blokkeert op het lege bord. Dit is te verwachten gezien de uitvoering van links naar rechts gebeurt en `q` op zich al blokkeert. Merk op dat de conjunctie uit dit en het vorige voorbeeld dus niet commuteert.

Geval 2: het bord bevat minstens één term.

- De vraag `?- p.` evalueert nog steeds met succes, ongeacht de inhoud van het bord.
- De vraag `?- q.` kan enkel met succes geëvalueerd worden indien er ofwel een term `a` of een term `b` als oudste term op het bord voorkomt. Indien er een term verschillend van `a` of `b` als oudste voorkomt, zal `q` blokkeren. Hier zien we reeds duidelijk dat de betekenis van een vraag tijdsafhankelijk is. Het is dus van belang `?- q.` op het gepaste ogenblik te evalueren, d.w.z. op het ogenblik dat `a` of `b` als oudste term op het bord voorkomt.
- De vraag `?- p,q.` zal enkel met succes geëvalueerd kunnen worden indien de oudste term op het bord `a` of `b` is. Op een leeg bord zal de term

die door q geconsumeerd wordt door p geleverd worden. Op een niet-leeg bord zal de term die door p geleverd wordt de jongste term zijn. Het doel q leest echter steeds de oudste term.

- Het resultaat van `?- q,p.` zal ook hier weer afhangen van de inhoud van het bord. Enkel indien de oudste term een a of een b is, zal deze berekening tot een goed einde kunnen gebracht worden.

Het is duidelijk dat de aanwezigheid van bordcommunicatieprimitieven de semantiek van PROLOG heel wat complexer maakt. De aanwezigheid van initële termen op het bord maakt het enkel maar erger. In dit hoofdstuk zal er gepoogd worden om een zo eenvoudig mogelijk model op te zetten dat ten minste de zonet beschreven effecten op elegante manier beschrijft en verklaart.

Om de complexiteit van de semantiek zoveel mogelijk te beperken, zullen enkel de taakcreatie, een schrijfbewerking, en een destructieve en niet-destructieve variant van een blokkerende niet-terugzoekende leesbewerking behandeld worden. De keuze voor de niet-terugzoekende leesbewerkingen werd gemaakt om (i) de semantiek eenvoudiger te maken en (ii) omdat het terugzoeken essentieel weinig nieuwe aspecten aan de semantiek van bordprimitieven toevoegt. Er werd gekozen voor de blokkerende varianten omdat deze wel afwijken van het gewone gedrag van Prologpredicaten, en dus een aantal nieuwe aspecten kunnen introduceren.

3.4 Enkele Basisbegrippen

De taal MULTI-PROLOG is opgebouwd uit dezelfde elementen als de taal PROLOG. Variabelen, functies, predicaten, termen, atomen en substituties worden dan ook op de klassieke manier gedefinieerd [Llo87, DB90b], net zoals het *Herbranduniversum* H_U, en de *Herbrandbasis* H_B. De verzameling van alle termen wordt S_t genoteerd en verzameling van alle atomen wordt S_a genoteerd[5]. De verzameling van substituties wordt S_θ genoteerd. Het symbool ϵ stelt de lege substitutie voor.

De definitie van doel dient echter uitgebreid te worden. Dit kan aan de hand van de volgende mutueel recursieve definities.

Definitie 16 (Doel) *Een* doel *wordt als volgt gedefinieerd:*

(i) δ *is het lege doel;*
(ii) elk element van S_a *is een doel;*
(iii) een borddoel is een doel;
(iv) indien d_1 *en* d_2 *doelen zijn, dan is de conjunctie* d_1, d_2 *ook een doel.*

[5] Deze verzamelingen bevatten ook alle termen en atomen met variabelen.

De verzameling van doelen wordt S_d genoteerd.

Definitie 17 (Borddoel) *Een* borddoel *wordt als volgt gedefinieerd:*

(i) `!t, ?t` *en* `?!t` *met* `t`$\in S_t$ *zijn borddoelen;*
(ii) `d&` *met* `d`$\in S_d$ *is een borddoel.*

Een programma wordt als volgt gedefinieerd:

Definitie 18 (Programma) *Een programma P is een verzameling van bepalingen van de vorm* `a:-d` *met* `a`$\in S_a$, *en* `d`$\in S_d$. *De verzameling van alle programma's wordt S_P genoteerd*[6].

Definitie 19 (Bord) *Een* bord *is een lijst van elementen uit $S_t \cup S_d$. De verzameling van alle borden wordt S_b genoteerd.*

Deze bordlijst wordt genoteerd tussen vierkante haakjes. Zo is [ ] het lege bord en $[a, b]$ het bord met twee elementen a en b. Hierbij is a het oudste element en b het meest recente. Voor het gemak van notatie wordt met $l_1 + l_2$ de aaneenschakeling van de lijsten l_1 en l_2 bedoeld.

Merk op dat een bord uit zowel termen als doelen bestaat. Hierdoor kan de creatie van een taak beschouwd worden als een communicatie met het bord. Het correspondeert met het plaatsen van een doel op het bord.

Om het effect van de borddoelen op het bord te beschrijven wordt hier het begrip van *bordgebeurtenis* geïntroduceerd. Een bordgebeurtenis is in staat om de inhoud van het bord te veranderen, en wordt daarom gedefinieerd als een functie.

Definitie 20 (Bordgebeurtenis) *Een* bordgebeurtenis *wordt gedefinieerd als een partiële functie: $S_b \rightarrow S_b$ die als volgt gedefinieerd wordt: gegeven $g \in (S_t \cup S_d)$ en $b, b' \in S_b$*

- $g^+(b) = b + [g]$
- $g^-(b) = b'$ *indien er in b een element kan gevonden worden dat unificeert met g, met t het oudste dergelijke element in b en met b' gelijk aan b waaruit t verwijderd werd.*
- $g^*(b) = b$ *indien er in b een element kan gevonden worden dat unificeert met g.*

De verzameling van bordgebeurtenissen S_g wordt gegenereerd door $(S_t \cup S_d) \times \{+, -, *\}$.

De relatie tussen de borddoelen en de bordgebeurtenissen wordt weergegeven in tabel 3.1. Termen en doelen worden onderscheiden door hun type.

Tabel 3.1 Borddoelen en bordgebeurtenissen.

Doel	Gebeurtenis
`!t`	t^+
`?t`	t^-
`?!t`	t^*
`d&`	d^+

```
?- p&, !a, ?!X, ?a.
```

Programmafragment 3.2 Voorbeeld van borddoelen.

Zoals eerder vermeld worden taken ook als doel op het bord geplaatst. Deze techniek wordt hier ingevoerd om de declaratieve semantiek later te vereenvoudigen. In de praktijk worden niet alle taken, maar slechts de geblokkeerde taken op het bord geplaatst (zie hoofdstuk 4). De gebeurtenissen d^- en d^* komen in deze tabel niet voor omdat zij niet corresponderen met een borddoel.

Doordat bordgebeurtenissen functies zijn kunnen ze aan de hand van functiecompositie samengesteld worden. Een dergelijke samenstelling wordt een *bordspoor* genoemd.

Definitie 21 (Bordspoor) *Een* bordspoor *is een opeenvolging van bordgebeurtenissen. Het lege spoor wordt voorgesteld door* λ*. De verzameling van bordsporen wordt* S_s *genoteerd.*

De opeenvolging van de gebeurtenissen g_1 en g_2 wordt genoteerd als $g_1.g_2$. Het bordspoor dat gegenereerd wordt door de vraag in programmafragment 3.2 is $p^+.a^+.X^*.a^-$. Verder in dit hoofdstuk wordt er ingegaan op de manier waarop dit bordspoor kan afgeleid worden.

Niet alle bordsporen kunnen uitgevoerd worden op een bord omdat ze bestaan uit partiële functies die niet overal gedefinieerd hoeven te zijn. Het spoor a^- kan bijvoorbeeld niet toegepast worden op het lege bord. Daarom wordt het begrip *geldig bordspoor* ingevoerd.

Definitie 22 (Geldig bordspoor) *Een bordspoor* $a_1.\cdots.a_n$ *is* geldig *indien het spoor ofwel het lege spoor* λ *is, ofwel indien de functiecompositie* $a_n \circ \cdots \circ a_1$ *gedefinieerd is op het lege bord.*

Geldige bordsporen zullen een belangrijke rol spelen in de semantiek van MULTI-PROLOG omdat ze te maken hebben met niet-blokkerende berekeningen. Een berekening die termineert zal dus steeds een geldig bordspoor genereren.

[6] Voor de eenvoud van de semantiek zal het effect van de knip niet bestudeerd worden.

```
p :- r(X), !X.
q :- ?X, r(X).

r(a).
r(b).
```

Programmafragment 3.3 Voorbeeld van een communicatieprogramma.

Tabel 3.2 Sporen voor de borddoelen uit het voorbeeldprogramma.

Doel	Spoor
`?- p.`	$\{a^+, b^+\}$
`?- q.`	$\{a^-, b^-\}$
`?- p,q.`	$\{a^+.a^-, a^+.b^-, b^+.a^-, b^+.b^-\}$
`?- q,p.`	$\{a^-.a^+, a^-.b^+, b^-.a^+, b^-.b^+\}$
`?- r(a).`	$\{\lambda\}$
`?- r(b).`	$\{\lambda\}$
`?- r(X).`	$\{\lambda\}$

Het spoor $p^+.a^+.X^*.a^-$ is geldig gezien de functie $a^- \circ X^* \circ a^+ \circ p^+$ gedefinieerd is op het lege bord $[\,]$. Het bewijs volgt uit de toepassing van definitie 20:

$$\begin{aligned}(a^- \circ X^* \circ a^+ \circ p^+)[\,] &= a^-(X^*(a^+(p^+[\,])))\\ &= a^-(X^*(a^+([p])))\\ &= a^-(X^*([p,a]))\\ &= a^-([p,a])\\ &= [p]\end{aligned}$$

Notatie *Gegeven twee sporen s_1 en s_2, dan wordt de concatenatie van s_1 en s_2 genoteerd als $s_1 \oplus s_2$, en de vermenging*[7] *als $s_1 \otimes s_2$. Merk op dat de vermenging van twee sporen aanleiding geeft tot een verzameling van sporen.*

De concepten uit deze sectie kunnen gebruikt worden om het gedrag van programmafragment 3.3 uit de inleiding nauwkeuriger te beschrijven. Zonder echt op de details in te gaan is het eenvoudig om in te zien dat de doelen uit programmafragment 3.3 de verzameling van sporen in tabel 3.2 zullen opleveren. Als illustratie bewijzen we dat $a^+.a^-$ een geldig spoor is.

$$\begin{aligned}(a^- \circ a^+)([\,]) &= a^-(a^+([\,]))\\ &= a^-([a])\end{aligned}$$

[7] De vermenging van twee sporen s_1 en s_2 is een permutatie van de gebeurtenissen van $s_1 + s_2$ zodanig dat de volgorde van de gebeurtenissen van s_1 en s_2 gerespecteerd wordt.

$$= \quad [\,].$$

Analoog kan bewezen worden dat $b^-.a^+$ een niet-geldig spoor is.

$$a^+ \circ b^-([\,]) \quad = \quad a^+(b^-([\,])).$$

Gezien b^- niet gedefinieerd is op het lege bord, is het totale spoor $b^-.a^+$ ongeldig.

3.5 Operationele Semantiek

De operationele semantiek wordt uitgedrukt aan de hand van een transitiesysteem. Een transitieregel legt een verband tussen elementen van de transitierelatie. Hij heeft de volgende syntaxis:

$$\frac{\text{Onderstelling}}{\text{Gevolg}} \text{ indien Voorwaarde}$$

De onderstelling en de voorwaarde kunnen eventueel achterwege gelaten worden. In dat geval zal het gevolg onvoorwaardelijk van toepassing zijn. In het algemeen geldt dat indien voldaan is aan de onderstelling en de voorwaarde het gevolg van toepassing is.

Een *transitierelatie* beschrijft de mogelijke overgangen tussen *configuraties*. Een configuratie is een drietal bestaande uit

(i) een lijst van termen $b \in S_b$ die de huidige inhoud van het bord weergeeft;

(ii) een multiset v van taken die op een bepaald ogenblik aan het uitvoeren zijn;

(iii) een substitutie $\theta \in S_\theta$ die de waarden van de variabelen weergeeft.

Merk op dat in een configuratie een onderscheid gemaakt wordt tussen de termen en de taken op het bord. De taken worden in een multiset bijgehouden terwijl de termen in een lijst opgeslagen worden. Er wordt hier dus geen gebruik gemaakt van de mogelijkheid om de taken ook op het bord op te slaan. Deze beschrijving staat dichter tegen de werkelijkheid omdat in tegenstelling tot de termen op het bord de taken in de praktijk niet geordend zijn.

Notatie Voor het gemak van de representatie wordt met de notatie $v[e]$ een multiset v bedoeld die het element e bevat.

Tot nog toe is er nog maar weinig gezegd over de taken die in een configuratie voorkomen. MULTI-PROLOG maakt een onderscheid tussen voorgrond- en achtergrondtaken. Er is slechts één voorgrondtaak, maar er kunnen verscheidene achtergrondtaken zijn. De voorgrondtaak is verbonden met de vraag waarmee het programma opgestart wordt. Tijdens de uitvoering van het programma kunnen er enkel achtergrondtaken gecreëerd worden.

Definitie 23 (Taak) *Voorgrond- en achtergrondtaken worden respectievelijk genoteerd als* $\leftarrow d$ *en* $\hookleftarrow d$, *met* $d \in S_d$. *De notatie* $\Leftarrow d$ *wordt gebruikt om een taak te noteren, zonder onderscheid te maken tussen voorgrond- en achtergrondtaken. De verzameling van taken wordt* S_T *genoteerd.*

De taak die geassocieerd wordt met de initële vraag `?- d.` wordt genoteerd als $\leftarrow d$.

Om de notatie te verlichten wordt de verzameling van alle multisets van elementen uit S_T de verzameling S_V genoemd. Een configuratie is dus een element van $S_b \times S_V \times S_\theta$.

Geregeld is het nodig alle vrije variabelen uit een uitdrukking te vervangen door nieuwe variabelen. De aldus nieuw geconstrueerde uitdrukking wordt een *duplicaat* genoemd.

De transitierelatie beschrijft de mogelijke overgangen tussen configuraties en wordt nu formeel als volgt gedefinieerd.

Definitie 24 (Transitierelatie) *Gegeven een programma* $P \in S_P$, *wordt de transitierelatie* $\rightarrow$ *gedefinieerd als de kleinste relatie over* $S_b \times S_V \times S_\theta$ *die aan de volgende voorwaarden voldoet. Voor het gemak van de representatie wordt het koppel* $((b_1, v_1, \theta_1), (b_2, v_2, \theta_2))$ *voorgesteld als* $< b_1, v_1, \theta_1 > \rightarrow < b_2, v_2, \theta_2 >$.

Atoomreductie

$$< b, v[\Leftarrow a], \theta > \rightarrow < b, v[\Leftarrow r], \theta\sigma >$$

$$\textit{indien} \left\{ \begin{array}{l} (k : -r) \in P \\ k \text{ en } a\theta \text{ unificeren en } \sigma = mgu(k, a\theta) \end{array} \right\}$$

Taakcreatiereductie

$$< b, v[\Leftarrow d\&], \theta > \rightarrow < b, v[\Leftarrow \delta] + [\hookleftarrow q], \theta >$$

$$\textit{indien } \{q \text{ een duplicaat is van } d\theta\}$$

Schrijfreductie

$$< b, v[\Leftarrow !t], \theta > \rightarrow < b + [u], v[\Leftarrow \delta], \theta >$$

$$\textit{indien } \{u \text{ een duplicaat is van } t\theta\}$$

Destructieve Leesreductie

$$< b, v[\Leftarrow ?t], \theta > \rightarrow < b', v[\Leftarrow \delta], \theta\sigma >$$

$$\textit{indien} \left\{ \begin{array}{l} \exists u \in b : u \text{ unificeert met } t\theta \\ w \text{ is de eerste dergelijke term } u \text{ in } b \\ \sigma = mgu(w, t\theta) \\ b' \text{ is } b \text{ waaruit } w \text{ verwijderd werd} \end{array} \right\}$$

Niet-destructieve Leesreductie

$$< b, v[\Leftarrow ?!t], \theta > \rightarrow < b, v[\Leftarrow \delta], \theta\sigma >$$

$$indien \left\{ \begin{array}{l} \exists u \in b : u \text{ unificeert met } t\theta \\ w \text{ is de eerste dergelijke term } u \text{ in } b \\ w' \text{ is een duplicaat van } w \\ \sigma = mgu(w', t\theta) \end{array} \right\}$$

Compositiereductie

$$< b, v[\Leftarrow (\delta, d)], \theta > \rightarrow < b, v[\Leftarrow d], \theta >$$

$$\frac{< b, v[\Leftarrow d_1], \theta > \rightarrow < b', v'[\Leftarrow d_1'], \theta' >}{< b, v[\Leftarrow (d_1, d_2)], \theta > \rightarrow < b', v'[\Leftarrow (d_1', d_2)], \theta' >}$$

De zonet gedefinieerde zes reducties beschrijven de transitierelatie voor Multi-Prolog. De eerste reductie beschrijft de klassieke atoomreductie. De unificatie van een doel met de kop van een bepaling geeft aanleiding tot een substitutie σ die gecombineerd wordt met de reeds bestaande substitutie θ.

De taakcreatiereductie, de schrijfreductie en de niet-destructieve-leesreductie maken allemaal gebruik van duplicaten om te verhinderen dat er via het bord gemeenschappelijke variabelen zouden ontstaan.

De leesreducties gaan steeds op zoek naar de eerste (en dus ook oudste) term uit het bord die unificeert met het argument van de leesbewerking. Indien er geen dergelijke term gevonden wordt, vindt de leesreductie niet plaats. Wordt er wel een term gevonden, dan wordt de geselecteerde term van het bord verwijderd indien het een destructieve leesreductie betreft. De reducties voor de compositie bevatten geen verrassingen en leggen de selectieregel vast doordat ze steeds het meest linkse doel eerst evalueren. Deze selectieregel is dezelfde als deze van PROLOG.

Het resultaat van de uitvoering van een logisch programma wordt uitgedrukt aan de hand van de substituties die deze uitvoering oplevert indien de berekening met succes beëindigd wordt. Een berekening is met succes beëindigd indien de multiset van taken met succes beëindigd is.

Definitie 25 (Succesvolle beëindiging van een multiset van taken)
Een multiset van taken is met succes beëindigd indien de (enige) voorgrondtaak in deze multiset tot $\leftarrow \delta$ gereduceerd is.

Deze definitie vereist niet dat de achtergrondtaken ook getermineerd zijn. Dit is echter geen probleem omdat wij enkel geïnteresseerd zijn in de substituties die door de voorgrondtaak gegenereerd worden. Van zodra de voorgrondtaak termineert kunnen de substituties niet langer door de achtergrondtaken

beïnvloed worden gezien er geen variabelen gemeenschappelijk zijn. De taken in de achtergrond mogen derhalve zelfs niet-terminerend zijn.

Alle multisets van taken die niet met succes termineren worden gecatalogeerd als falende berekeningen. De oorzaak van dit falen kan drieërlei zijn:

(i) ofwel termineert de berekening niet, en blijft steeds maar verder reduceren (niet-termineren in de klassieke zin [Llo87]);

(ii) ofwel termineert de berekening niet, maar ditmaal doordat er informatie op het bord ontbreekt (blokkerende taak);

(iii) ofwel ontbreken er geschikte bepalingen (echte faling).

Definitie 25 wordt nu gebruikt om de *derivatierelatie* te definëren.

Definitie 26 (Derivatierelatie) *Definieer* $\vdash$ *als de volgende relatie: voor elk programma* $P \in S_P$, $d \in S_d$, *en* $\theta \in S_\theta$ *geldt dat* $P \vdash d[\theta]$ *indien er borden* $b_1, \ldots, b_m \in S_b$, *multisets van taken* $v_1, \ldots, v_m \in S_V$, *en substituties* $\theta_1, \ldots, \theta_{m-1} \in S_\theta$ *bestaan zodanig dat* v_m *getermineerd is, en* v_i *niet,* $1 \leq i < m$, *en dat er een transitie* $< [\,], \{\leftarrow d\}, \epsilon > \rightarrow < b_1, v_1, \theta_1 > \rightarrow \cdots \rightarrow < b_m, v_m, \theta >$ *bestaat.*

Definitie 27 (Operationele Semantiek) *De operationele semantiek wordt gedefinieerd als een functie* O: $S_P \times S_d \rightarrow \mathcal{P}(S_\theta)$. *Gegeven een programma* $P \in S_P$ *en een doel* $d \in S_d$ *geldt dat* $O(P, d) = \{\theta_{|d} : P \vdash d[\theta]\}$.

De notatie $\theta_{|d}$ staat voor de restrictie van de substitutie θ tot de variabelen van d. De operationele semantiek $O(P, d)$ is dus de verzameling van alle substituties die het resultaat zijn van de berekening van d.

Dit is een analoge beschrijving als diegene die gebruikt wordt in de operationele semantiek van klassieke Hornbepalingen. Hieruit blijkt dat de semantiek van de initiële vraag in MULTI-PROLOG niet verschillend is van die van PROLOG.

3.6 Declaratieve Semantiek

Daar waar de operationele semantiek een formalisering is van de evaluatie van een logische uitdrukking, is het in de declaratieve semantiek te doen om de waarheid van een uitdrukking. In PROLOG is de waarheid tijdloos. Dit is echter niet langer het geval in MULTI-PROLOG, gezien het al dan niet waar zijn van een uitdrukking nu afhankelijk kan zijn van de inhoud van het bord en deze inhoud het resultaat is van eerder gebeurde bordbewerkingen.

Daarom zal er bij het opstellen van de declaratieve semantiek expliciet rekening moeten gehouden worden met de neveneffecten van de borddoelen. Deze neveneffecten introduceren een tijdsnotie in MULTI-PROLOG. In de nu volgende declaratieve semantiek zal deze notie van tijd geformaliseerd worden

door bordsporen. Doordat de bordgebeurtenissen in een bordspoor chronologisch geordend zijn, leggen zij meteen ook de selectie- en zoekregel bij de resolutie vast[8].

Voor het opstellen van de declaratieve semantiek werd inspiratie gevonden in de semantiek van de *gespreide logica* van L. Monteiro [Mon84, Mon86]. Bij het opstellen van de declaratieve semantiek werd er getracht om zo dicht mogelijk bij de klassieke beschrijvingen te blijven. Net zoals voor de taal MULTI-PROLOG hebben we getracht om ons te beperken tot de minimale uitbreidingen van deze semantiek.

Bij de studie van de semantiek van Multi-Prologtaken moet er een onderscheid gemaakt worden tussen de voorgrondtaak en de achtergrondtaken. De voorgrondtaak gedraagt zich min of meer als een gewone Prologtaak. Het feit dat de voorgrondtaak tijdens zijn uitvoering mogelijk via het bord communiceert met andere taken is uiterlijk niet merkbaar.

De semantiek van de achtergrondtaken is daarentegen verre van traditioneel omdat het gedrag van deze taken slechts belangrijk is met betrekking tot de bordcommunicatie. Of een dergelijke taak nu slaagt, al dan niet termineert of faalt is niet echt belangrijk. Het enige wat telt is de communicatie met het bord (het externe gedrag). Een achtergrondtaak is een soort van slaaf die sommige van de vragen die door andere taken op het bord geplaatst worden beantwoordt. Het gedrag van een achtergrondtaak kan dan ook volledig beschreven worden aan de hand van de verzameling van bordsporen die ze kan genereren.

De eerste opdracht is het vinden van een geschikte uitdrukking voor een interpretatie. Een interpretatie zal nu naast de klassieke ingredïenten ook een interpretatie voor de bordcommunicatieprimitieven moeten bevatten. Een atoom zal waar zijn indien het een logisch gevolg is van een programma in de klassieke betekenis en als bovendien ook zijn communicatie met het bord slaagt. Daarom zullen atomen voortaan niet langer onvoorwaardelijk waar of vals zijn, maar zal hun al dan niet waar zijn afhangen van de bordinhoud. Deze afhankelijkheid wordt gemodelleerd aan de hand van bordsporen. Een atoom a zal waar zijn indien zijn geassocieerd spoor s met succes op het bord kan uitgevoerd worden. Een interpretatie voor een voorgrondtaak bestaat dan uit een verzameling van koppels bestaande uit een spoor en een atoom. Een atoom kan waar zijn voor verscheidene sporen en verschillende atomen kunnen waar zijn voor hetzelfde spoor.

Deze interpretatie volstaat om de voorgrondtaak te bestuderen, maar is niet bruikbaar voor de achtergrondtaken. Hier manifesteert zich de moeilijkheid dat achtergrondtaken niet-terminerend kunnen zijn en toch kunnen bijdragen

[8] Doordat MULTI-PROLOG een parallelle uitbreiding van PROLOG is, worden zowel de selectie- als zoekregel van PROLOG overgenomen.

tot de totale berekening. Wij dus moeten uitkijken naar een manier om niet-terminerende taken toch declaratief te beschrijven. Wij stellen voor om de interpretatie voor een achtergrondtaak te laten bestaan uit een verzameling van drietallen bestaande uit één spoor en twee doelen. Het spoor moet kunnen uitgevoerd worden om een geldige overgang van het eerste naar het tweede doel te kunnen maken.

Een veralgemeende interpretatie[9] zal hier dus bestaan uit twee verzamelingen. De eerste verzameling heeft te maken met de voorgrondtaak, de tweede verzameling heeft te maken met de achtergrondtaken. Vooraleer een meer formele definitie te geven, moeten we eerst een concretisatiefunctie definiëren.

Een logische uitdrukking waarin geen vrije variabelen voorkomen wordt *volledig geconcretiseerd* genoemd. In wat volgt zal een aantal van de eerder gedefinieerde verzamelingen beperkt moeten worden tot die elementen die volledig geconcretiseerd zijn. Om dit op een handige manier te kunnen uitdrukken wordt er een concretisatiefunctie γ gedefinieerd.

Definitie 28 (Concretisatiefunctie) *Gegeven een verzameling V van logische uitdrukkingen wordt de concretisatiefunctie γ gedefinieerd als*

$$\gamma(V) = \{e \in V : e \text{ is volledig geconcretiseerd}\}.$$

Definitie 29 (Veralgemeende interpretatie) *Een veralgemeende interpretatie is een element van $\mathcal{P}(\gamma(S_s) \times \gamma(S_a)) \times \mathcal{P}(\gamma(S_s) \times \gamma(S_d) \times \gamma(S_d))$.*

De verzameling van alle veralgemeende interpretaties wordt S_I genoteerd. Deze verzameling wordt omgezet in een complete tralie door de volgende complete partiële ordening over S_I te definiëren.

Definitie 30 (Complete partiële ordening over S_I) *Definieer $\leq$ als de volgende relatie over S_I: voor elke twee elementen (I_v, I_a) en $(J_v, J_a) \in S_I$ geldt dat $(I_v, I_a) \leq (J_v, J_a)$ als en slechts als $I_v \subseteq J_v$ en $I_a \subseteq J_a$.*

3.6.1 Modeltheorie

Modeltheorie houdt zich bezig met het model van een programma. Net zoals voor de klassieke semantiek voor Hornbepalingen zal ook hier een model ten minste moeten bestaan uit alle logische gevolgen van een programma. Het verschil met een klassiek model is nu wel dat een atoom niet langer onvoorwaardelijk waar of vals is, maar dat dit nu zal afhangen van een bordspoor. Enkel indien er aan de voorwaarden voldaan is om het bordspoor te kunnen uitvoeren, zal het atoom waar zijn.

[9] De hier beschreven interpretatie wordt veralgemeend genoemd om het onderscheid met een klassieke interpretatie voor Hornbepalingen te kunnen maken.

Definitie 31 (Het begrip waarheid) *Gegeven een spoor s, een interpretatie $I = (I_v, I_a)$, en een uitdrukking u, wordt het feit dat u waar is met betrekking tot s en I genoteerd als $s \models_I u$, en gedefinieerd aan de hand van de volgende gevallen:*

Uitdrukking

$s \models_I u$ *indien* $s \models_I u_0$, *voor gelijk welke volledig geconcretiseerde instantie* u_0 *van* u

Volledig Geconcretiseerd Atoom

$s \models_I a$ *indien* $(s, a) \in I_v$

Volledig Geconcretiseerde Bepaling

$s \models_I (k : -r)$ *indien* $s \models_I k$ *telkens wanneer* $s \models_I r$

Volledig Geconcretiseerd Doel

$s \models_I [\leftarrow d]$ *indien* s *een geldig spoor is, en indien er* $s', u_1, \ldots, u_n \in S_s, \quad p_1, \ldots, p_n, q_1, \ldots, q_n \in \gamma(S_d)$ *zodat*

(i) $s' \models_I d$

(ii) $(u_i, p_i, q_i) \in I_a, \quad i = 1, \ldots, n$

(iii) $s \in s' \otimes (p_1^-.u_1.q_1^+) \otimes \cdots \otimes (p_n^-.u_n.q_n^+)$

Volledig Geconcretiseerde Taakcreatie

$d^+ \models_I d\&$ *indien* $d \in \gamma(S_d)$

Volledig Geconcretiseerde Schrijfbewerking

$t^+ \models_I !t$ *indien* $t \in \gamma(S_t)$

Volledig Geconcretiseerde Destructieve Leesbewerking

$t^- \models_I ?t$ *indien* $t \in \gamma(S_t)$

Volledig Geconcretiseerde Niet-destructieve Leesbewerking

$t^* \models_I ?!t$ *indien* $t \in \gamma(S_t)$

Volledig Geconcretiseerde Compositie

$s \models_I (d_1, d_2)$ *indien er* $s_1, s_2, u_1, \ldots, u_n \in S_s$ *en* $p_1, \ldots, p_n, q_1, \ldots, q_n \in S_d$ *bestaan zodat*

(i) $s_1 \models_I d_1, \quad s_2 \models_I d_2$

(ii) $(u_i, p_i, q_i) \in I_a, \quad i = 1, \ldots, n$

(iii) $s \in (s_1 \oplus s_2) \otimes (p_1^-.u_1.q_1^+) \otimes \cdots \otimes (p_n^-.u_n.q_n^+)$

Een uitdrukking u is waar met betrekking tot een interpretatie I indien er een spoor s kan gevonden worden zodat $s \models_I u$. Dit wordt dan genoteerd als $\models_I u$. Een verzameling van uitdrukkingen V is waar met betrekking tot een interpretatie I indien er voor elk van de elementen e van de verzameling V geldt dat $\models_I e$.

Het gedeelte van de interpretatie dat toegeschreven wordt aan de achtergrondtaken moet een minimale hoeveelheid informatie bevatten. Zij wordt formeel gedefinieerd door de relatie Ttp.

Definitie 32 *De relatie Ttp is de kleinste relatie over $\gamma(S_s) \times \gamma(S_d) \times \gamma(S_d)$ die voldoet aan de volgende voorwaarden.*

Leeg spoor
$(\lambda, d, d) \in Ttp$

Transitieve Sluiting
$(s, d_1, d_3) \in Ttp$ *indien er* $s_1, s_2 \in S_s$, *en* d_2 *in* $\gamma(S_d)$ *bestaan zodanig dat* $(s_1, d_1, d_2) \in Ttp$, $(s_2, d_2, d_3) \in Ttp$ *en* $s = s_1 \oplus s_2$.

Volledig Geconcretiseerd Atoom
$(s, a, d) \in Ttp$ *indien er een volledig geconcretiseerde instantie* $(a : -r)$ *van een bepaling uit het programma* P *bestaat zodat* $(s, r, d) \in Ttp$.

Volledig geconcretiseerde Taakcreatie
$(d^+, d\&, \delta) \in Ttp$ *indien* $d \in \gamma(S_d)$

Volledig Geconcretiseerde Schrijfbewerking
$(t^+, !t, \delta) \in Ttp$ *indien* $t \in \gamma(S_t)$

Volledig Geconcretiseerde Destructieve Leesbewerking
$(t^-, ?t, \delta) \in Ttp$ *indien* $t \in \gamma(S_t)$

Volledig Geconcretiseerde Niet-Destructieve Leesbewerking
$(t^*, ?!t, \delta) \in Ttp$ *indien* $t \in \gamma(S_t)$

Volledig Geconcretiseerde Compositie

(i) $(s, (d_1, d_2), (d_1', d_2)) \in Ttp$ *indien* $(s, d_1, d_1') \in Ttp$ *en* $d_2 \in \gamma(S_d)$,

(ii) $(s, (d_1, d_2), d_2') \in Ttp$ *indien er* $s_1, s_2 \in S_s$ *bestaan zodat* $(s_1, d_1, \delta) \in Ttp$, *en* $(s_2, d_2, d_2') \in Ttp$, *en bovendien* $s = s_1 \oplus s_2$.

Definitie 33 (Veralgemeend model) *Een veralgemeend model voor een verzameling van uitdrukkingen V is een veralgemeende interpretatie $I = (\mathcal{I}, I_a)$ zodanig dat $Ttp \subseteq I_a$ en $\models_I V$. Een uitdrukking U is een logisch gevolg van een verzameling van uitdrukkingen V als en slechts als elk model voor V ook een model voor U is. Dit wordt vervolgens genoteerd als $V \models U$.*

Naar analogie met de modellen voor Hornbepalingen zijn ook hier de doorsnede en de unie van twee modellen een model. Onder doorsnede en unie moeten dan wel de componentsgewijze doorsnede en unie verstaan worden. Doordat $(S_I, \leq)$ een complete tralie is, zal de doorsnede van alle modellen bestaan.

Definitie 34 (Minimaal model) *Gegeven een programma P, wordt het* minimale model *gedefinieerd als de doorsnede van alle modellen voor P.*

Definitie 35 (Modeltheoretische semantiek) *De declaratieve semantiek gebaseerd op de modeltheorie wordt gedefinieerd als een functie $D_m : S_P \times S_d \to \mathcal{P}(S_\theta)$. Gegeven een programma $P \in S_P$ en een doel $d \in S_d$ geldt dat*

$$D_m(P, d) = \{\theta_{|d} : P \models d\theta\}.$$

3.6.2 Fixpunttheorie

De definities uit de voorgaande sectie leren ons iets over het bestaan van een declaratieve semantiek. Zij bieden ons echter geen bruikbaar algoritme om de semantiek te berekenen. In deze sectie wordt een alternatieve, meer algoritmische formulering van de declaratieve semantiek gegeven, hierbij gebruik makend van de fixpunttheorie.

De modellen van een verzameling van uitdrukkingen V kunnen gekarakteriseerd worden als de prefixpunten van de continue operator $T_P : S_I \to S_I$ die de 'gevolgenoperator' genoemd wordt en als volgt gedefinieerd wordt.

Definitie 36 (De gevolgenoperator) *Gegeven een programma $P \in S_P$ en S de verzameling van rompdoelen van de volledig geconcretiseerde instanties van de bepalingen van P. Definieer $T_P : S_I \to S_I$ als de volgende functie: $T_P(I_v, I_a) = (J_v, J_a)$ met*

$$\begin{aligned}
J_v \quad = \quad & \{(s, a) \ : \ (a : -r) \text{ is een volledig geconcretiseerde instantie} \\
& \qquad \text{van een bepaling in } P \text{ en } s \models_I r\} \\
& \cup \{(t^+, !t) \ : !t \in S\} \\
& \cup \{(t^-, ?t) \ : ?t \in S\} \\
& \cup \{(t^*, ?!t) \ : ?!t \in S\} \\
& \cup \{(d^+, d\&) \ : \ d\& \in S\}
\end{aligned}$$

$$\begin{aligned}
J_a \quad = \quad & \{(t^+, !t, \delta) \ : !t \in S\} \\
& \cup \{(t^-, ?t, \delta) \ : ?t \in S\} \\
& \cup \{(t^*, ?!t, \delta) \ : ?!t \in S\} \\
& \cup \{(d^+, d\&, \delta) \ : \ d\& \in S\} \\
& \cup \{(s, a, d) \ : \ (a : -r) \text{ is een volledig geconcretiseerde instantie} \\
& \qquad \text{van een bepaling in } P \text{ en } (s, r, d) \in I_a\} \\
& \cup \{(s, (d_1, d_2), (d_1', d_2)) \ : \ (s, d_1, d_1') \in I_a \text{ en } d_2 \in \gamma(S_d)\} \\
& \cup \{(s, (d_1, d_2), d_2') : (s_1, d_1, \delta) \in I_a, \ (s_2, d_2, d_2') \in I_a \text{ en } s = s_1 \oplus s_2\}
\end{aligned}$$

De nu volgende vier proposities vormen het sluitstuk van deze semantische beschrijving. Zij worden hier evenwel zonder bewijs gegeven omdat het bewijs ervan ons inziens buiten het bestek van dit proefschrift valt.

De twee eerste proposities betreffen de equivalentie tussen de modeltheoretische formulering en de fixpuntformulering van de declaratieve semantiek.

Propositie 1

(i) De operator T_p is continu.
(ii) Elke veralgemeende interpretatie I is een veralgemeend model voor het programma P als en slechts als $T_p(I) \subseteq I$.

Propositie 2 *Voor elk programma P geldt dat $M_P = lfp(T_P) = T_P \uparrow \omega$.*

De declaratieve semantiek gebaseerd op de fixpuntsemantiek wordt dan als volgt gedefinieerd.

Definitie 37 (Declaratieve semantiek) *De declaratieve semantiek gebaseerd op de fixpuntsemantiek wordt gedefinieerd als een functie $D_f : S_P \times S_d \rightarrow \mathcal{P}(S_\theta)$. Gegeven een programma $P \in S_P$ en een doel $d \in S_d$ geldt dat $D_f(P,d) = \{\theta_{|d} : \exists$ een geldig spoor $s \in S_s$ zodat $s \models_{lfp(T_p)} d\theta\}$.*

De nu volgende propositie drukt uit dat de modeltheoretische semantiek en de fixpuntsemantiek equivalent zijn.

Propositie 3 *Voor elk programma $P \in S_P$, $d \in S_d$ en $\theta \in S_\theta$ geldt dat $P \models d\theta$ als en slechts als er een geldig spoor $s \in S_s$ bestaat zodat $s \models_{lfp(T_P)} d\theta$. Bovendien geldt dat $D_m(P,d) = D_f(P,d)$.*

De nu volgende propositie drukt uit dat de operationele semantiek correct en compleet is. Dit wil zeggen dat elke berekende oplossing ook een echte oplossing moet zijn, en dat elke oplossing ook moet kunnen berekend worden.

Propositie 4 *Voor elk programma $P \in S_P$, $d \in S_d$ en $\theta \in S_\theta$*

(i) indien $P \vdash d[\theta]$ geldt dat voor elke $\sigma \in S_\theta$, $P \models d\theta\sigma$;
(ii) indien $P \models d\theta$ geldt dat er een $\sigma \in S_\theta$ bestaat zodanig dat $d\sigma \geq d\theta$ en $P \vdash d[\sigma]$.

3.7 Veralgemeningen

In deze sectie wordt bekeken op welke manier de hier beschreven semantiek kan uitgebreid worden om ook de andere Multi-Prologpredicaten te omvatten. Om de semantiek zo eenvoudig mogelijk te houden is het belangrijk om de communicatieprimitieven zoveel mogelijk in aantal te beperken.

Uit de bespreking van de voorwaardelijke communicatieprimitieven (blz. 35 en verder) blijkt dat deze primitieven bijna steeds het bord voor een langere tijd moeten vergrendelen dan met de primitieven uit de kerntaal mogelijk is. Indien er een primitief zou bestaan om het bord expliciet te vergrendelen (lockbb)

```
blokkerend_lezen(X) :- herhaal, niet_blokkerend_lezen(X).

herhaal.
herhaal :- herhaal.
```

Programmafragment 3.4 Blokkerende leesbewerking.

```
niet_terugzoekend_lezen(X) :- terugzoekend_lezen(X), !.
```

Programmafragment 3.5 Niet-terugzoekende leesbewerking.

en nadien terug vrij te geven (`unlockbb`)[10], dan zouden de meeste uitbreidingen gewoon gesimuleerd kunnen worden. Het steeds weer vergrendelen van het totale bord is niet zeer efficiënt, maar voor een theoretische studie is de efficiëntie van geen belang omdat alle bewerkingen ogenblikkelijk gebeuren en er geen notie van uitvoeringstijd bestaat.

Als we de twee vergrendelingsprimitieven invoeren dan kunnen alle hiervoor besproken leesbewerkingen herleid worden tot een tweetal bewerkingen met name

(i) het niet-blokkerend, terugzoekend, destructief lezen van een term,

(ii) het niet-blokkerend, terugzoekend, niet-destructief lezen van een term.

De blokkerende varianten worden dan verkregen door een herhaal-faallus zoals geschetst in programmafragment 3.4. De niet-blokkerende leesbewerking wordt herhaald uitgevoerd totdat zij slaagt, d.w.z. totdat er een term kan gelezen worden. In de praktijk zou het natuurlijk onverstandig zijn om een blokkerende leesbewerking op deze manier te implementeren, maar in de theorie is dit geen bezwaar.

De niet-terugzoekende varianten kan men realiseren met behulp van een knip zoals geschetst in programmafragment 3.5. Van zodra een term gelezen werd, wordt hierdoor elke mogelijkheid tot het lezen van een alternatieve term ongedaan gemaakt.

De voorwaardelijke schrijfbewerking kan geïmplementeerd worden aan de hand van programmafragment 3.6. Doordat tussen de evaluatie van de voorwaarde `V` en de uitvoering van de schrijfbewerking het bord gegarandeerd stabiel blijft, is dit een correct werkende implementatie.

Analoog kan de voorwaardelijke leesbewerking geïmplementeerd worden aan de hand van programmafragment 3.7. Door de bordvergrendeling kan er

[10] De predicaten `lockbb` en `unlockbb` hebben het averechtse effect bij het terugzoeken. Het predicaat `lockbb` zal dan het bord vrijgeven en `unlockbb` zal het bord vergrendelen. De kritische sectie die afgebakend wordt door beide predicaten kan dus zowel voorwaarts als achterwaarts doorlopen worden.

```
X!V :- lockbb, V, !X, unlockbb.
```

Programmafragment 3.6 Voorwaardelijke schrijfbewerking met bordvergrendeling.

```
X?V :- lockbb, ?!*X, V, !, ?X, unlockbb.
```

Programmafragment 3.7 Voorwaardelijke leesbewerking met bordvergrendeling.

```
Bord!X :- !bord(Bord,X).
Bord?X :- ?bord(Bord,X).
Bord??X :- ??bord(Bord,X).
Bord??*X :- ??*bord(Bord,X).
Bord?X:V :- ?bord(Bord,X):V.
...
```

Programmafragment 3.8 Simulatie van lokale borden.

nu wel een correcte synchronisatie gegarandeerd worden, en dit in tegenstelling met de situatie in programmafragment 2.17. Vergelijkbare implementaties kunnen geconstrueerd worden voor de zeven andere leesvarianten.

Meervoudige borden kunnen gesimuleerd worden door de termen in te pakken in een structuur die de naam van het bord bevat. Net zoals voor kanalen (zie blz. 44), kunnen de schrijf- en leesbewerkingen op een lokaal bord gesimuleerd worden door het programmafragment 3.8 te gebruiken.

Doordat elk Multi-Prologprogramma zodanig kan herschreven worden dat het enkel gebruik maakt van niet-blokkerende, terugzoekende leesbewerkingen en bordvergrendeling volstaat het in principe om een semantische studie te baseren op louter deze primitieven. Het zal wel geen toeval zijn dat de semantiek van de twee leesbewerkingen precies overeenstemmen met de semantiek van de predicaten `retract` en `clause` uit PROLOG.

Samenvatting

In dit hoofdstuk werd een aantal van de bordcommunicatieprimitieven aan een meer theoretisch onderzoek onderworpen. Zo werd onder meer hun effect op de semantiek van de onderliggende logische programmeertaal onderzocht. Als voornaamste resultaat kan vermeld worden dat de semantiek van een Multi-Prologvraag in feite in niets verschillend is van de semantiek van een Prologvraag. Hij kan nog steeds beschreven worden als een deelverzameling van de Herbrandbasis. Dit wijst erop dat de bordcommunicatie, ondanks het feit dat het een neveneffect betreft, niet drastisch interfereert met de onderliggende taal PROLOG.

Een operationele semantiek en een declaratieve semantiek werden opgesteld. Zij beschrijven zowel het effect van de voorgrond- als van de achtergrondtaken.

4 Implementatie

Het is pas dank zij de ondervinding
dat de wetenschap en de kunst vooruit gaan.

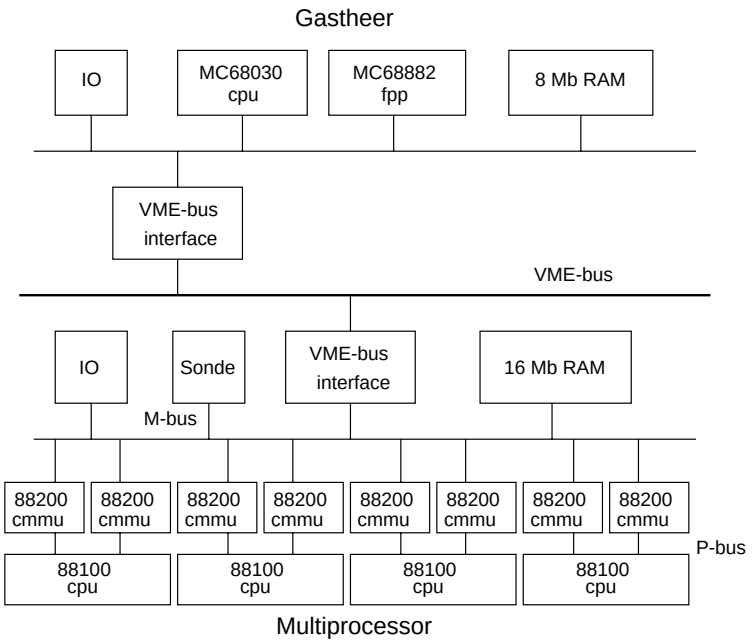
— ARISTOTELES, *Metafysica* (c. 325 v.C.)

Dit hoofdstuk geeft een beschrijving van de prototype-implementatie van de kerntaal van MULTI-PROLOG *die gerealiseerd werd op een multiprocessor met gemeenschappelijk geheugen. Na een summiere bespreking van de* WAM*-implementatie en de Multi-Prologtaken wordt er overgegaan naar een gedetailleerde bespreking van de implementatie van het bord, de meest kritische component. Er wordt besloten met een hoeveelheid prestatiegegevens. Sommige gegevens werden gemeten op het prototype, andere volgen uit een simulatiestudie van het bord.*

Met het oog op de validatie van het paradigma van het taakparallellisme en expliciete bordcommunicatie in PROLOG werd een implementatie van de taal MULTI-PROLOG gerealiseerd. De implementatie en het werkelijke gebruik vormt immers het onweerlegbare bewijs voor de degelijkheid van een paradigma. Door budgettaire beperkingen werd er een implementatie gemaakt voor een vierprocessormachine. Aan het einde van dit hoofdstuk wordt evenwel aangetoond dat het aantal processoren probleemloos kan verhoogd worden zonder een aanzienlijke aanpassing van de programmatuur.

4.1 Het Platform

Om zo snel mogelijk tot een werkend prototype te kunnen komen werd er geopteerd om zoveel mogelijk bestaande apparatuur en programmatuur te gebruiken bij de realisatie. Wij bespreken hierna de motivering voor elk van deze componenten, plus de voornaamste wijzigingen die moesten aangebracht worden. Een meer gedetailleerde discussie vindt men in [DBW91].



Afbeelding 4.1 Het hardwareplatform bestaande uit een gastheer en multiprocessor met gemeenschappelijk geheugen en vier RISC-processoren.

4.1.1 De Apparatuur

Als hardwareplatform werd er van meet af aan geopteerd voor een commercieel beschikbaar platform. Eind de tachtiger jaren begonnen de eerste RISC multiprocessoren met gemeenschappelijk geheugen de markt te betreden. De MVME188 van Motorola [Mot89b] behoorde toen tot de eerste generatie multiprocessorborden met RISC-processoren en coherente tussengeheugens. Er werd een systeem aangekocht, bestaande uit een multiprocessorkaart met vier 88100 RISC-processoren [Mot88a] en acht 88200 CMMUs [Mot88b] (16 KB tussengeheugen en geheugenbeheer) een UNIX gastheer met voldoende schijfcapaciteit en de nodige ontwikkelingshulpmiddelen (GREENHILL's cross C-compilator en Assembler) [OAS91a, OAS91b]. De multiprocessorkaart en de gastheer communiceren via de gemeenschappelijke VME-bus (zie afbeelding 4.1). De multiprocessor zelf bestond echter uit een naakte machine. Dit wil zeggen dat er geen toepassingen bijgeleverd werden behalve een primitieve 188BUG monitor [Mot89a] die een aantal elementaire ontluistaken op het machinetaalniveau en het geheugen mogelijk maakte.

In een later stadium van het project werd de VMETRO M-BUS-SONDE [Ind]

aangeschaft. Dit is een toestel waarmee de trafiek over de M-bus nauwkeurig geobserveerd kan worden. Deze informatie werd aangewend om het bordontwerp te optimaliseren.

4.1.2 MTOS-UX

De naakte multiprocessorhardware gaf niet veel ondersteuning bij de implementatie van MULTI-PROLOG. Voor de ontwikkeling van de sequentiële versie hadden wij geen nood aan veel meer dan een elementaire ontluizer op Assemblerniveau (de sequentiële vertolker werd immers omwille van de efficëntie in ASSEMBLER ontwikkeld), en een elementair communicatieprogramma tussen de gastheer en de multiprocessor.

Maar van zodra de parallelle versie moest ontwikkeld worden, was de nood aan meer ondersteuning voelbaar. Er werd toen uitgekeken naar een besturingssysteemkern voor de multiprocessor, om zelf niet te veel tijd te verliezen met de ontwikkeling van een werkverdeler, geheugenbeheer, enz... Een versie van UNIX voor de multiprocessor werd niet opportuun geacht omwille van de overlast die gepaard gaat met de taakcreatie en taakwisseling. Een lichtgewicht meerdradig uitvoeringspakket zoals μSYSTEM [BS90] of MACH bovenop UNIX had een oplossing kunnen bieden, maar deze leek ons onnodig complex in vergelijking met de kant en klare kernen die op dit soort van hardware aangeboden werden. We hadden trouwens maar nood aan een heel beperkt aantal functies, en de meeste kernen voldeden reeds ruimschoots aan deze behoefte.

Omdat we een efficiënte implementatie wilden realiseren werd er uitgekeken naar een ware-tijdskern. De Multi-Prologtoepassingen hoeven nu wel niet in ware tijd uitgevoerd te worden, maar ze zijn wel vaak reactief van aard (d.w.z. dat het programma reageert op externe stimuli; in ons geval gegevens die op het bord gezet worden). Er werd uiteindelijk geopteerd voor MTOS-UX [Rip89, Ind89], een ware-tijdskern die rond die tijd voor het eerst op een 88000-platform beschikbaar werd. Deze kern had drie voor ons belangrijke voordelen: (i) de programmatekst was beschikbaar, (ii) hij was beschikbaar op een groot aantal hardwareplatformen, en (iii) hij was voor ons betaalbaar.

Het installeren van MTOS-UX op de multiprocessor, en de volledige integratie ervan nam nog ongeveer een tweetal manmaand in beslag. De grootste tijd werd geïnvesteerd in de ontwikkeling van een protocol voor een procedureoproep van elders (RPC) [rpc88] tussen enerzijds MTOS-UX en anderzijds de gastheer. Dit vereiste een hoeveelheid programmeerwerk aan zowel de UNIX- als de MTOS-UX-kant. Eens dit gerealiseerd was, werd het wel mogelijk de multiprocessor te gebruiken vanaf eender welke terminal die toegang had tot het laboratoriumnetwerk. De fysische nabijheid van de multiprocessor was dan niet meer vereist, hetgeen het gebruik ervan fel vereenvoudigde.

4.1.3 Sicstus Prolog

De taal MULTI-PROLOG is een uitbreiding van PROLOG. Tijdens het ontwerp van de taal werd ervoor gezorgd dat de uitbreidingen conform de Prologsyntaxis waren. De bedoeling was om een bestaande Prologomgeving te kunnen gebruiken als compilator. Gezien SICSTUS-PROLOG als efficiënte en goedkope implementatie gemakkelijk te verkrijgen was, werd er besloten tot de aankoop ervan. SICSTUS PROLOG [AAF+ 91] werd ontwikkeld aan het Zweedse Instituut voor Computerwetenschappen (SICS) en bestaat uit een compilator en een vertolker voor PROLOG, samen met een WAM bytecode vertolker, een bibliotheek en een stuursysteem. Het produkt wordt geleverd als programmatekst, hetgeen het voor ons mogelijk maakte om alle nodige aanpassingen te doen.

De Sicstus-Prologomgeving wordt gebruikt om de syntaxis van de Multi-Prologprogramma's te controleren en als compilator (compilatie naar WAM-code). Er werd voor gezorgd dat, gegeven een vraag, alle informatie nodig voor de uitvoering op de multiprocessor (alle op te roepen doelen en de symbolentabel, ...) naar een bestand geschreven wordt. Dit bestand is in een formaat [DBW91] dat rechtstreeks, zonder extra manipulaties (behalve verplaatsing) op de multiprocessor door de SICSTUS WAM kan uitgevoerd worden. Het doorgronden van de interne structuur van SICSTUS PROLOG [Car87] was niet eenvoudig, maar loonde toch de moeite omdat het ons de moeite bespaarde om zelf een compilator te ontwikkelen. Het totaal van de aanpassingen nam ongeveer een drietal manmaand in beslag.

4.2 De WAM

Van meet af aan werd er gekozen voor een zogenaamde bytecodevertolker omdat deze een orde sneller kan gemaakt worden dan een programmatekstvertolker [DBW90]. Het ontwikkelen van een compilator die machinetaal of een programma in een hoge-niveautaal genereert zou een nog hogere snelheid toegelaten hebben, maar zou te veel tijd in beslag genomen hebben. Wij hadden gekozen om zo snel mogelijk tot een werkend prototype te komen. De sequentiële versie van de SICSTUS WAM werd ontwikkeld op een paar manmaand tijd. Het voorbeeld van de SICSTUS PROLOG was hierbij van groot nut. In tegenstelling met de C-implementatie van SICSTUS PROLOG, kozen wij echter voor een rechtstreekse implementatie in 88000-ASSEMBLER om de efficëntie te verhogen. Uit tabel 4.1 blijkt dat de versnelling niet te versmaden is[1], maar toch

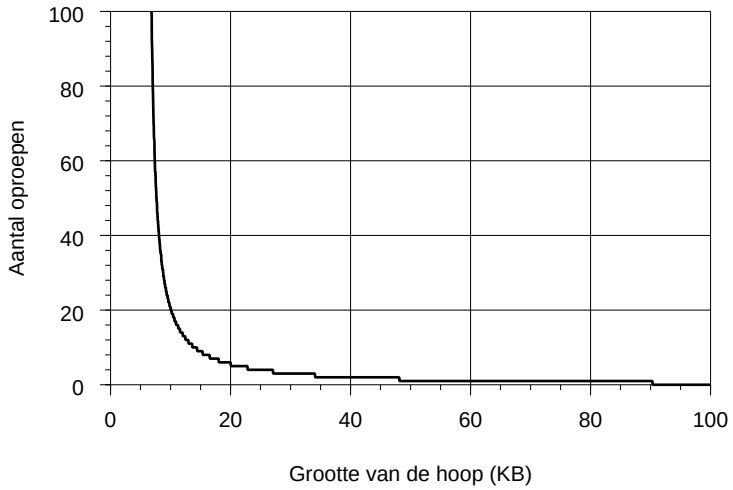
[1] Het is opmerkelijk dat de evaluatieprogramma `naive reverse 30` de kleinste versnelling oplevert. Dit is wellicht te wijten aan het feit dat dit programma in SICSTUS-PROLOG speciaal geoptimaliseerd werd omdat het vaak als maatgevend voor de snelheid van een implementatie beschouwd wordt. In onze implementatie werd `naive reverse` echter niet speciaal geoptimaliseerd omdat de snelheid van de sequentiële WAM voor ons onderzoek van secundair belang was.

Tabel 4.1 ASSEMBLER- tegenover C-efficiëntie. De machine die MULTI-PROLOG uitvoert is ongeveer viermaal sneller dan de machine die SICSTUS-PROLOG uitvoert. Hiermee werd rekening gehouden bij de berekening van de verhouding.

Evaluatieprogramma	MULTI-PROLOG (KLIPS)	SICSTUS PROLOG (KLIPS)	Verhouding M/S
`naive reverse 30`	118.50	22.29	1.33
`quicksort 50`	90.79	13.76	1.65
`hanoi 9`	88.52	15.68	1.41
`serialize`	67.44	9.19	1.83
`query`	100.67	17.24	1.46
`derive`	54.79	7.80	1.76

relatief beperkt blijft. Hieruit blijkt dus nogmaals dat het gebruik van de taal C het mogelijk maakt om een efficiëntie vergelijkbaar met die van handgecodeerde ASSEMBLER te bereiken. De optimalisaties die uitgevoerd worden in de hedendaagse compilators en de kracht van de hedendaagse processoren zal hieraan zeker niet vreemd zijn. Dit wordt trouwens bevestigd in [GDBD92], waar men zelfs zover gaat om de taal C te verkiezen boven ASSEMBLER om code in te genereren. C heeft het voordeel nagenoeg onbeperkt overdraagbaar te zijn naar alternatieve platformen.

De WAM-code vertolker heeft een geheugensaneringsroutine voor de hoop en de stapel. De programmatekst voor deze routine werd grotendeels overgenomen van SICSTUS PROLOG [ACHS88]. Om de prestatie van het gebruikte algoritme uit te testen wordt een variant van het programma `master mind` gebruikt. Dit programma berekent de optimale volgorde van vragen om een gegeven kleurencombinatie zo snel mogelijk te vinden. Dit programma loopt ongeveer gedurende 50 s en maakt intensief gebruik van de hoop omdat de gegevens bijgehouden worden in lijsten en omdat het programma recursief is. Het programma wordt uitgevoerd met een hoopgeheugen dat varieert tussen 6 KB en 100 KB. De uitvoering van het programma zonder geheugensanering heeft nood aan een hoopgeheugen van 90.3 KB. Uit afbeelding 4.2 blijkt dat in die buurt inderdaad voor het eerst de geheugensaneringsroutine opgeroepen wordt. De tweede oproep komt er pas bij een hoop van 48.1 KB. Minimaal is er nood aan een hoop van 6.149 KB. Per keer dat de saneringsroutine opgeroepen wordt, wordt er slechts 2 KB geheugen gerecupereerd. De gecumuleerde hoeveelheid gerecycleerd geheugen staat weergegeven in afbeelding 4.3. Deze grafiek vertoont sterke gelijkenis met afbeelding 4.2. Het gedrag van de geheugensaneringsroutine is echter enkel belangrijk met betrekking tot de uitvoeringstijd van het programma. De totale uitvoeringstijd in functie van de grootte van de hoop staat weergegeven in afbeelding 4.4. De totale uitvoeringstijd blijft



Afbeelding 4.2 Totaal aantal oproepen van de geheugensaneringsroutine i.f.v. de grootte van de hoop.

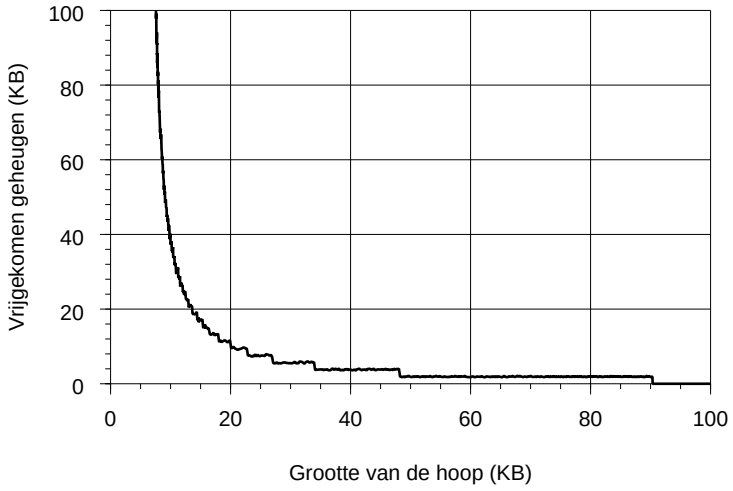
nagenoeg constant tot op 10% van de benodigde grootte van de hoop. Vanaf hoopgrootte 7 KB schiet hij dan plotseling de hoogte in. De grootte van de hoop is dus niet echt kritisch en kan variëren van 10 KB tot 100 KB zonder merkbaar prestatieverlies. Een te kleine hoop laat zich pas echt gelden vlakbij de minimale grootte van de hoop. Het oproepen van de geheugensaneringsroutine is anderzijds wel duur. Het markeren en compacteren kost ongeveer 1 ms per 1.5 KB. Voor een hoop van 100 KB komt dit neer op 66 ms. Het gebruikte algoritme houdt gelukkig wel bij hoever het gevorderd was tijdens de laatste oproep. Hierdoor moet slechts een stukje van de hoop opnieuw doorlopen worden.

4.3 De Multi-Prologtaken

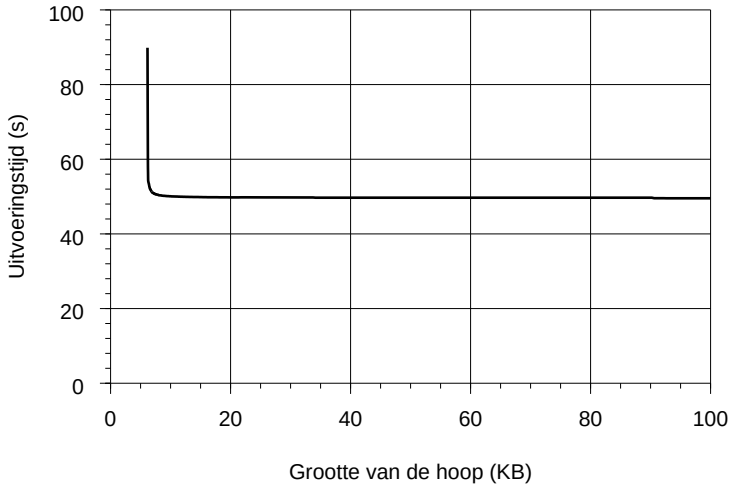
Op het taakniveau wordt er in de implementatie van MULTI-PROLOG onderscheid gemaakt tussen drie soorten taken.

(i) MTOS-UX-TAKEN. Dit zijn essentieel C of Assemblerroutines met een individuele stapel waarvan de uitvoering gecontroleerd wordt door de werkverdeler van MTOS-UX.

(ii) WAM-TAKEN. Dit zijn volledig onafhankelijk van elkaar uitvoerende WAM-vertolkers. Zij beschikken elk over de noodzakelijke stapels en een geheugensaneringsroutine om de uitvoering van een WAM-programma mogelijk te maken. Een WAM-taak wordt geïmplementeerd aan de hand van een



Afbeelding 4.3 Totale hoeveelheid gerecycleerd geheugen i.f.v. de grootte van de hoop.



Afbeelding 4.4 Evolutie van de uitvoeringstijd van het programma `master mind` i.f.v. de grootte van de hoop.

MTOS-UX-taak. Een WAM-taak wacht totdat het een Multi-Prologtaak te pakken krijgt en voert die dan uit totdat de Multi-Prologtaak termineert. Dan herneemt de cyclus zich.

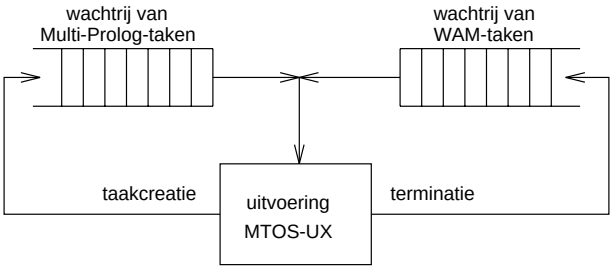
(iii) MULTI-PROLOGTAKEN. Dit zijn taken op het niveau van MULTI-PROLOG. Er kan slechts één voorgrondtaak, maar er kan daarentegen een overvloed aan achtergrondtaken zijn. Een Multi-Prologtaak kan slechts uitgevoerd worden nadat ze aan een WAM-taak toegewezen werd.

Bij het creëren van een taak wordt er een onderscheid gemaakt tussen het initialiseren van een taak (dit bestaat voornamelijk uit het reserveren en initialiseren van de gegevensstructuren), en het opstarten van een taak (het doorgeven aan de werkverdeler). Er zij nogmaals op gewezen dat het opstarten enkel betekent dat een taak bekend is bij de werkverdeler, niet dat ze reeds aan het uitvoeren is. Dit kan slechts wanneer de taak ook toegewezen wordt aan een processor.

Bij het initialiseren van de Multi-Prologomgeving wordt er eerst een bord aangemaakt en geïnitialiseerd. Naderhand worden er (in ons geval 50) WAM-taken geïnitialiseerd en door MTOS-UX opgestart. Deze WAM-taken gaan meteen in een wachttoestand omdat er nog geen Multi-Prologtaken aanwezig zijn. Van zodra de initiële vraag in de wachtrij met Multi-Prologtaken verschijnt (zie afbeelding 4.5), wordt ze door een WAM-taak uit de wachtrij gehaald en uitgevoerd. Nieuwe Multi-Prologtaken worden bij hun creatie in principe louter in de wachtrij met Multi-Prologtaken gestopt. Van zodra er een WAM-taakén een processor beschikbaar zijn, wordt de uitvoering van de Multi-Prologtaak aangevat. Om dit alles zo snel mogelijk te laten gebeuren worden in het prototype de Multi-Prologtaken van zodra ze geïnitialiseerd zijn meteen doorgegeven aan een beschikbare WAM-taak. Enkel indien er geen vrije WAM-taken beschikbaar zijn wordt een Multi-Prologtaak in de wachtrij opgenomen.

Merk terloops op dat in onze implementatie niet meer dan vier WAM-taken simultaan actief kunnen zijn[2]. Dit belet echter niet dat een groot aantal WAM-taken zich binnenin MTOS-UX in slapende toestand kan bevinden. In feite is het zo dat van zodra een Multi-Prologtaak aan een WAM-taak toegewezen is de controle volledig overgedragen wordt aan MTOS-UX die dan verder zal instaan voor de uitvoering van de taak, inclusief alle synchronisatie en communicatie. De uitbreiding van dit schema ligt voor de hand. Indien er meer geheugen beschikbaar komt, kunnen er meer WAM-taken gecreëerd worden. Indien er processoren toegevoegd worden zullen zij automatisch een deel van de belasting op zich nemen. Door het gebruik van dit schema kan de kost voor het opstarten van een Multi-Prologtaak binnen redelijke perken gehouden worden omdat er per nieuwe Multi-Prologtaak geen nieuwe WAM-taak moet gecreëerd

[2] Er zijn immers maar vier processoren.



Afbeelding 4.5 Interactie tussen WAM-taken en Multi-Prologtaken.

Tabel 4.2 Taakcreatietijden.

taak	tijd (μs)
MTOS-UX-taak	260.0
WAM-taak	515.0
Multi-Prologtaak	208.6

en opgestart worden. Een WAM-taak is klaar om een Multi-Prologtaak uit te voeren na het herinitialiseren van een klein aantal WAM-registers. De tijden voor de taakcreatie staan vermeld in tabel 4.2. De totale tijd nodig voor de creatie van een Multi-Prologtaak is ongeveer gelijk verdeeld tussen het initialiseren van de gegevensstructuren en het opstarten van de taak (beide goed voor ongeveer 100 μs). Zowel bij het initialiseren als bij het opstarten wordt MTOS-UX minstens éénmaal opgeroepen (40 μs per oproep). Bij het initialiseren gaat nogal wat tijd verloren met het zoeken naar het ingangsadres in de code van het doel dat gebruikt wordt om een taak te crëeren. Hierbij moet een term op de hoop omgezet worden naar een adres in de WAM-code. Uit tabel 4.2 blijkt dat het opnieuw gebruiken van WAM-taken de kost van de taakcreatie aanzienlijk weet terug te dringen. Het telkens opnieuw crëeren van een WAM-taak zou per creatie van een Multi-Prologtaak een additionele kost van 515 μs veroorzaken. Deze tijd bestaat uit de tijd nodig om een taak te initialiseren (255 μs) en de tijd nodig om een MTOS-UX-taak op te starten (260 μs).

4.4 Het Bord

Het belangrijkste onderdeel van de implementatie is echter de realisatie van het bord. Doordat het bord de enige manier van communiceren is, moet het bijzonder efficiënt geïmplementeerd worden, zoniet kan het de fatale flessehals voor het totale systeem worden. Daarom zal er onderzocht moeten worden op welke manier een maximale doorvoercapaciteit uit het bord kan gehaald worden.

In deze sectie zal de implementatie van het bord uitvoerig besproken worden. Er wordt begonnen met een zogenaamd naïef bordmodel. Dit is een nagenoeg letterlijke implementatie van het logische model. De implementatie van het naïeve model is echter niet zeer efficënt en daarom worden er optimalisaties voorgesteld die de doorvoercapaciteit van het bord zullen verhogen. De eerste optimalisatie heeft tot doel de gemiddelde lengte van de bordlijsten te verkorten door ze op te splitsen over verscheidene deelborden. Door de tweede optimalisatie zullen verscheidene taken simultaan hetzelfde deelbord kunnen gebruiken. Het uiteindelijke resultaat zal het gespreide bordmodel genoemd worden.

Het bord is essentieel een stuk gemeenschappelijk geheugen. Het gebruik ervan vereist een zekere vorm van geheugenbeheer. Dit geheugenbeheer maakt deel uit van de interne werking van het bord en is niet zichtbaar voor de gebruiker. Het is echter belangrijk voor de prestatie van het totale bord.

4.4.1 Het Geheugenbeheer

Om een gegeven op het bord te kunnen schrijven moet er vooraf noodzakelijk een hoeveelheid bordgeheugen gereserveerd worden. Gezien verscheidene taken simultaan een stuk geheugen kunnen aanvragen, moet het reserveren en vrijgeven ervan kritisch gebeuren om de integriteit van het geheugenbeheer te bewaren. Om de synchronisatieoverlast op de reservatie tot een minimum te beperken, wordt er per bordgebeurtenis eerst berekend hoeveel geheugen er in totaal vereist zal zijn. Deze hoeveelheid wordt dan inéén keer gereserveerd.

MTOS-UX voorziet in alle primitieven die nodig zijn om aan geheugenbeheer te kunnen doen. Deze primitieven zijn evenwel vrij traag omdat ze met alle taken in de multiprocessor moeten synchroniseren. Het geheugenbeheer van het bord staat echter los van al het andere geheugenbeheer in de multiprocessor en daarom werd er de voorkeur aan gegeven om de nodige programmatuur voor het geheugenbeheer op maat te ontwerpen.

Gezien er geen beperking bestaat op de maximale grootte van een bordterm, bestaat er dus ook geen bovengrens op de maximale grootte van een te reserveren geheugenblok. Gezien alle gereserveerde blokken contigu moeten zijn, wordt er geopteerd voor het gebruik van een partnersysteem. Dit algoritme

```
partnertest1 :- !t, ?t, !t, ?t, partnertest1.

partnertest2 :- !t1, !t2, ?t1, ?t2, partnertest2.
```

Programmafragment 4.1 `partnertest` evaluatieprogramma.

Tabel 4.3 Uitvoeringstijden van `partnertest` per cyclus, t.t.z., twee schrijf- en twee leesbewerkingen. Vertrekken van een leeg bord is nadelig omdat het reserveren en vrijgeven van het allereerste blok geheugen meer tijd vergt.

initieel bord	communicatieduur (μs)	
	`partnertest1`	`partnertest2`
leeg	161.9	128.7
niet-leeg	94.1	99.2

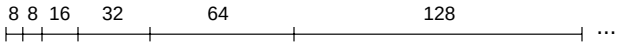
deelt het beschikbare geheugen op in blokken (partners) waarvan de grootte een macht van twee is. Per reservatie wordt steeds het kleinste blok dat nog net groot genoeg is gereserveerd. Soms vereist dit dat een groter blok moet opgesplitst worden.

Bovendien is het met dit algoritme ook mogelijk om op efficënte manier vrijgekomen blokken opnieuw samen te voegen tot grotere blokken. Op die manier kan verhinderd worden dat het totale geheugen te versnipperd geraakt. De implementatie van een dergelijk algoritme stelt verder geen enkele moeilijkheid.

Vermeldenswaard zijn echter wel een paar totaal onverwachte gedragingen van het partnersysteem die het noodzakelijk maakten de recombinatiestrategie enigszins aan te passen.

Om het partnersysteem uit te testen werd gebruik gemaakt van de twee bepalingen uit programmafragment 4.1. De eerste bepaling schrijft en leest afwisselend de term `t` op het bord. De tweede bepaling plaatst eerst twee termen `t1` en `t2` op het bord, en leest ze nadien in dezelfde volgorde van het bord. Voor het overige zijn de beide bepalingen volledig gelijklopend zodat er geen invloed te verwachten valt van factoren die niets met het bord zelf te maken hebben. Dit programma werd nu in twee verschillende omstandigheden uitgevoerd: (i) op een leeg bord, en (ii) op een bord metéén constante `a` (niet-leeg). De resultaten uit tabel 4.3 zijn op zijn minst merkwaardig. De resultaten voor een niet-leeg bord zijn duidelijk beter dan voor een leeg bord. De oorzaak ligt voor de hand: het reserveren van het allereerste blokje kost beduidend meer tijd dan het reserveren van een tweede blokje omdat de totale hoeveelheid geheugen[3] voor het eerste blokje moet opgesplitst worden in

[3] Dit is in onze implementatie te wijten aan het feit dat de initiële hoeveelheid geheugen precies een macht van twee is.



Afbeelding 4.6 Partnersysteem.

stukjes. De reservatie van de hoeveelheid geheugen nodig om de term `t` op te slaan (acht woorden) geeft aanleiding tot een onderverdeling zoals geschetst in afbeelding 4.6. Van zodra deze hoeveelheid geheugen vrijgegeven wordt zullen alle blokjes terug samengevoegd worden tot één blok dat de initiële hoeveelheid geheugen omvat. Van zodra er minstens één object op het bord aanwezig is, is het totale bordgeheugen opgesplitst en doordat het geheugen voor dit object niet vrijgegeven wordt tijdens de uitvoering van het programma kan het bordgeheugen nooit volledig gerecombineerd worden. Dit is te merken aan de communicatieduur.

Het evaluatieprogramma `partnertest1` geeft aanleiding tot een 'oscillerend' gedrag van het partnersysteem. Het programma `partnertest2` oscilleert eveneens, maar dan wel op de halve frequentie hetgeen duidelijk blijkt uit de communicatieduur per cyclus. Op het niet-lege bord doet `partnertest2` het iets minder goed omdat er op een bepaald ogenblik drie constanten op het bord staan wat het telkens opnieuw opsplitsen en nadien terug samenvoegen van een blok van 16 woorden noodzakelijk maakt.

Het gedrag voor het niet-lege bord kan gemakkelijk geforceerd worden door bij de initialisatie van het partnersysteem automatisch de kleinste hoeveelheid geheugen (8 woorden) te reserveren, zodat er na de initialisatie een blokje van elke grootte beschikbaar is. Dit komt de prestatie zeker ten goede, maar nog steeds blijft het effect van het opsplitsen en het achteraf terug recombineren bij sterk repetitieve evaluatieprogramma's voelbaar zoals blijkt uit tabel 4.4. Het programma `partnertest2` uitgevoerd op een leeg bord ondervindt hier nog steeds de negatieve effecten van het steeds opnieuw opsplitsen en recombineren van een blokje van zestien woorden. De resultaten liggen wel in de lijn van die van tabel 4.3.

Het vooraf plaatsen van één constante op het bord (niet-leeg bord) consumeert nu wel het enig beschikbare blokje van acht woorden waardoor opnieuw blokjes moeten opgesplitst en samengevoegd worden. Dit verklaart de iets hogere uitvoeringstijd voor `partnertest1`. Het mag verwonderlijk lijken dat `partnertest2` iets trager loopt op het lege bord dan op een niet-leeg bord. De oorzaak hiervoor is dat in beide gevallen een blokje van 16 woorden moet opgesplitst en nadien terug gerecombineerd worden. In het geval van een leeg bord komen de twee te recombineren blokjes uit een lijst met drie blokjes, in het geval van een niet-leeg bord komen deze blokjes uit een lijst met slechts

Tabel 4.4 Resultaten van `partnertest` met een vooraf gereserveerd blokje. Dit gedrag is vrij stabiel.

initieel bord	communicatieduur (μs)	
	`partnertest1`	`partnertest2`
leeg	92.5	97.4
niet-leeg	100.4	96.1

Tabel 4.5 Resultaten van partnertest met recombinatie op aanvraag.

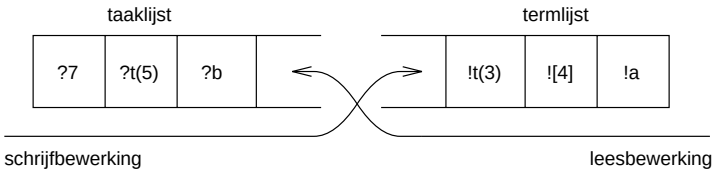
initieel bord	communicatieduur (μs)	
	`partnertest1`	`partnertest2`
leeg	88.0	86.9
niet-leeg	88.1	87.0

twee blokjes. De tijd nodig om de twee geschikte blokjes te vinden is dus iets korter.

De invloed van het partnersysteem in de communicatietijd kan echter nog verder teruggedrongen worden. Observatie van het gedrag van het partnersysteem maakt duidelijk dat bijna alle gereserveerde blokjes de grootte acht of zestien woorden hebben. Grotere blokjes komen slechts sporadisch voor[4]. Om de overlast ten gevolge van het repetitief opsplitsen en recombineren van blokjes tegen te gaan besloten wij om de recombinatie niet langer automatisch door te voeren bij het vrijkomen van een blokje, maar slechts op het ogenblik dat er geen blokje met geschikte grootte meer aanwezig was. Deze recombinatie op aanvraag zorgt voor een betere efficëntie van het partnersysteem omdat er zich na zekere tijd een evenwicht in beschikbare geheugenblokjes instelt. Deze kunnen dan efficiënt gereserveerd en terug vrijgegeven worden. Het nadeel van deze aanpak is wel dat in extreme omstandigheden het geheugen sneller uitgeput kan geraken. Het is immers best mogelijk dat er bij de aanvraag van een blok geheugen nog wel voldoende geheugen beschikbaar is, maar dat het niet gerecombineerd kan worden tot de vereiste grootte. De resultaten van dit algoritme staan vermeld in tabel 4.5. De resultaten zijn nu relatief onafhankelijk van (i) het programma en (ii) de initële inhoud van het bord én bovendien sneller dan alle vorige. Er moet wel vermeld worden dat de gegevens uit tabel 4.5 regimewaarden zijn. Zij worden dus gemeten nadat het programma reeds een tijdje aan het uitvoeren is en er zich een stabiel evenwicht van blokjes gevormd heeft.

We gaan nu verder met de studie van de bordmodellen.

[4] Dit is niet echt verwonderlijk als men bedenkt dat men in zestien woorden een structuur met twaalf constante argumenten of een lijst van zes constanten kan opslaan.



Afbeelding 4.7 Interne bordstructuur.

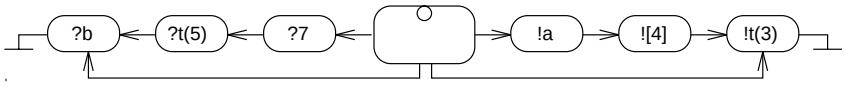
4.4.2 Het Conceptuele Bordmodel

Het bord is essentieel een chronologische ordening van gegevens (zie afbeelding 2.1). Het gebruik van een lijst legt meteen ook de volgorde van de gegevens chronologisch vast. Alle binnenkomende gegevens worden op het einde van de lijst bijgevoegd, en aanvragen voor gegevens worden verwerkt, te beginnen bij het begin van de lijst (oudste gegevens). De integriteit van het bord kan bewaard worden door de bordbewerkingen sequentieel op deze lijst uit te voeren. Dit is het model dat in het hoofdstuk over de taal voorgesteld werd (zie blz. 22).

4.4.3 Het Naïeve Bordmodel

Het conceptuele model bevat echter niet genoeg informatie om ook als model voor een implementatie gebruikt te kunnen worden. Er wordt bijvoorbeeld niet vermeld op welke manier de geblokkeerde taken bijgehouden moeten worden en hoe ze opnieuw geactiveerd kunnen worden. In de praktijk moet er zich ergens een lijst met wachtende taken bevinden. Gezien taken enkel kunnen wachten op een term, en gezien het bord louter termen opslaat, ligt het voor de hand de lijst met wachtende taken ook op het bord op te slaan. Het *werkelijke* bord bestaat dus niet uit één, maar uit twee lijsten (zie afbeelding 4.7), één voor de wachtende taken en één voor de termen. Wachtende taken worden genoteerd met een vraagteken omdat zij door het leespredicaat gecreëerd werden. Termen worden met een uitroepteken genoteerd omdat zij door het schrijfpredicaat gecreëerd werden. De structuur van het bord wordt hierdoor symmetrisch hetgeen de implementatie vergemakkelijkt. De verwerking van een borddoel gebeurt als volgt.

- Indien het een leesbewerking betreft, dan wordt eerst de termlijst doorlopen te beginnen met de oudste term. Indien een geschikte term gevonden wordt, wordt hij verwijderd uit de termlijst. Indien er geen geschikte term gevonden wordt, en het een blokkerende leesbewerking betreft dan wordt de taak achteraan de lijst met geblokkeerde taken (taaklijst) toegevoegd; zoniet faalt de leesbewerking.



Afbeelding 4.8 Concrete structuur van het naïef bordmodel.

- Indien het een schrijfbewerking betreft, dan wordt analoog eerst de taaklijst doorlopen om na te gaan of er een taak staat te wachten op de aangeboden term. Indien dit het geval is, wordt de taak uit de lijst genomen, wordt de aangeboden term rechtstreeks gecommuniceerd met de net verwijderde taak, waarna deze taak aan de werkverdeler gegeven wordt voor verdere verwerking. Indien er geen taak aan het wachten is, wordt de term op het einde van de termlijst bijgevoegd.

Om op efficiënte manier een element te kunnen toevoegen aan een lijst, wordt de geketende gegevensstructuur uit afbeelding 4.8 gebruikt. De centrale knoop bevat wijzers naar het oudste element van de termlijst (!a) en van de taaklijst (?7). Om snel een element te kunnen toevoegen aan het einde van een lijst wordt er in twee extra wijzers voorzien naar het laatste, meest recente, element uit de lijst.

Om de integriteit van het bord te verzekeren moeten er beperkingen opgelegd worden aan de taken die het bord willen gebruiken. Wij hebben gekozen voor semaforen omwille van de efficiëntie waarmee ze geïmplementeerd kunnen worden [Tay89]. Het feit dat ze op ongestructureerde manier gebruikt kunnen worden is van weinig belang omdat de bordprogrammatuur slechts éénmaal ontwikkeld wordt en voor de gebruiker van het bord zijn deze semaforen toch niet zichtbaar.

Om de kritische secties die door de semaforen beschermd worden zo klein mogelijk te houden, wordt er eerst een schema met twee semaforen geprobeerd, één voor de termlijst, en één voor de taaklijst. Alvorens een bordbewerking uitgevoerd wordt, wordt er één van beide semaforen vergrendeld[5]. Drie gevallen kunnen zich voordoen.

(i) Bij het overlopen van de lijst wordt een geschikt lijstelement gevonden. Het selecteerde element wordt uit de lijst verwijderd en de semafoor wordt vrijgegeven.

(ii) Er wordt geen element gevonden en er moet dus een element aan het eind van de andere lijst toegevoegd worden. Daarvoor moet de semafoor van die andere lijst vergrendeld worden. Dit kan op twee manieren gebeuren.

[5] Door de symmetrie maakt het niet uit of het nu gaat om een schrijfbewerking of een leesbewerking.

(i) Eerst wordt de semafoor van de andere lijst vergrendeld, en nadien wordt de eerst vergrendelde semafoor vrijgegeven. Het is niet moeilijk om in te zien dat twee simultane lees- en schrijfbewerkingen een patstelling kunnen veroorzaken indien zij simultaan een element in de 'andere' lijst willen creëren.

(ii) Eerst wordt de vergrendelde semafoor vrijgegeven, en pas nadien wordt de semafoor van de andere lijst vergrendeld. Jammer genoeg deugt ook deze aanpak niet. Onderstel dat de doelen `!a` en `?a` simultaan uitgevoerd worden, en dat er aan geen van beide meteen voldaan kan worden. Beide lijsten worden vrijgegeven, en zowel `!a` en `?a` worden aan de 'andere' lijst toegevoegd. Het gevolg is dat deze communicatie verloren gaat omdat unificatie tussen deze twee gebeurtenissen[6] enkel mogelijk is bij het doorlopen van de lijst, en deze fase is dan reeds achter de rug.

Hieruit kunnen we besluiten dat het synchronisatieschema op basis van twee semaforen niet goed werkt, en dat er, hoe jammer ook, moet gekozen worden voor één enkele semafoor voor de beide lijsten[7]. Het gevolg hiervan is dat alle bordbewerkingen puur sequentieel door de bordprogrammatuur moeten verwerkt worden. Dit is niet bepaald dramatisch indien de lijsten zeer kort zijn[8], maar geeft aanleiding tot onaanvaardbaar lange communicatietijden in het geval de lijsten lang worden.

De rest van deze sectie is nu gewijd aan twee optimalisaties die als doel hebben de tijd dat het bord vergrendeld is zoveel mogelijk te verkorten om aldus de doorvoercapaciteit te verhogen.

4.4.4 Het Gepartitioneerde Bordmodel

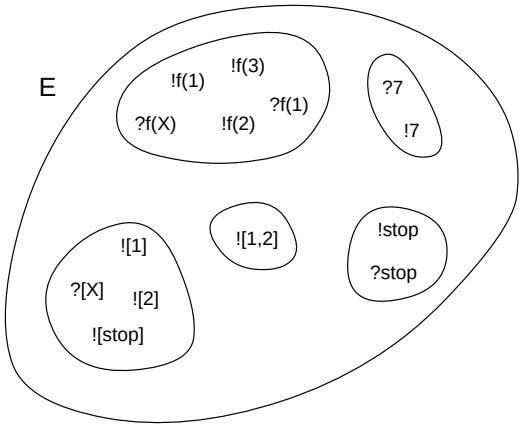
Het idee voor een gepartitioneerd bord is gegroeid vanuit de vaststelling dat het helemaal niet nodig is om trachten te unificeren met alle elementen van een lijst. Een voor de hand liggende vereiste is bijvoorbeeld dat het type van de te unificeren termen moet overeenstemmen.

Om duidelijk te maken wat er precies bedoeld wordt met partitionering van het bord, wordt er nu een concreet voorbeeld ingelast. Onderstel dat er geweten is dat een programma P enkel bordgebeurtenissen uit de verzameling E zal uitvoeren.

[6] Twee gebeurtenissen `!w` en `?r` unificeren indien `w` en `r` unificeren.

[7] In afbeelding 4.8 wordt een semafoor symbolisch weergegeven door een klein cirkeltje in de centrale knoop.

[8] In dit geval biedt één semafoor zelfs voordelen gezien er slechts één moet vergrendeld worden, en men zich na het vergrendelen totaal geen zorgen meer hoeft te maken over synchronisatie. Alle manipulaties op het bord zijn dan volkomen toegelaten.



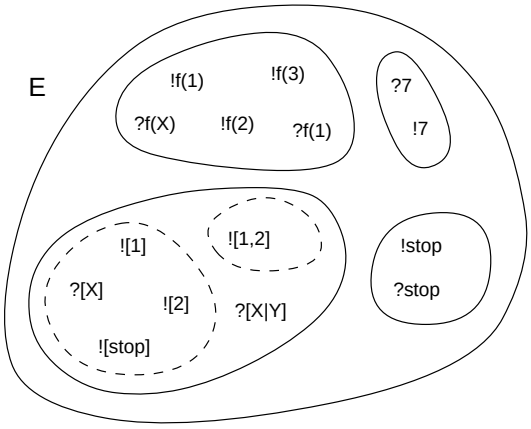
Afbeelding 4.9 Ideale partitie van E.

$$E = \{\ \texttt{!f(1)}, \texttt{!f(2)}, \texttt{!f(3)}, \texttt{?f(1)}, \texttt{?f(X)}, \texttt{!stop}, \texttt{?stop}, \texttt{?[X]}, \texttt{![1]}, \texttt{![2]}, \texttt{![stop]}, \texttt{![1,2]}, \texttt{!7}, \texttt{?7}\ \}$$

Een bordgebeurtenis hoeft dan in feite enkel maar rekening te houden met die (deel)verzameling van bordgebeurtenissen die potentieel aanleiding kunnen geven tot een succesvolle communicatie. Het gepartitioneerde bordmodel deelt de verzameling van bordgebeurtenissen op in een aantal deelverzamelingen van gebeurtenissen die op de één of andere manier met elkaar kunnen unificeren. De gepartitioneerde verzameling is geschetst in afbeelding 4.9. Het wordt nu mogelijk om elk element van de partitie te implementeren aan de hand van een afzonderlijke termlijst en een taaklijst (d.w.z. een deelbord). In het voorbeeld van afbeelding 4.9 kunnen we volstaan met een vijftal deelborden. Het effect van deze partitionering is dat enerzijds de communicatie een weinig zal vertragen omdat bij elke communicatie een deelbord moet gekozen worden. De communicatie zal anderzijds wel aanzienlijk versneld worden omdat de lengte van de lijsten zal inkrimpen. Daar waar de *maximale* lengte voorheen negen was, is deze nu slechts drie meer. De gemiddelde lengte van zowel de termlijst als de taaklijst zal hierdoor ook afnemen.

Op het eerste gezicht ziet dit model er veelbelovend uit, maar er zitten toch een tweetal addertjes onder het gras. Het eerste is van theoretische aard, het tweede vooral van praktische.

Het theoretische probleem kan het best geïllustreerd worden aan de hand van een voorbeeld. Onderstel dat we aan de verzameling E de gebeurtenis `?[X|Y]` toevoegen. Het effect op de partitie wordt geschetst in afbeelding 4.10. Het blijkt dat het toevoegen van een extra gebeurtenis de unie van twee ele-



Afbeelding 4.10 Ideale partitie van E met inbegrip van lijstbewerkingen.

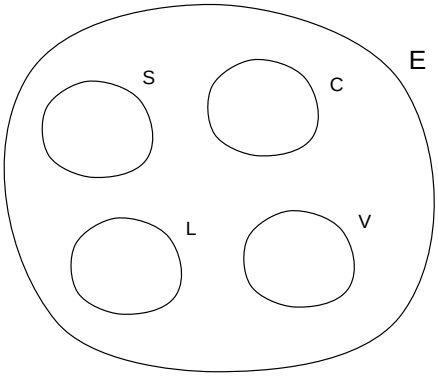
```
onbekende_gebeurtenis :- read(X), !X.
```

Programmafragment 4.2 Programma met een ongekende gebeurtenis.

menten van de partitie noodzakelijk gemaakt heeft om een correcte partitie te kunnen vormen. Dit is vervelend om twee redenen: (i) het aantal elementen van de partitie is verkleind, en hiermee samenhangend, (ii) zijn sommige deelverzamelingen groter geworden, wat de gemiddelde lijstlengte doet toenemen en dus nefast is voor de efficiëntie. Daarenboven moet opgemerkt worden dat het met `?[X|Y]` al bij al nogal meevalt. De gebeurtenis `?X` zorgt ervoor dat het aantal elementen van de partitie herleid wordt tot één gezien `?X` potentieel met eender welke term kan unificeren.

Het praktische probleem heeft te maken met de creatie en de selectie van de deelborden. Het probleem is dat de *totale* verzameling van alle bordgebeurtenissen voor een gegeven programma niet gekend is tijdens het compileren. Een typisch voorbeeld is programmafragment 4.2. Het is onmogelijk tijdens de compilatie te voorspellen wat voor soort bordgebeurtenis door dit programma gegenereerd zal worden. In de praktijk is het dus onmogelijk om een partitie te creëren tijdens het compileren omdat niet alle bordgebeurtenissen gekend zijn op dat ogenblik. Dit houdt in dat de partitie dynamisch zal moeten gecreëerd worden, en dat hiervoor de eerder beschreven partitionering niet in aanmerking komt omdat ze vertrekt van de totale informatie en derhalve niet incrementeel kan toegepast worden.

De dynamische creatie van een partitie zal extra overlast tijdens de uitvoering met zich meebrengen. Om deze overlast tegen te gaan, verkozen wij om



Afbeelding 4.11 Partitie op basis van het type.

initieel een vooraf gedefinieerde partitie te creëren en om te zorgen voor een soort van lidmaatschapsfunctie die efficiënt kan geëvalueerd worden en aangeeft tot welk element van de partitie een bepaalde bordgebeurtenis behoort. Om dit te realiseren moeten er twee problemen opgelost worden: (i) de keuze van een statische partitie, en hiermee samenhangend, (ii) het opstellen van een efficiënte lidmaatschapsfunctie.

Statische partitie

De voornaamste vereiste voor de statische partitie is dat elke bordgebeurtenis in een deelverzameling moet kunnen ondergebracht worden. Het hoofddoel is nog steeds het verkorten van de bordlijsten door ze op te splitsen. Het opsplitsen van een lijst zal slechts echt voordelig zijn indien naderhand ook slechts één van die lijsten moet doorzocht worden. Alle gebeurtenissen die potentieel met elkaar kunnen unificeren moeten dus noodzakelijk ondergebracht worden in dezelfde deelverzameling.

Een eerste, triviale, partitie is gebaseerd op het type van de te communiceren term. Deze partitie wordt schematisch voorgesteld in afbeelding 4.11 (C: constanten, V: vrije variabelen, L: lijsten, S: structuren). Voor elke bordgebeurtenis kan een element van de partitie gevonden worden, maar jammer genoeg volstaat het niet steeds om slechts één element van de partitie te doorlopen om een geschikte gebeurtenis te vinden. Dit is te wijten aan de aanwezigheid van vrije variabelen in de partitie. Alvorens een variabele op het bord te plaatsen (!X) moet men er zich van vergewissen dat er geen wachtende taken op het bord staan (in alle elementen van de partitie). Een variabele die op het bord geschreven wordt unificeert immers met eender welke leesbewerking die staat te wachten. Analoog met het lezen van een variabele (?X); eender welke term

op het bord kan met deze leesbewerking unificeren. De enige correcte manier om het lezen en schrijven van variabelen te implementeren is het beperken van de partitie tot één element. Dit zou echter het totale voordeel van een gepartitioneerd bord ongedaan maken. Er zal dus moeten gezocht worden naar een compromis dat de voordelen van het gepartitioneerde bord blijft bieden maar met zo weinig mogelijk toegevingen.

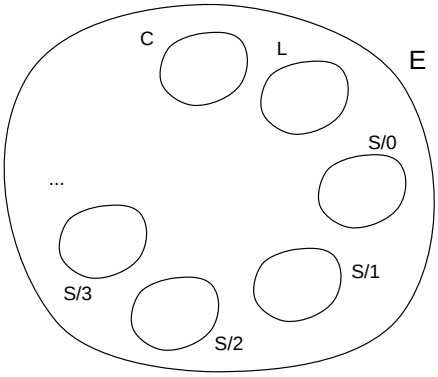
In de deelverzameling V kunnen slechts twee generische gebeurtenissen voorkomen, namelijk ?X en !X. De implementatie van deze twee gebeurtenissen is in feite eenvoudiger dan van een algemene bordgebeurtenis omdat het vinden van een geschikte term triviaal is. Eender welke term komt in aanmerking.

De eenvoudigste oplossing om het gebruik van variabelen zoals in ?X en !X te behouden is ze als een speciaal geval te beschouwen en ze buiten het partitiesysteem te houden. Het consequent laten samenwerken van deze gebeurtenissen met de rest van het bord beïnvloedt echter ook de efficiëntie van de andere bordgebeurtenissen omdat ook zij er ondanks alles toch rekening moeten mee houden.

Een eenvoudig voorbeeld kan dit duidelijk maken. Theoretisch moet ?X blokkeren indien het bord leeg is. De controle of een gepartitioneerd bord leeg is, is echter een dure aangelegenheid omdat alle deelborden eerst moeten vergrendeld worden. Zoniet kunnen tijdens het controleren van de deelborden eerder gecontroleerde deelborden opnieuw gemanipuleerd worden door andere taken. Het blokkeren van alle deelborden neemt een onaanvaardbare hoeveelheid tijd in beslag. Dit wil zeggen dat ?X gedurende een bepaalde tijd het totale bord zal monopoliseren. Analoog zal elke gebeurtenis moeten nagaan of er geen ?X of !X klaar staat om mee te unificeren. Zelfs voor programma's die geen gebruik maken van ?X of !X zal deze test een vaste bijkomende belasting betekenen. Bovendien schept het bijhouden van de volgorde waarin de gebeurtenissen plaatsvinden extra problemen indien dit ook tussen de elementen van de partitie moet gebeuren.

Bovendien dient men zich de vraag te stellen of men een gebeurtenis zoals ?X of !X wel echt nodig heeft. De voornaamste toepassing van ?X is wellicht het leegmaken van het bord. Hiervoor kan echter beter het ingebouwd `cleanbb`-predicaat gebruikt worden, i.p.v. de efficëntie van het totale systeem te belasten. Dit `cleanbb`-predicaat zal bovendien sneller zijn dan een expliciet Multi-Prologprogramma om het bord te wissen.

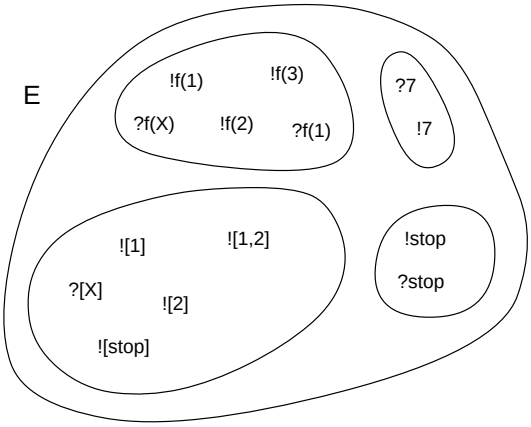
De voornaamste toepassing voor !X is wellicht om een taak die staat te wachten op een bordterm te deblokkeren. Inderdaad, een geblokkeerde taak zal de variabele van het bord lezen, en hierdoor actief worden. Heel veel zal de taak echter wel niet kunnen aanvangen met die variabele omdat hij vrij is.



Afbeelding 4.12 Partitie op basis van het type en het aantal argumenten.

Het beperkte nut van ?X en !X maakt ze in feite tot geschikte kandidaten om ze op te offeren aan het compromis dat we aan het zoeken zijn om een geschikte partitie te vinden. Door ze voortaan niet meer toe te laten, kan de deelverzameling V uit afbeelding 4.11 gewoon weggelaten worden. De resulterende partitie, bestaande uit drie elementen, voldoet dan aan de beide voorwaarden (elke gebeurtenis kan in één van de elementen van de partitie ondergebracht worden, en per gebeurtenis hoeft er slechts één element doorzocht te worden).

Een partitie bestaande uit drie elementen is echter nogal beperkt. Het valt te verwachten dat structuren en lijsten het meest frequent als gegevens zullen gebruikt worden. Bij de structuren kan er een verdere onderverdeling doorgevoerd worden op basis van het aantal argumenten van de structuur. Naast het type moet immers ook het aantal argumenten overeenstemmen. Bij lijsten kan een dergelijke opsplitsing op basis van bijvoorbeeld de lengte van de lijst niet doorgevoerd worden. De toegelaten gebeurtenis ?[X|Y] vereist dat alle lijsten in hetzelfde element van de partitie ondergebracht worden. Het verbieden van ?[X|Y] is te drastisch om te overwegen. Het resultaat van de verfijning van de partitie is te zien in afbeelding 4.12. Een praktisch probleem bij deze partitie is het in principe oneindig aantal elementen. In praktijk zal het aantal argumenten wel beperkt zijn tot een bepaalde waarde, maar nog zal het aantal elementen van de partitie groot zijn, en zullen de meeste elementen zelden of nooit gebruikt worden. Een praktische oplossing voor dit probleem is om in plaats van elk element van de partitie op een afzonderlijk deelbord af te beelden, een aantal elementen op hetzelfde deelbord af te beelden. Wellicht de eenvoudigste manier is om ze af te beelden modulo het aantal deelborden. Door het aantal deelborden tot bijvoorbeeld tien te beperken zullen dan S/n, S/(n+10), S/(n+20),... op elkaar afgebeeld worden.



Afbeelding 4.13 Statische partitie.

Gezien er nu een middel gevonden werd om een partitie met een groot aantal elementen af te beelden op een beperkt aantal deelborden, kan de partitie uit afbeelding 4.12 nog verder verfijnd worden. Constanten kunnen elk hun individueel element van de partitie krijgen (de constante a kan enkel door de gebeurtenis ?a gelezen worden omdat ?X niet meer toegelaten is). Ook bij structuren kan er een bijkomend onderscheid gemaakt worden op basis van de naam van de structuur. Dit zal finaal aanleiding geven tot een partitie met enorm veel elementen. Deze kunnen echter steeds afgebeeld worden op een kleiner aantal deelborden. Het uiteindelijke resultaat van het voorbeeld uit het begin van deze sectie is te zien in afbeelding 4.13. Er valt op te merken dat zelfs na de afbeelding van deze partitie op de statische deelborden slechtséén deelbord volledig moet doorzocht worden om een bordgebeurtenis te kunnen verwerken. Wel kan niet meer gegarandeerd worden dat dit deelbord werkelijk het minimaal aantal elementen zal bevatten. Door de statische partitie en de eerder besproken afbeelding kunnen immers een aantal elementen uit de partitie samengevoegd worden.

Lidmaatschapsfunctie

Het wordt nu mogelijk een lidmaatschapsfunctie te construeren die op eenduidige en snelle manier zal berekenen tot welk element van de partitie een bepaalde gebeurtenis behoort.

De berekening van de lidmaatschapsfunctie stelt geen enkel fundamenteel probleem en is gebaseerd op een combinatie van het type van de gebeurtenis en, afhankelijk van het type een hoeveelheid extra informatie. Voor symboli-

sche constanten is dit de ingang in de symbolentabel, voor numerische constanten de waarde, en voor structuren de ingang van de functienaam in de symbolentabel en het aantal argumenten van de structuur. Het resultaat van deze berekening is een geheel getal.

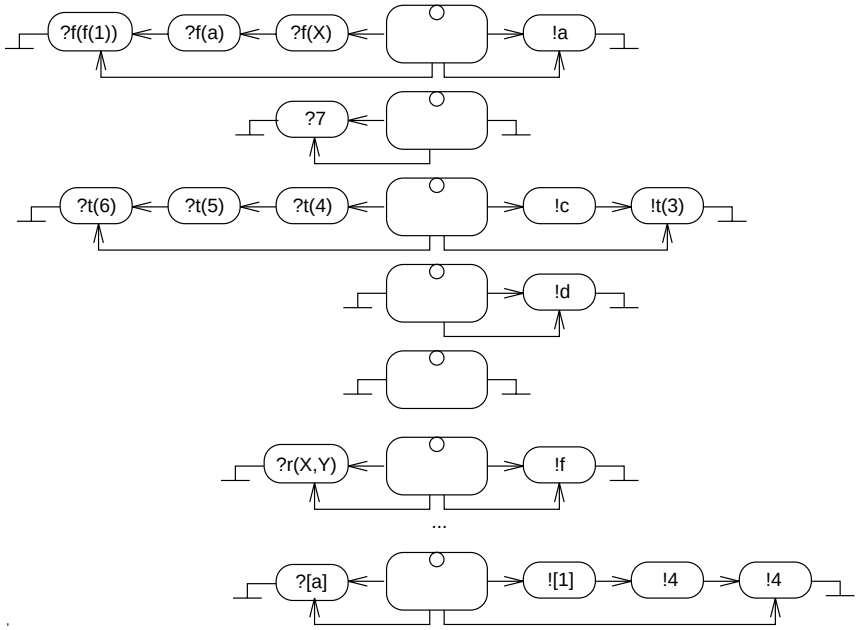
Afbeelding op een deelbord

De afbeelding van het getal dat door de lidmaatschapsfunctie geproduceerd wordt op één van de statische deelborden gebeurt aan de hand van een moduloberekening met het aantal deelborden.

Het is moeilijk voorspelbaar welke elementen van de partitie in deze stap op hetzelfde deelbord zullen afgebeeld worden. Dit hangt af van de symbolentabel die door de compilator gegenereerd wordt. Een kleine wijziging aan het programma zou in principe een totaal andere verdeling over de deelborden kunnen opleveren. In de praktijk heeft dit echter nooit problemen opgeleverd. De kennis van de interne structuur van het bord zoals zonet besproken maakt het wel mogelijk om de te communiceren termen zodanig te kiezen dat ze wellicht zullen behoren tot verschillende deelborden. Het zou bijvoorbeeld een slecht idee zijn om twee onafhankelijke communicatiestromen dezelfde structuur `f/1` te geven. Dit zou de snelheid van de communicatie zeker niet ten goede komen. Het geven van twee verschillende namen zoals `f/1` en `g/1` is zeker een beter idee alhoewel ook dit geen garantie biedt dat zij niet op hetzelfde deelbord afgebeeld zullen worden. De kans dat ze op hetzelfde deelbord afgebeeld worden is gemiddeld gezien ongeveer omgekeerd evenredig met het aantal statische deelborden (in ons geval 64, d.w.z. dat de kans kleiner is dan 2%).

De uiteindelijk concrete structuur van het bord is weergegeven in afbeelding 4.14. Het is essentieel een n-voudige replica van het naïeve systeem. De inhoud van het bord in dit voorbeeld is verschillend van de zonet besproken partities omdat er in een concreet bord geen onderling unificerende gebeurtenissen mogen opgeslagen worden.

Het geheugenbeheer zou in plaats van centraal voor het bord, nu ook per deelbord georganiseerd kunnen worden, gezien het toevoegen en het verwijderen van termen steeds binnen hetzelfde deelbord gebeurt. Het voorzien in een geheugenbeheer per deelbord geeft echter moeilijkheden indien men de hoeveelheid bordgeheugen zo optimaal mogelijk wil gebruiken. Bij een statische geheugenverdeling is het geheugen dat voorbehouden werd voor de deelborden die achteraf leeg blijken te zijn verloren. Een dynamische verdeling van het geheugen over de deelborden (een hërarchisch geheugenbeheersysteem dus) biedt een oplossing, maar is een heel stuk complexer en veroorzaakt ook een bijkomende overlast. Gezien het geheugenbeheer niet echt een flessehals was werd er niet overgegaan tot de implementatie van dit complex schema.



Afbeelding 4.14 Concrete structuur van een gepartitioneerd bord.

Door de aard van de moduloafbeelding van de elementen van de partitie op de deelborden kan het voorkomen dat een aantal elementen van de partitie toch op hetzelfde deelbord afgebeeld wordt, en dit ondanks het feit dat er nog heel wat vrije deelborden zijn. De afbeelding op een deelbord houdt immers geen rekening met het al dan niet beschikbaar zijn van lege deelborden omdat ze statisch is. De afbeelding kan vrij gemakkelijk dynamisch gemaakt worden door in plaats van een moduloafbeelding een meer complexe afbeelding te gebruiken. Dit kan door een lijst met de afbeelding in het geheugen op te bouwen. Bij elke communicatie moet eerst het element van de partitie berekend worden en nagegaan worden of dit element reeds eerder afgebeeld werd op een deelbord. Indien wel, dan wordt het corresponderende deelbord gebruikt. Indien niet, dan wordt een nieuw deelbord genomen. Pas nadat alle deelborden in gebruik zijn, ontstaat de noodzaak om een deelbord dubbel te gebruiken. De afbeelding op een deelbord kan al dan niet ongedaan gemaakt worden op het ogenblik dat een deelbord terug leeg wordt. Op die manier kan het sneller opnieuw gebruikt worden zonder conflicten.

Het bijhouden van de afbeelding kan op twee manieren. Ofwel wordt het nummer van het deelbord telkens opnieuw herberekend door de lijst met de afbeelding te overlopen. Dit is echter te tijdrovend gezien de lijst die de af-

beelding bevat associatief moet doorzocht worden om te controleren of aan een bepaald element van de partitie al dan niet reeds een deelbord toegekend werd. Het alternatief is dat het nummer van het deelbord opgeslagen wordt samen met de symbolentabel (die gebruikt wordt om het nummer van de partitie te berekenen). Dit maakt de symbolentabel zwaarder en neemt niet weg dat het nummer van het deelbord toch een eerste maal zal moeten berekend worden. We zijn niet overgegaan tot de implementatie van dit schema van alternatieve deelbordafbeelding omdat het vrij veel overlast met zich meebrengt en omdat het berekenen van het deelbord geen flessehals in ons systeem is.

Samenvattend kunnen we stellen dat door het laten vallen van de gebeurtenissen `?X` en `!X` we er in geslaagd zijn om een efficiënt partitioneringsalgoritme te creëren. Hierbij hebben we zelfs geen gebruik moeten maken van een dynamische creatie van de partitie of van de deelborden. De partitie is statisch en ook alle deelborden worden statisch aangemaakt. Enkel de berekening van het element van de partitie en de afbeelding ervan op een deelbord gebeurt tijdens de uitvoering, maar zou in heel wat gevallen (b.v. `!f(1)`) op voorhand door de compilator kunnen gebeuren. Gezien de efficiëntie waarmee dit gebeurt, was er niet de noodzaak om over te gaan tot deze vorm van optimalisatie.

Uiteindelijk zijn de voordelen van een gepartitioneerd bord tweeledig.

1. De gemiddelde lengte van de gepartitioneerde bordlijsten wordt korter. Dit verkort de tijd nodig om de lijsten te doorzoeken, en dus ook de tijd dat de lijsten vergrendeld blijven. Het aantal gebeurtenissen dat per tijdseenheid kan verwerkt worden, neemt hierdoor toe.

2. Gebeurtenissen die tot verschillende deelborden behoren kunnen nagenoeg onafhankelijk van elkaar verwerkt worden doordat de lijsten enkel lokaal moeten vergrendeld worden.

Het nadeel van partitionering is dat de winst niet hard voorspelbaar is. In het slechtste geval worden alle gebeurtenissen toch op hetzelfde deelbord afgebeeld en is er helemaal geen winst. De kans dat dit zich in de praktijk zou voordoen is echter klein, zeker als men op de hoogte is van de manier waarop de partitie berekend wordt. In de praktijk heeft zich nooit echt een probleem voorgedaan, maar de regel blijft niettemin heuristisch.

Partitionering van het bord of de veeltallenruimte is een vaak toegepaste techniek in Linda-implementaties. De hier beschreven methode lijkt bijzonder goed op de implementatie van de veeltallenruimte in [PM91], maar is wel algemener. Dit artikel bespreekt de invoering van de Lindaprimitieven in de taal JOYCE, een Pascalachtige taal met communicatieprimitieven die gebaseerd zijn op Hoare's CSP [Hoa78].

De veeltallen in de veeltallenruimte hebben elk een uniek type (samengesteld uit de types van de componenten). Een voorwaarde opdat een Lindabe-

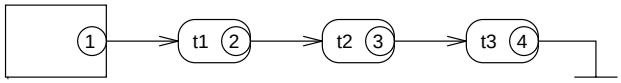
werking kan uitgevoerd worden is dat de types overeenstemmen. Het is dus gemakkelijk om de veeltallenruimte op te delen in functie van het type van de veeltallen. Het probleem van ?X en !X stelt zich hier niet omdat elke Joycevariabele in principe een type heeft. Bovendien eisen sommige implementaties dat de eerste component van het veeltal een waarde is (geen formele parameter dus). Dit maakt het partitioneren nog een stuk gemakkelijker. De structuur die voorgesteld wordt in [PM91] is precies dezelfde als het voorstel in deze sectie: een deelveeltallenruimte bestaat uit een lijst met beschikbare veeltallen en een lijst met wachtende taken. De Lindabewerkingen worden sequentieel per deelveeltallenruimte afgewerkt.

4.4.5 Het Gespreide Bordmodel

Ondanks het gebruik van het gepartitioneerd bordmodel blijft het bord nog steeds dé flessehals van het systeem indien het aantal effectief gebruikte deelborden klein is, of het aantal termen groot. Dit werd onder andere vastgesteld bij het gedrag van de demultiplexer (zie blz. 155). Een deelbord wordt daar simultaan doorzocht door een aantal taken. Telkens wanneer een taak klaar is met de verwerking van een term, overloopt zij de termlijst om een nieuwe term te vinden. Indien de lijst lang wordt, worden de wachttijden onaanvaardbaar lang. Dit is bijzonder erg omdat de taken klaar zijn om een taak uit te voeren, maar niet tijdig aan een term geraken.

Het basisprobleem is eigenlijk dat een deelbord geblokkeerd blijft gedurende de hele periode dat een taak de lijsten aan het doorlopen is. In feite is dit onnodig lang en zou een taak enkel exclusieve toegang moeten hebben tot precies dat deel van het bord dat door een taak kan veranderd worden, én enkel op het ogenblik dat deze taak een verandering wil aanbrengen. Een gedeelte van het bord zal dus zeker kritisch moeten gemaakt worden om een element uit een lijst weg te halen, of om een element aan een lijst toe te voegen. Om een element te lezen zou men in principe geen exclusieve toegang moeten hebben.

De synchronisatie van het bord heeft echter een dubbel doel. Naast het garanderen van de *integriteit van het bord* moet er ook zorg gedragen worden voor de *preventie van patstellingen*. Het probleem van de integriteit en preventie van patstellingen kwam ook reeds ter sprake in het naïeve bordmodel, en gaf toen aanleiding tot het voorzien in slechts één semafoor voor het totale bord (later per deelbord in het gepartitioneerde model). Deze twee doelstellingen van de synchronisatie zullen aan een nader onderzoek onderworpen worden.



Afbeelding 4.15 Detail van een lijst met één semafoor per lijstelement.

Integriteit van het bord

Om de integriteit van het bord te kunnen verzekeren moet er ten minste exclusieve toegang kunnen gegarandeerd worden bij het toevoegen van een element aan of bij het verwijderen van een element uit de bordlijsten. Jammer genoeg heeft dit ook consequenties voor de taken die de lijstelementen louter consulteren. Het mag immers niet mogelijk zijn dat een taak aan het unificeren is met een element dat op datzelfde ogenblik door een andere taak uit de lijst verwijderd wordt. Om dit te vermijden zullen we er moeten voor zorgen dat zelfs het consulteren van een lijstelement exclusief gebeurt. Een lijst kan enkel ongesynchroniseerd geraadpleegd worden indien men kan garanderen dat ze stabiel is, en dus niet terzelfder tijd door een andere taak veranderd wordt. Om exclusieve toegang te verlenen zijn er twee manieren.

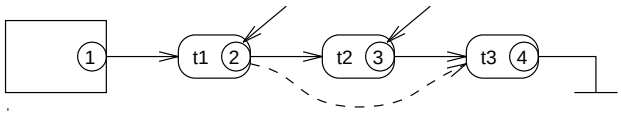
(i) Een globale vlag per lijst die aangeeft of de lijst stabiel is. Taken mogen de lijst enkel consulteren in een stabiele fase. Probleem is dat deze voorwaarde voortdurend gecontroleerd moet worden. Er kan niet gegarandeerd worden dat een taak die de lijst enkel consulteert tijdig op de hoogte zal zijn dat de lijst onstabiel geworden is. Dit kan enkel door de vlag als een semafoor te beschouwen en te vergrendelen alvorens de lijst te consulteren. Dit brengt ons echter terug bij het gepartitioneerde bordmodel van een vorige sectie.

(ii) Een lokale semafoor per lijstelement zoals in afbeelding 4.15. Alle taken die toegang willen hebben tot een lijstelement moeten dan eerst deze semafoor vergrendelen. Zoniet moeten ze wachten totdat de semafoor vrijkomt. Dit is een maatregel die een extra belasting met zich meebrengt, maar het is een noodzakelijk kwaad als we het bord door verscheidene taken simultaan willen laten manipuleren.

De keuze van één semafoor per element legt nog niet vast hoe de overgang tussen twee elementen zal gebeuren. Twee mogelijkheden dienen zich aan.

(i) Alvorens de semafoor van het volgende element te vergrendelen wordt de semafoor van het huidige element vrijgegeven.

(ii) Alvorens de semafoor van het huidige element vrij te geven wordt de semafoor van het volgend element vergrendeld.



Afbeelding 4.16 Detail van het verwijderen van een element uit een lijst.

Het is gemakkelijk om in te zien dat de eerste mogelijkheid niet in aanmerking komt als synchronisatieschema. Een taak mag de semafoor van haar huidige lijstelement t1 niet vrijgeven alvorens zij de semafoor van het volgend element t2 heeft weten te vergrendelen omdat anders de kans bestaat dat een andere taak element t1 uit de lijst verwijdert en er hierdoor een corrupte wijzer zou ontstaan.

Preventie van Patstellingen

De zonet geschetste oplossing voor het probleem van de integriteit van het bord staat weergegeven in afbeelding 4.15. Hierbij wordt er voorzien in één lokale semafoor per element. De meest voor de hand liggende plaats om de semafoor van een element op te slaan is de lijstknoop van het element zelf. Dit schept echter een probleem bij het verwijderen van een element. Dan zal de semafoor van de vorige knoop immers ook vergrendeld moeten worden omdat er een wijzer moet aangepast worden zoals te zien is in afbeelding 4.16. Het vergrendelen van de voorgaande knoop t1 kan echter een patstelling veroorzaken omdat deze knoop reeds vergrendeld kan zijn door een andere taak en deze taak de knoop t1 niet mag vrijgegeven alvorens t2 door haar vergrendeld is.

Om dit te vermijden wordt er de voorkeur aan gegeven om de semafoor niet op te slaan in de lijstknoop van het element zelf, maar in de knoop van het vorige element. De semafoor beschermt dus niet enkel het element, maar ook de wijzer naar de knoop die het element bevat. Om een knoop te verwijderen moet dan niet de semafoor van de vorige knoop, maar van de volgende knoop vergrendeld worden[9], en dit kan nooit een patstelling veroorzaken.

In het naïeve bordmodel kon er mogelijk een patstelling ontstaan bij het toevoegen van een element. Om dit op te lossen werd er gekozen voor één enkele semafoor om het toevoegen van een nieuw element te controleren. Ook nu zullen wij tot deze maatregel moeten overgaan.

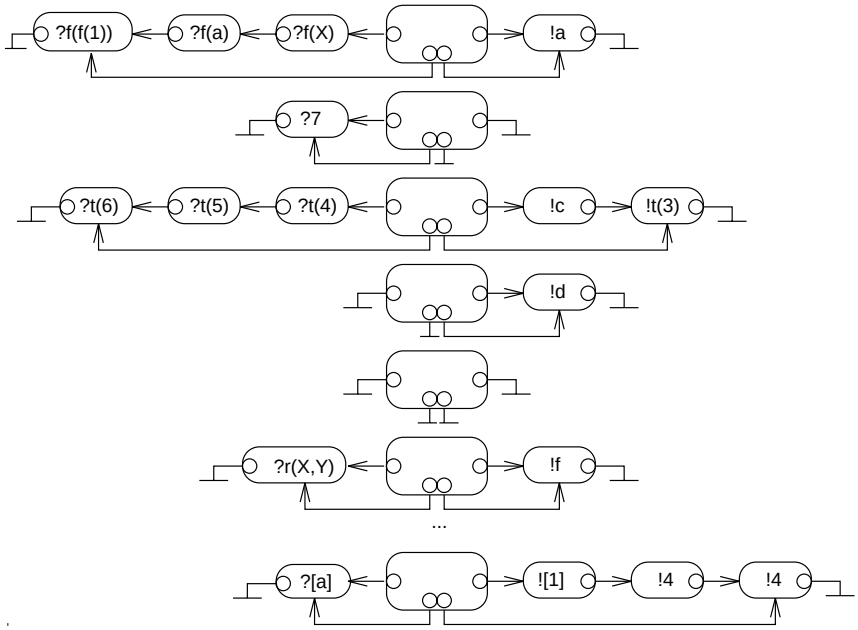
Indien bij het doorlopen van de lijst geen geschikt element gevonden werd,

[9] Aangezien deze semafoor samen met de knoop verwijderd wordt, moet men er eerst eigenaar van worden. Zoniet zouden er corrupte wijzers kunnen ontstaan.

en uiteindelijk semafoor 4 vergrendeld werd, en semafoor 3 vrijgegeven werd, moet er overgegaan worden tot het toevoegen van een element aan het einde van de andere lijst. Om een element aan het einde van de andere lijst toe te voegen moet echter de semafoor uit de laatste knoop aldaar ook vergrendeld worden. Indien nu een taak op het einde van die lijst precies dezelfde bewerking wil uitvoeren, belanden we onvermijdelijk in een patstelling. Om dit conflict op te lossen zou één van beide taken de semafoor van een eindknoop moeten vrijgeven. Het vrijgeven van een knoop maakt dan de weg vrij voor het toevoegen van extra knopen. Het probleem is echter dat de taak die de semafoor van de eindknoop heeft vrijgegeven in principe de volledige lijst opnieuw moet overlopen, te beginnen met de oudste term. Door het vrijgeven van de enige semafoor is immers ook de wijzer naar de laatste knoop niet meer te vertrouwen. Een alternatief is om de voorlaatste semafoor proberen te vergrendelen, maar dit is ook niet mogelijk omdat dit een patstelling kan veroorzaken.

Een oplossing bestaat erin in een centrale semafoor te voorzien die het toevoegen van een knoop controleert. Deze semafoor wordt in de centrale bordknoop opgenomen. De functie van deze semafoor kan bovendien uitgebreid worden om ook de wijzers naar het einde van de termlijst en de taaklijst te beschermen. Daardoor zal deze semafoor ook moeten vergrendeld worden indien de laatste knoop uit een lijst moet verwijderd worden. De semafoor beschermt dus in feite elke manipulatie van de eindknopen van de beide lijsten. Om te vermijden dat de taaklijst vergrendeld moet worden om de eindknoop van de termlijst te verwijderen kan deze semafoor beter opgesplitst worden in twee semaforen, één die de wijzer naar de laatste knoop van de taaklijst beschermt en één die de wijzer naar de laatste knoop van de termlijst beschermt. Het verwijderen van een knoop vereist dan het vergrendelen van slechts één van beide semaforen. Het toevoegen van een knoop in één van beide lijsten vereist dat beide semaforen vergrendeld worden. Om een patstelling te vermijden moeten ze steeds in dezelfde volgorde vergrendeld worden. Er wordt actief gewacht om beide semaforen te vergrendelen. De centrale semaforen worden de *arbitersemaforen* genoemd omdat zij instaan voor de arbitrage tussen taken die aan het einde van een lijst gekomen zijn (zie afbeelding 4.17).

Het gewoon toevoegen van deze twee semaforen lost echter niet alle problemen op. Nog steeds kan het voorvallen dat twee taken elk aan het einde van één van de lijsten gekomen zijn, en een element willen toevoegen aan het einde van de andere lijst. De semaforen in de eindknopen zijn dan vergrendeld, en in principe is het onmogelijk een element toe te voegen omdat men niet het recht heeft om de laatste wijzer aan te passen. In dit geval wordt er gekozen voor een compromis. De arbitersemaforen hebben voorrang op de lokale semaforen. Dit wil zeggen dat eenmaal zij vergrendeld zijn, een taak de wijzer in



Afbeelding 4.17 Concrete structuur van een gespreid bord.

de laatste knoop naar believen mag wijzigen, ook al heeft zij de semafoor ervan niet vergrendeld.

Dit heeft wel als gevolg dat een taak die erin geslaagd is de arbitersemaforen te vergrendelen, extra moet controleren of de wijzer die door zijn lokale semafoor beschermd wordt ondertussen niet veranderd werd door een andere taak. Indien dit wel gebeurd is (in dit geval zullen er elementen toegevoegd zijn[10]), moeten de arbitersemaforen terug vrijgegeven worden, en moet de lijst verder afgelopen worden. Indien er dan nog geen geschikt element gevonden werd, moet de taak de arbitersemaforen opnieuw zien te bemachtigen.

Op deze manier wordt er gezorgd voor de integriteit van niet enkel het bord, maar ook van zijn inhoud. Twee unificerende bordgebeurtenissen zullen nooit simultaan op het bord opgeslagen worden, maar steeds unificeren op het ogenblik dat ze de bordlijsten overlopen.

De uiteindelijke bordarchitectuur is geschetst in afbeelding 4.17. Het grote voordeel van dit gespreide bordmodel is dat het sneller is in het geval er lange lijsten, of veel taken zijn. Dit volgt trouwens duidelijk uit de simulatieresultaten op het einde van dit hoofdstuk. De schaalbaarheid van het bord wordt er ook fel door verbeterd. Kleine bordtoepassingen kunnen jammer genoeg niet

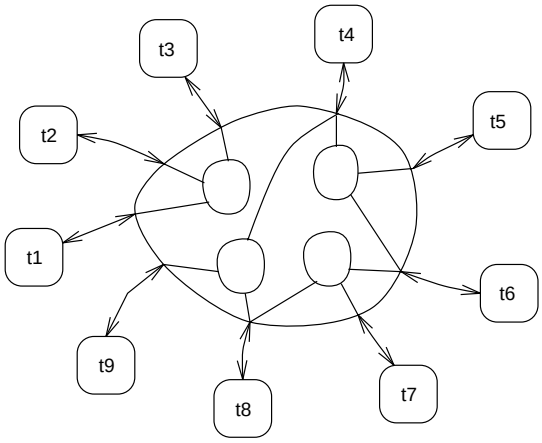
[10] Het verwijderen van de eindknoop kan enkel door de taak zelf.

van deze versnelling profiteren. In het geval een bordlijst leeg is, of slechts één element bevat, zal immers steeds het complexe algoritme met de arbitersemaforen moeten gebruikt worden. Dit wil zeggen dat in plaats van één semafoor in het naïeve model, er nu één semafoor om de knoop te beschermen, één semafoor die het einde van de lijst beschermt, en twee arbitersemaforen moeten vergrendeld worden alvorens een nieuw element kan toegevoegd worden aan een lijst met één element.

In het gespreide bordmodel wordt de chronologie van de binnenkomende gebeurtenissen gerespecteerd door eerst de volgende semafoor te vergrendelen en pas dan de vorige vrij te geven. Deze manier van werken creëert een ononderbroken ketting van kritische secties waardoor taken elkaar niet kunnen inhalen. Dit is een gevolg van het gebruikte synchronisatieschema, maar wordt niet opgelegd door MULTI-PROLOG. De enige opgelegde restrictie is dat de termen op het bord moeten opgeslagen worden 'in volgorde van aankomst'. Voor termen die door een sequentiële taak op het bord gezet worden is aan deze voorwaarde triviaal voldaan omdat de tweede bordbewerking slechts kan uitgevoerd worden nadat de eerste tot een goed einde gebracht is. Voor termen die door verschillende taken op het bord gezet worden is er geen 'eerste' en 'laatste' aan te duiden. Daar is het laatste woord aan het bord. De bordbewerking die als eerste een lijst kan beginnen af te lopen zal ook als eerste zijn term op het bord kunnen plaatsen omdat taken elkaar tijdens het aflopen van de lijsten van een deelbord niet kunnen inhalen.

Gezien de taken een relatief korte tijd nodig hebben om met een lijstelement trachten te unificeren, worden er actief wachtende semaforen gebruikt. Dat wil zeggen dat een taak die moet wachten op een semafoor, zijn processor niet verliest, maar actief zal staan wachten totdat de semafoor vrijgegeven wordt. Dit werd zo geïmplementeerd omdat het doen wachten van een taak vele malen langer duurt dan het unificeren met een element. Tegen de tijd dat de taak zich effectief in de wachttoestand bevindt zal de semafoor waarop ze aan het wachten is reeds vrijgegeven zijn. Het is dus beter gewoon actief te wachten. Dit wordt bevestigd door de metingen in afbeelding 4.21 op blz. 125.

Het beeld van een Multi-Prologprogramma zoals geschetst in de inleiding in afbeelding 1.3 kan nu verfijnd worden tot het beeld van de prototype-bordimplementatie in afbeelding 4.18. Hierin is het effect van het gepartitioneerde bord duidelijk te zien. De hier voorgestelde architectuur is voor zover wij weten origineel voor de bordgebaseerde talen. De implementatie uit [PM91] gaat niet verder dan het gebruik van een gepartitioneerd bord. Een verklaring hiervoor kan zijn dat indien men complexere communicatieprimitieven gaat implementeren men liever meteen het complete bord vergrendelt en dat het gebruik van het gespreide bordmodel minder voordelen geeft voor implementaties op een multiprocessor zonder gemeenschappelijk geheugen.



Afbeelding 4.18 Logische bordarchitectuur.

De drie zonet besproken bordmodellen werden effectief geïmplementeerd en geëvalueerd. De resultaten hiervan zijn verder in dit hoofdstuk terug te vinden.

4.5 De Werkverdeler

De taal MULTI-PROLOG dringt geen bepaalde strategie voor de werkverdeler op aan de implementatie. Deze keuze wordt volledig opengelaten. In de prototype-implementatie werd er geëxperimenteerd met twee soorten werkverdelers. Een preëmptieve werkverdeler en een niet-preëmptieve werkverdeler.

Preëmptieve werkverdeler. Bij een preëmptieve werkverdeler wordt de processortijd van de vier processoren zoveel mogelijk gelijkmatig verdeeld over de taken die klaar zijn voor uitvoering. Dit impliceert dat taken door de werkverdeler op regelmatige tijdstippen onderbroken worden om een andere taak de gelegenheid te geven verder uit te voeren.

Niet-preëmptieve werkverdeler. Bij een niet-preëmptieve werkverdeler worden taken nooit door de werkverdeler onderbroken. Zij verliezen enkel de processor indien zij om de één of andere reden niet verder meer kunnen uitvoeren (omdat zij moeten wachten op een synchronisatiepunt bijvoorbeeld).

Beide systemen hebben zo hun voor- en nadelen. Een preëmptieve strategie is iets algemener omdat alle taken min of meer gelijkmatig uitgevoerd worden.

Het voordeel van de niet-preëmptieve strategie is de efficiëntie. Er gaat minder tijd verloren omdat de werkverdeler slechts sporadisch moet tussenkomen. Het laten wachten en terug doen opstarten van taken gebeurt in feite door de taken zelf. Nadeel van dit systeem is dat men er op het programmaniveau eigenlijk expliciet rekening moet mee houden dat het maximale aantal taken dat op elk ogenblik kan uitgevoerd worden gelijk is aan het aantal processoren. Dit nadeel bestaat niet in de preëmptieve werkverdeler omdat alle taken daar ongeveer gelijkmatig uitgevoerd worden. Zij zullen wel trager uitgevoerd worden en bovendien moet de werkverdeler er ook op letten dat een taak die zich in een kritische sectie bevindt niet zomaar mag onderbroken worden. Een vervelend kenmerk van de niet-preëmptieve werkverdeler is dat een op hol geslagen taak een processor voor altijd vast zal houden. Meer details over de invloed van de strategie van de werkverdeler staan verder in dit hoofdstuk te lezen (blz. 130).

4.6 De Prestatieresultaten

De prestatie van het bord werd op twee manieren bestudeerd. Vooreerst werd de concrete implementatie van de zonet besproken algoritmen op een vierprocessormachine met gemeenschappelijk geheugen gebruikt om een aantal metingen te verrichten. De uitvoeringstijden voor de bordbewerkingen werden in variërende omstandigheden gemeten, en deze informatie werd gebruikt om de implementatie van deze bewerkingen te optimaliseren.

Daarnaast werd er ook een simulatiemodel van het bord ontwikkeld in de taal SIMSCRIPT [Rus83]. Aan de hand hiervan werd het bord bestudeerd in functie van de belasting. Door de belasting te verhogen werd er onderzocht wat de maximale doorvoercapaciteit van het bord is.

4.6.1 Metingen op het Prototype

De prestatiemetingen op het prototype werden uitgevoerd aan de hand van een aantal evaluatieprogramma's. Er werden voornamelijk twee aspecten gemeten: (i) de prestatie van het bord als geheugen of met andere woorden, als opslagruimte van gegevens en (ii) de prestatie van het bord als communicatiemedium, d.w.z. als doorgeefluik van informatie tussen taken. Het is duidelijk dat beide aspecten niet geheel onafhankelijk van elkaar kunnen gemeten worden. Beide hebben te maken met de implementatie van het bord, en de synchronisatie kan nooit volledig uitgeschakeld worden. Men kan er ten hoogste voor zorgen dat de synchronisatieprimitieven nooit hoeven te blokkeren.

```
schrijf_lees(X) :- !X, ?X, schrijf_lees.
```

Programmafragment 4.3 Evaluatie van het bord als geheugen.

Het bord als geheugen

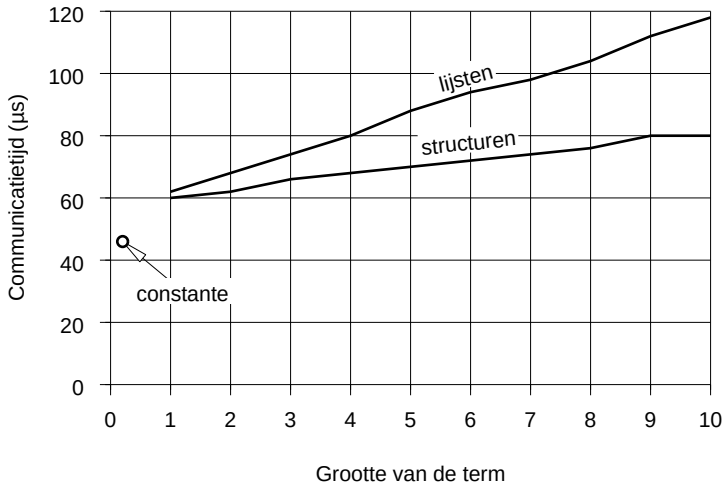
Een viertal evaluatieprogramma's wordt hier besproken. Het eerste programma onderzoekt de prestatie van de bordbewerkingen in functie van de complexiteit van de term die met het bord gecommuniceerd wordt. Hoe complexer de term, des te meer tijd er nodig zal zijn om de term daadwerkelijk te communiceren. Een tweede evaluatieprogramma onderzoekt de prestatie van de bordbewerkingen in functie van de bordinhoud. Hoe langer de bordlijsten zijn, des te meer tijd zal er nodig zijn om een gegeven met het bord te communiceren. Een derde evaluatieprogramma gaat de invloed van de parallel uitvoerende taken na. Hoe meer taken er parallel uitgevoerd worden, des te meer verstoring (vertraging) dit zal veroorzaken bij de bordcommunicatie. Een vierde en laatste evaluatieprogramma maakt het mogelijk de prestatie van het bord te vergelijken met bestaande implementaties door de uitvoeringstijden van de programma's `setof` en `bagof` te meten.

De meeste communicatietijden die in dit hoofdstuk voorgesteld worden zijn cyclustijden. Met cyclustijd wordt de tijd nodig voor het schrijvenén lezen van dezelfde term bedoeld. Wij geven er de voorkeur aan om met cyclustijden te werken in plaats van de individuele tijden voor een schrijfbewerking en een leesbewerking (i) om de hoeveelheid gegevens te beperken, en (ii) omdat er bij een echte communicatie toch altijd sprake is van een schrijfbewerking en een leesbewerking.

Evaluatieprogramma 1 De snelheid waarmee een gegeven op het bord gezet, of eraf gehaald kan worden zal onder meer afhangen van de grootte van dat gegeven. Om dit te onderzoeken werd programmafragment 4.3 gebruikt. Metingen werden gedaan voor structuren en lijsten met stijgende complexiteit. De voornaamste resultaten zijn grafisch voorgesteld in afbeelding 4.19. Als vuistregel kunnen de waarden uit tabel 4.6 gebruikt worden. Hieruit blijkt dat de communicatietijd (lezen of schrijven) voor een kleine term ongeveer rond de 30 μs ligt. Dit is in de orde van drie logische inferenties van het evaluatieprogramma `naive reverse 30`. Het feit dat cyclustijden voor lijsten ongeveer een factor drie sneller stijgen dan voor structuren is te wijten aan hun interne voorstelling. Een lijst wordt in het geheugen opgeslagen als een geketende lijst van de elementen terwijl de argumenten van een structuur contigu in het geheugen opgeslagen worden. Het copiëren van een structuur is dus eenvoudiger, en wat meer is, de lokaliteit is beter. Dit laatste zorgt ervoor dat structuren beter gebruik kunnen maken van het tussengeheugen.

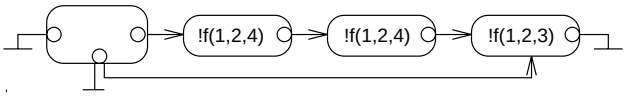
Tabel 4.6 Uitvoeringstijd per cyclus en per termtype voor het `schrijf_lees` evaluatieprogramma. De argumenten van de structuren en de lijsten zijn constanten. De parameter n is het aantal argumenten van de structuur of het aantal elementen van de lijst.

term	communicatietijd (μs)
constante	46
f(...)	$60 + 2n$
[...]	$58 + 6n$



Afbeelding 4.19 Communicatietijden met het bord per cyclus van `schrijf_lees`.

Evaluatieprogramma 2 Dit evaluatieprogramma bestudeert het gedrag van de bordprimitieven voor toenemende lengte van de lijsten. Het programmafragment 4.3 wordt opnieuw uitgevoerd, maar nu met initieel een aantal termen op het bord. De term die geschreven en nadien gelezen wordt is `f(1,2,3)`, en de termen die initieel tot vijfmaal toe op het bord geplaatst worden zijn achtereenvolgens `f(1,2,3)`, `f(1,2,4)`, `f(0,2,3)` en `g(1,2,3)`. Het lezen en schrijven van de term `f(1,2,3)` zal zeker beïnvloed worden door de inhoud van het bord zoals te zien is in afbeelding 4.20. Het resultaat van de metingen staat in tabel 4.7 en is grafisch weergegeven in afbeelding 4.21. Dit evaluatieprogramma stelt het bord op de proef omdat een nieuwe term steeds achteraan de lijst toegevoegd wordt en hierbij de arbitersemaforen moeten vergrendeld worden. Bij het lezen zal het gedrag afhangen van het aantal en de aard van de elementen op het bord. In het geval `f(1,2,3)` kan steeds het eerste element gekozen worden. In het geval `f(1,2,4)` moet de volledige lijst



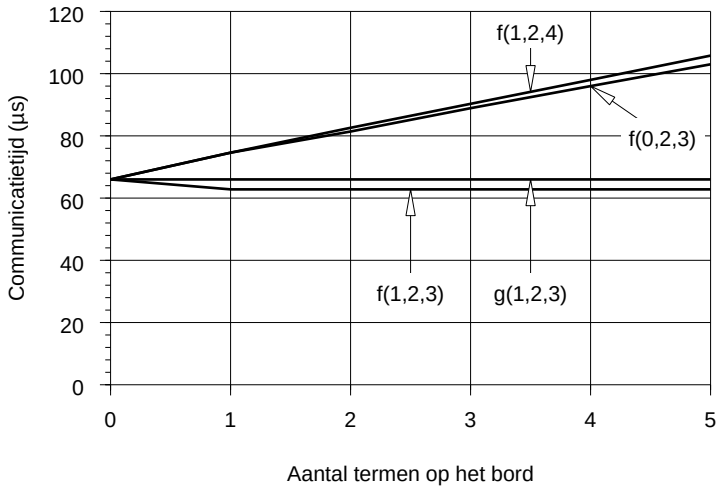
Afbeelding 4.20 Deelbord met drie structuren.

Tabel 4.7 Evolutie van de cyclustijden van `schrijf_lees` bij stijgende lijstlengte.

aantal	term			
termen	`f(1,2,3)`	`f(1,2,4)`	`f(0,2,3)`	`g(1,2,3)`
0	66.0	66.0	66.0	66.0
1	62.8	74.6	74.6	66.0
2	62.8	82.6	81.4	66.0
3	62.8	90.3	88.9	66.0
4	62.8	98.0	96.0	66.0
5	62.8	105.8	103.0	66.0

eerst overlopen worden. Hierbij moeten de termen bijna volledig geünificeerd worden omdat pas bij het derde argument zal gefaald worden. In het derde geval (`f(0,2,3)`) wordt er eerder gefaald gezien het eerste argument niet overeenstemt. In het vierde geval zitten de termen in een ander deelbord waardoor zij geen invloed meer kunnen uitoefenen. Uit tabel 4.7 blijkt dat het vergrendelen van arbitersemaforen ongeveer een 3.2 μs extra rekentijd vergt. Verder blijkt dat unificatie met `f(1,2,4)` en `f(0,2,3)` slechts een klein tijdsverschil veroorzaken. De unificatie is dus blijkbaar niet de grootste kost bij de bordbewerkingen. De kost voor de unificatie met een element van een bordlijst is ongeveer 7 μs (vergrendelen van de semafoor inbegrepen). De keuze voor actief wachtende semaforen om de elementen te beschermen is dus verantwoord. Het doen wachten van een taak en terug opstarten ervan zou ongeveer een 67 μs vergen [Wul92].

Evaluatieprogramma 3 Doordat parallelle taken uitgevoerd worden op dezelfde fysische multiprocessorhardware zullen ze elkaar ook onvermijdelijk beïnvloeden. Dit evaluatieprogramma maakt het mogelijk de verstoring die door de parallelle taken veroorzaakt wordt te meten. Het programma bestaat uit 1 tot 4 identische `schrijf_lees`-taken die termen communiceren met verschillende deelborden zoals in programmafragment 4.4 geschetst is voor twee taken (`X` en `Y` in verschillende deelborden). Afbeelding 4.22 geeft de resultaten weer voor 1 tot 4 taken die structuren met toenemende complexiteit communiceren. De vier krommen zijn de uitvoeringstijden per cyclus van `schrijf_lees` voor één tot vier simultaan uitvoerende taken. Het resultaat



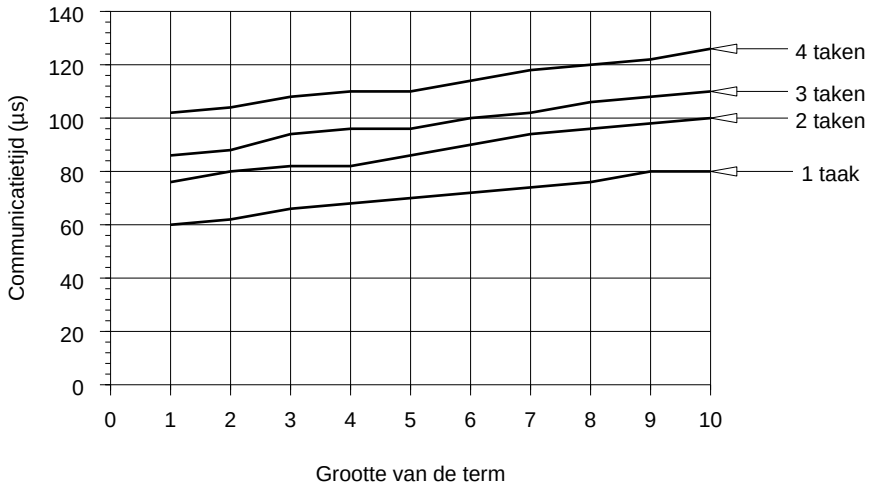
Afbeelding 4.21 Cyclustijden i.f.v. de lijstlengte.

```
schrijf_lees(X) :-
   !X, ?X, schrijf_lees(X).

dubbele_schrijf_lees(X,Y) :-
   schrijf_lees(X)&, schrijf_lees(Y)&.
```

Programmafragment 4.4 Twee concurrente `schrijf_lees` taken.

voor 1 taak is identisch met het resultaat in afbeelding 4.19.

Uit afbeelding 4.22 volgt dat er voor dit evaluatieprogramma een niet-onbelangrijke beïnvloeding uitgaat van de parallelle taken. Voor de verklaring hiervan moet er rekening gehouden worden met twee effecten. Vooreerst is er het geheugenbeheer voor het bord dat nog steeds globaal voor alle deelborden is en dus alle aanvragen sequentieel moet verwerken. In dit evaluatieprogramma is het aandeel van het geheugenbeheer in de totale uitvoeringstijd relatief groot in vergelijking met de rest van het werk dat moet gebeuren omdat de lijsten nooit meer dan 1 element lang zijn en het aflopen van de lijsten dus snel kan gebeuren. Ten tweede is er het effect van de belasting van de globale bus. Doordat deze programma's louter communiceren en voor het overige aan bijna geen verwerking doen, wordt de globale bus sterk belast (synchronisatie gaat rechtstreeks via het geheugen en de interne bordstructuur kan maar beperkt gebruik maken van de tussengeheugens omdat de bordgegevens door alle taken gebruikt worden en de coherentiehardware voortdurend moet tussenkomen). Waarden tot meer dan 10 MB/s op de M-bus werden experimenteel vastgesteld. Gezien de M-bus een maximaal debiet van 80 MB/s heeft, impli-



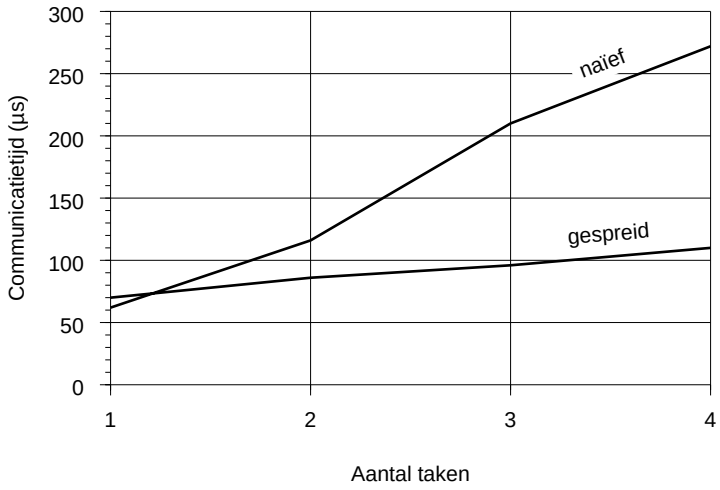
Afbeelding 4.22 Cyclustijden voor 1 tot 4 `schrijf_lees`-taken.

ceert dit dat bij vier dergelijke taken de bus voor 50% belast is en dat er dus met zekerheid wachtstaten zullen beginnen optreden die de uitvoering vertragen. Dit verklaart de eerder grote invloed van de parallelle taken in dit evaluatieprogramma. In realistische programma's zal de vertraging kleiner zijn. Dit wordt bevestigd door metingen van de belasting van de gemeenschappelijke M-bus bij niet communicerende programma's. In dat geval wordt slechts een belasting van 300 KB/s gemeten.

Dit evaluatieprogramma stelt ons ook in de mogelijkheid om de invloed van van de optimalisaties uit het begin van dit hoofdstuk uit te testen. In afbeelding 4.23 staat de invloed van het aantal taken op de cyclustijd van `schrijf_lees(f(1,2,3,4,5))` uitgezet in twee gevallen: voor het gespreide bordmodel en voor het naïeve bordmodel. Hieruit blijkt dat de uitvoeringstijd duidelijk sneller stijgt voor het naïeve bordmodel dan voor het gespreide model. Zoals verwacht ligt de prestatie van het naïeve bordmodel wel hoger voor één taak omdat er toch maar één deelbord gebruikt wordt en er minder semaforen moeten vergrendeld worden.

Evaluatieprogramma 4 Om de prestatie van deze Multi-Prologimplementatie te kunnen vergelijken met een bestaande implementatie werden `bagof` en `setof` aan de hand van het bord geïmplementeerd. De programmatekst staat in hoofdstuk 2 op blz. 28, en wordt hier gebruikt in programmafragment 4.5.

Het doel `g(X)` genereert 100 oplossingen van de vorm `f(f(N))` met `N` uniform verdeeld tussen `1` en `10`. Het programma `bagof` zal dus 100 oplossingen



Afbeelding 4.23 Effect van een gespreid bord op `schrijf_lees(f(1,2,3,4,5))`.

```
g(f(f(X))) :- tussen(_,1,10), tussen(X,1,10).

tussen(X,X,X) :- !.
tussen(X,X,_).
tussen(X,B,E) :- B<E, B1 is B+1, tussen(X,B1,E).

setof(L) :- setof(X,g(X),L).
bagof(L) :- bagof(X,g(X),L).
```

Programmafragment 4.5 `bagof` en `setof` evaluatieprogramma.

opleveren, terwijl `setof` slechts een tiental oplossingen zal weerhouden. De resultaten voor `setof(L)` en `bagof(L)` staan in tabel 4.8. Hieruit blijkt dat zowel `bagof` als `setof` sneller zijn in MULTI-PROLOG dan in SICSTUS PROLOG. Dit wijst erop dat de implementatie efficiënt is gezien er, in tegenstelling tot `assert` en `retract`[11] ook nog moet gesynchroniseerd worden tussen de verschillende taken die het bord willen gebruiken. Rekening houdend met het feit dat onze WAM-implementatie gemiddeld anderhalf keer sneller is dan de WAM in SICSTUS PROLOG (zie tabel 4.1) blijft de uitvoering van `bagof` en `setof` toch nog 20% sneller in MULTI-PROLOG dan in SICSTUS PROLOG.

De implementatie van `setof` in SICSTUS PROLOG wordt sterk vertraagd omdat alle oplossingen eerst gegenereerd worden met `bagof/3` en pas nadien gesorteerd worden waarbij alle dubbels uit de lijst verwijderd worden. Doordat in MULTI-PROLOG de dubbels eerst verwijderd worden, en de lijst pas nadien

[11] In SICSTUS PROLOG worden `setof` en `bagof` geïmplementeerd aan de hand van de ingebouwde predicaten `assert` en `retract`.

Tabel 4.8 Resultaten van `bagof` en `setof`. Het absolute snelheidsverschil tussen de machine die SICSTUS PROLOG uitvoert en één processor van de multiprocessor bedraagt een factor 4. Hiermee werd rekening gehouden bij de berekening van de verhouding.

evaluatieprogramma	MULTI-PROLOG (ms)	SICSTUS PROLOG (ms)	verhouding S/M
`bagof`	17.0	112.5	1.80
`setof`	24.2	205.8	2.13

```
ping(X,Y) :- !X, ?Y, ping(X,Y).
pong(X,Y) :- ?X, !Y, pong(X,Y).

pingpong(X,Y) :- ping(X,Y)&, pong(X,Y)&.
```

Programmafragment 4.6 Ping-pong.

Tabel 4.9 Uitvoeringstijd per cyclus en per termtype voor het `ping-pong` evaluatieprogramma. De argumenten van de structuren en lijsten zijn constanten. De parameter n is het aantal argumenten van de structuur of het aantal elementen van de lijst.

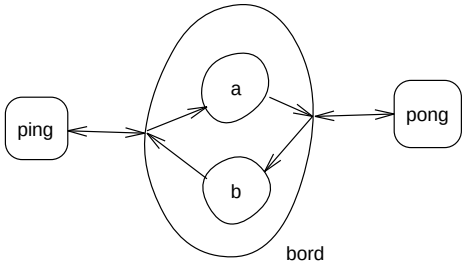
term	communicatietijd (μs)
constante	81
f(...)	82 + 2n
[...]	102 + 4n

gesorteerd wordt, is onze implementatie een stuk sneller voor het programmafragment 4.5. Om de waarheid geen geweld aan te doen is het daarom beter om enkel de resultaten voor `bagof` in rekening te brengen.

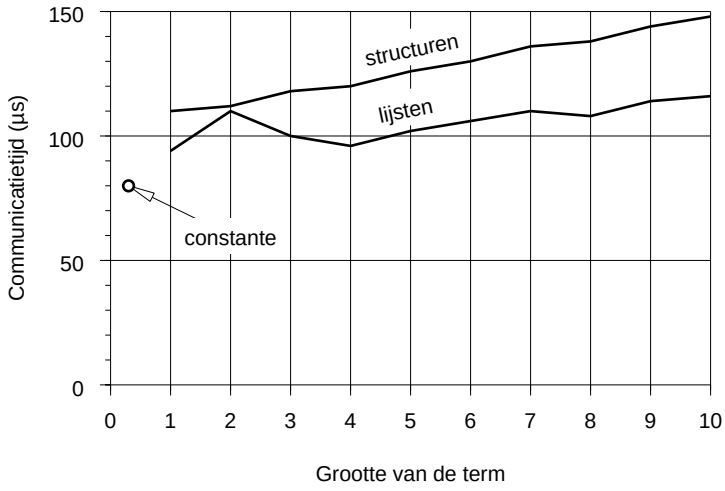
Het bord als communicatiemedium

In alle voorgaande evaluatieprogramma's werd er nooit echt gecommuniceerd tussen taken en hoefde de werkverdeler nooit tussen te komen om taken te blokkeren of terug op te starten. We gaan nu na wat er gebeurt indien dit wel het geval is.

Het `ping-pong` evaluatieprogramma uit programmafragment 4.6 kan ons een idee geven van de communicatiesnelheid tussen een producent en een consument van gegevens. De taken (`ping` en `pong`) communiceren termen heen en terug tussen elkaar. Er wordt voor gezorgd dat de termen `X` en `Y` in verschillende deelborden zitten. De communicatie met deze deelborden is zwaar: er zal ten hoogste één term of één wachtende taak zijn. Dit vereist dat de arbitersemaforen telkens opnieuw vergrendeld moeten worden. Het evaluatieprogramma wordt grafisch voorgesteld in afbeelding 4.24.



Afbeelding 4.24 Structuur van de uitvoering van `pingpong(a,b)`.



Afbeelding 4.25 BordCommunicatietijden per term voor `ping-pong`.

Zoals blijkt uit afbeelding 4.25 zijn de communicatietijden in sommige gevallen niet zeer stabiel. Dit is te wijten aan de manier van communiceren. Nadat een taak een term op het bord gezet heeft, probeert deze taak meteen het antwoord van de andere taak te lezen. Indien dit antwoord reeds voorhanden is, kan het onmiddellijk van het bord afgehaald worden. Indien de term echter nog niet aanwezig is, zal de taak geblokkeerd worden totdat de term op het bord verschijnt. Het blokkeren en nadien opnieuw wakker maken van een taak neemt relatief veel tijd in beslag. Bovendien zal tegen de tijd dat een taak effectief geblokkeerd is de term reeds op het bord aanwezig zijn zodat dit in feite een maat voor niets geweest is. Dit gedrag kan in het algemeen echter moeilijk vermeden worden omdat een leesbewerking onmogelijk kan weten wanneer een geschikte term op het bord zal verschijnen. Bij de `ping-pong`-test stelt zich

Tabel 4.10 Aantal oplossingen voor het n-koninginnenprobleem.

n	Oplossingen
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680

dit probleem echter in verhevigde mate omdat ten eerste het programma uit bijna uitsluitend communicatie bestaat en ten tweede door de sterk repetitieve natuur elke verstoring extra versterkt wordt. Dit verklaart dan ook waarom de tijden voor de structuren niet zeer stabiel zijn. De communicatietijden van het bord gebruikt als geheugen ten opzichte van de communicatietijden van het bord gebruikt als communicatiemedium zijn ongeveer 40 μs kleiner. Deze extra 40 μs zijn volledig ten laste van de interventies van de werkverdeler. Als vuistregel kunnen de waarden uit tabel 4.9 gebruikt worden.

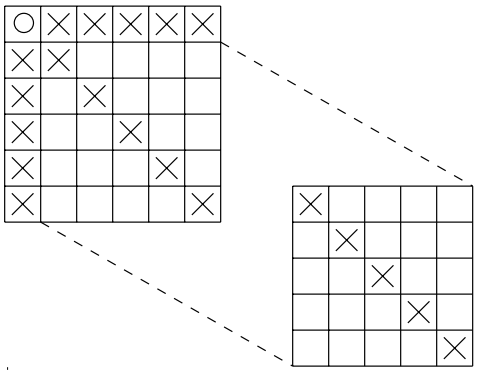
4.6.2 Metingen op een Toepassing

De metingen uit de voorgaande sectie geven een idee over de kwaliteit van de implementatie maar vertellen weinig over het gedrag van een Multi-Prologapplicatie. In deze sectie wordt het gedrag van het n-koninginnenprobleem tot in de details besproken. De vraagstelling is als volgt: plaats n koninginnen zodanig op een schaakbord met n^2 cellen dat de koninginnen elkaar niet bedreigen. Per rij, kolom of diagonaal mag er dus slechts één koningin voorkomen. Het algoritme dat gebruikt wordt om de koninginnen te plaatsen is eenvoudig. Een koningin kan enkel geplaatst worden op een plaats waar ze niet bedreigd wordt (en dus zelf niemand kan bedreigen). Telkens wanneer een extra koningin geplaatst wordt, zal het aantal nog beschikbare cellen verminderen. Indien er geen beschikbare cellen meer overblijven voor de laatste koningin moet er gezocht worden naar een alternatieve oplossing door één van de eerder geplaatste koninginnen een andere plaats te geven. Dit probleem is sterk combinatorisch van aard. Afhankelijk van de grootte van het schaakbord zullen er ofwel geen (voor kleine schaakborden van 9 cellen of minder) of meer dan één oplossing zijn (alle spiegelingen en rotaties van een oplossing zijn automatisch ook oplossingen) (zie tabel 4.10, overgenomen uit [Tic91]).

Het zopas beschreven algoritme kan op de volgende manier gebruikt worden. Gezien er maar één koningin per rij mag voorkomen wordt er begonnen

```
koningin(N) :-
  maak_schaakbord(N,B),
  plaats_koningin_op_eerste_rij(B),
  zoek_oplossing(B),
  fail.
```

Programmafragment 4.7 Sequentiële oplossing voor het n-koninginnenprobleem.



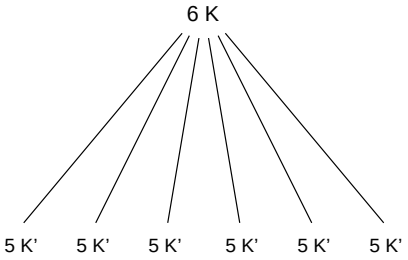
Afbeelding 4.26 Detail van het ontstaan van één pseudo-5-koninginnenprobleem uitgaande van een 6-koninginnenprobleem.

```
koningin(N) :-
  maak_schaakbord(N,B),
  plaats_koningin_op_eerste_rij(B),
  zoek_oplossing(B)&,
  fail.
```

Programmafragment 4.8 Parallelle oplossing voor het n-koninginnenprobleem.

met een koningin op de eerste rij te plaatsen. Dan wordt er overgegaan naar de tweede rij, en zo verder tot de laatste rij. De structuur van het sequentiële programma staat weergegeven in programmafragment 4.7. Het koninginnenprobleem met een aantal koninginnen vooraf op het schaakbord geplaatst wordt hier een pseudo-koninginnenprobleem genoemd. De structuur van een pseudo-koninginnenprobleem is licht verschillend van de structuur van een echt koninginnenprobleem omdat de beginvoorwaarden op het schaakbord gewijzigd zijn. Een gedetailleerde schets van een pseudo-5-koninginnenprobleem is te zien in afbeelding 4.26.

Het programmafragment 4.7 is geschreven in de vorm van een genereer-en-controleer algoritme en kan op bijna triviale manier geparallelliseerd worden. Het resultaat hiervan staat in programma 4.8.



Afbeelding 4.27 Het ontstaan van 6 pseudo-5-koninginnenproblemen uit één 6-koninginnenprobleem.

Tabel 4.11 Uitvoeringstijd voor alle oplossingen met een preëmptieve werkverdeler.

aantal koninginnen	Uitvoeringstijd (s) sequentieel	parallel	versnelling
8	1.07	0.32	3.32
9	4.58	1.34	3.42
10	20.49	5.69	3.60
11	101.02	26.79	3.77
12	534.54	137.06	3.90
13	3000.62	783.45	3.83
14	17900.16	4625.36	3.87

Het geparallelliseerde programma verdeelt de zoekboom voor een n-koninginnenprobleem op in n deelzoekbomen voor pseudo-(n-1)-koninginnenproblemen zoals geschetst in afbeelding 4.27 voor een 6-koninginnenprobleem. Elk van deze pseudo-koninginnenproblemen wordt simultaan door een individuele taak opgelost.

Om de sequentiële en parallelle programma's met elkaar te vergelijken zijn niet de absolute uitvoeringstijden, maar de versnelling door het gebruik van parallellisme belangrijk. De uitvoeringstijden voor het sequentieel en geparallelliseerd koninginnenprobleem op een multiprocessor met vier processoren staan vermeld in tabel 4.11. Dit zijn de uitvoeringstijden voor het vinden van alle oplossingen, met het gebruik van een preëmptieve werkverdeler. Uit deze tabel blijkt dat de versnelling in de buurt van het aantal processoren ligt. Bovendien stijgt de versnelling met het aantal taken. Dit is te verwachten omdat de belastingsverdeling beter zal zijn bij een groter aantal taken (en bij een langere uitvoeringstijd). Uit deze gegevens blijkt dat de overlast voor de taakcreatie en communicatie beperkt blijft tot een vijftal procent hetgeen aanvaardbaar is.

Tabel 4.12 Uitvoeringstijd voor het vinden van één oplossing met een preëmptieve werkverdeler.

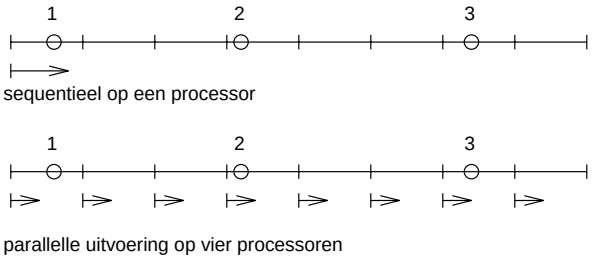
aantal koninginnen	Uitvoeringstijd (s) sequentieel	parallel	versnelling
8	0.068	0.034	1.99
9	0.035	0.040	0.89
10	0.086	0.097	0.87
11	0.061	0.068	0.88
12	0.242	0.304	0.80
13	0.133	0.152	0.88
14	1.845	1.316	1.40
15	1.473	0.885	1.66
16	10.370	1.728	**6.00**
17	6.686	1.434	**4.66**
18	48.037	22.897	2.09
19	3.551	10.374	0.34
20	253.738	19.159	**13.25**
21	11.682	24.206	0.48
22	2366.262	31.869	**74.25**
23	40.280	16.355	2.46
24	643.084	101.682	**6.32**

Tijdens de metingen voor dit evaluatieprogramma kwam er echter een eerder onverwacht fenomeen aan het licht. Zoals vermeld in het begin van deze sectie (tabel 4.10) bestaan er voor de meeste koninginnenproblemen verscheidene oplossingen. In heel wat gevallen volstaat één van deze oplossingen[12]. Door nu op zoek te gaan naar één oplossing (ongeacht welke) in plaats van alle oplossingen kan het programma termineren van zodra één oplossing gevonden wordt. In tabel 4.12 staan de resultaten voor dit nieuwe probleem. De versnellingen zijn grondig verschillend van het originele probleem. Vaak is de versnelling kleiner dan één, en voor een aantal van de problemen is de versnelling dan weer superlineair.

Om een verklaring te vinden voor dit fenomeen moet er even stilgestaan worden bij de manier waarop het geparallelliseerde programma uitgevoerd wordt.

Het programma waarvan de resultaten in tabel 4.12 vermeld staan maakt gebruik van een preëmptieve werkverdeler. Dit heeft als gevolg dat de totale rekentijd gelijkmatig verdeeld wordt over alle taken. Hoe meer taken in het systeem, des te trager de individuele taken uitgevoerd worden. Afbeelding 4.28 geeft dit grafisch weer. Oplossingen worden voorgesteld door kleine cirkeltjes.

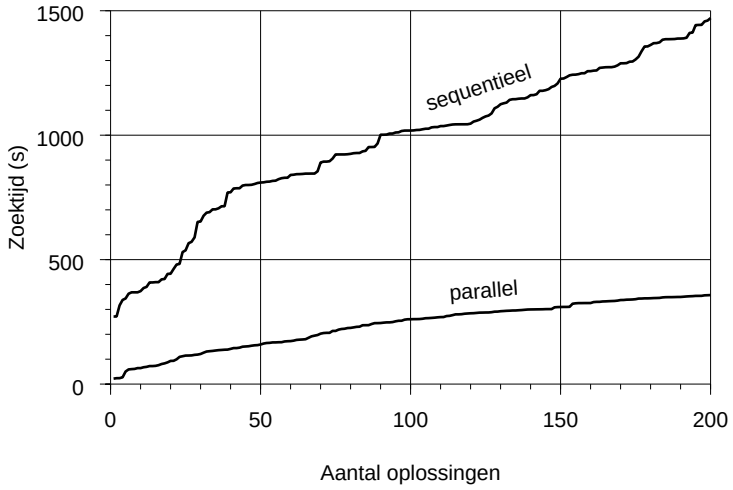
[12] Dit is de normale manier van werken bij de geëngageerde-keuzetalen.



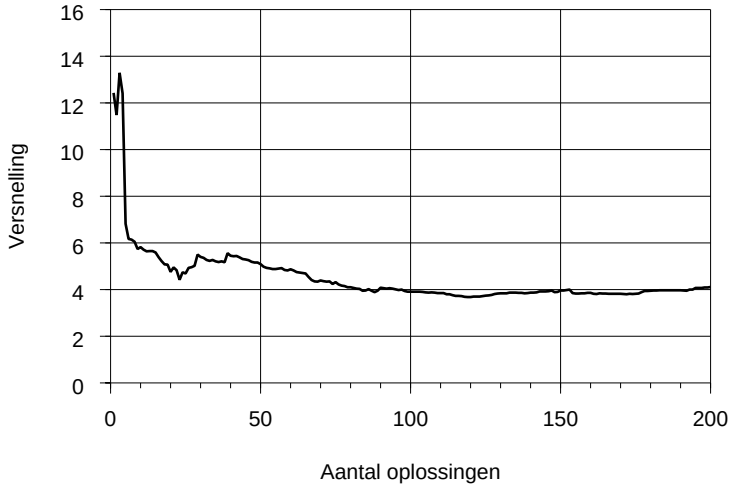
Afbeelding 4.28 Het effect van een preëmptieve werkverdeler op de parallelle uitvoering van een programma.

De pijl geeft aan hoever de evaluatie gevorderd is. In het sequentiële geval zal oplossing 1 eerst gevonden worden. Omdat in het parallelle geval de beschikbare rekentijd (die nu viermaal groter is) verdeeld moet worden over acht taken zal het nu langer duren alvorens oplossing 1 gevonden wordt. Oplossing 2 ligt echter heel wat gunstiger en wordt bij de parallelle uitvoering eerst gevonden. Men zou kunnen stellen dat de parallelle uitvoering eerder in de breedte dan in de diepte gaat zoeken.

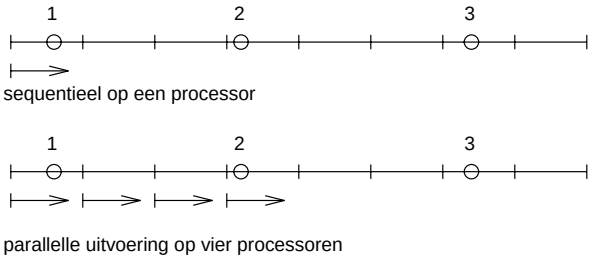
Indien de tijdstippen waarop het sequentiële programma oplossingen vindt uniform verdeeld zijn over de totale uitvoeringstijd van het programma kan men voor het parallelle programma gemiddeld een versnelling verwachten die in de buurt ligt van het aantal processoren. Merkwaardig genoeg blijken voor tal van programma's de tijdstippen waarop de oplossingen gevonden worden helemaal niet uniform over de totale uitvoeringstijd verdeeld te zijn [WDB92]. In afbeelding 4.29 staat de totale uitvoeringstijd in functie van het aantal oplossingen dat tot dan toe gevonden werd voor het 20-koninginnenprobleem. Het blijkt dat het parallelle algoritme meteen oplossingen begint te genereren, terwijl het sequentiële algoritme in het begin niet veel teken van leven geeft. Omdat het parallelle algoritme gebruik maakt van vier processoren, en het sequentiële algoritme slechts van één, is het beter niet de uitvoeringstijden, maar de versnelling te beschouwen in afbeelding 4.30. Hier blijkt dat er in het begin een voorsprong van ongeveer een factor 12 is die langzaam teloor gaat om na een honderdtal oplossingen tot ongeveer een factor vier gereduceerd te zijn. Deze niet uniforme verdeling verklaart meteen waarom in sommige gevallen werkelijk ongelooflijk hoge versnellingen gemeten worden. Ook de lage versnellingen kunnen op dezelfde manier verklaard worden. Indien de oplossingen werkelijk zeer ongunstig liggen zal na verloop van tijd toch de sequentiële oplossing gegenereerd worden, zij het dan een stuk trager dan in het sequentiële geval omdat verscheidene taken nu dezelfde processor moeten delen. Het is ook belangrijk te vermelden dat men op voorhand niet kan voor-



Afbeelding 4.29 Uitvoeringstijd i.f.v. het aantal oplossingen voor het 20-koninginnen-probleem.



Afbeelding 4.30 Versnelling voor het 20-koninginnenprobleem i.f.v. van het aantal oplossingen.



Afbeelding 4.31 Het effect van een niet-preëmptieve werkverdeler op de parallelle uitvoering van een programma.

spellen welke oplossingen er gegenereerd zullen worden en dat de oplossingen in het sequentiële en parallelle geval zeker niet dezelfde hoeven te zijn.

Voorgaande resultaten zijn enkel geldig voor een preëmptieve werkverdeler. Bij een niet-preëmptieve werkverdeler worden er ten hoogste vier taken parallel uitgevoerd en moeten de andere taken wachten. Dit heeft als gevolg dat een oplossing nooit trager kan gevonden worden dan in het sequentiële geval. Dit wordt opnieuw grafisch voorgesteld in afbeelding 4.31. Tabel 4.13 ondersteunt deze bewering. De enkele keren dat er een vertraging optreedt zijn te wijten aan de overlast voor het creëren van de parallelle taken. Uit de tabel blijkt dat de resultaten nu veel stabieler zijn. Het gebruik van parallellisme werkt nu niet echt vertragend meer, maar anderzijds is ook nagenoeg elke vorm van superlineaire versnelling verdwenen. Enkel voor het 22-koninginnenprobleem blijft er nog een superlineaire versnelling bewaard omdat de oplossingen daar blijkbaar zeer gunstig liggen voor de parallelle evaluatie. De versnelling is echter wel ongeveer een zestal keer kleiner dan bij een preëmptieve werkverdeler.

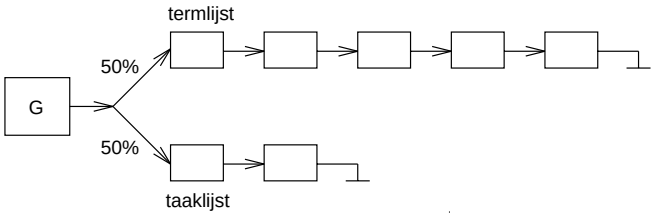
De lezer kan zich afvragen waarom er geen vergelijking gemaakt wordt met andere bordgebaseerde logische programmeertalen. De verklaring hiervoor is dubbel. Ten eerste hebben wij in de literatuur bijna geen prestatieresultaten voor implementaties van bordgebaseerde logische programmeertalen gevonden en gezien wij geen toegang hadden tot deze systemen konden we de metingen evenmin zelf uitvoeren. Indien we deze metingen wel ter beschikking zouden gehad hebben zou de vergelijking toch mank gelopen hebben omdat (i) wij geen standaardplatform gebruiken waardoor het moeilijk is absolute prestaties te vergelijken en (ii) voor zover ons bekend is wij de enige bordimplementatie voor een multiprocessor met gemeenschappelijk geheugen hebben. Het zou niet eerlijk zijn de prestaties hiervan te vergelijken met de prestatie van een bordimplementatie die gebaseerd is op het gebruik van een lokaal netwerk.

Tabel 4.13 Uitvoeringstijd voor het vinden van één oplossing met een niet-preëmptieve werkverdeler.

aantal koninginnen	Uitvoeringstijd (s) sequentieel	parallel	versnelling
8	0.0673	0.0293	2.30
9	0.0352	0.0312	1.13
10	0.0850	0.0896	0.95
11	0.0601	0.0653	0.92
12	0.2416	0.2482	0.97
13	0.1328	0.1400	0.95
14	1.8485	1.3661	1.35
15	1.4710	0.6951	2.12
16	10.5500	8.5400	1.24
17	6.6700	6.6900	1.00
18	47.3600	17.8100	2.66
19	3.5500	3.5700	0.99
20	252.6800	79.5500	3.18
21	12.4400	12.4400	1.00
22	2356.8600	188.2600	**12.52**
23	39.2800	47.3400	0.83
24	639.2300	398.9800	1.60

De vergelijking met de andere bordgebaseerde talen situeert zich dan ook meer op het vlak van de uitdrukkingskracht dan op het vlak van de prestatie (zie hiervoor hoofdstuk 2 en hoofdstuk 5).

Het enige vergelijkingspunt dat wij in de literatuur konden vinden is een vergelijkende prestatiestudie [RAN89] tussen CAGE en POLYGON, twee bordgebaseerde multi-expertsystemen die in het midden van de tachtiger jaren aan de Stanford universiteit ontwikkeld werden. Beide systemen werden ook werkelijk geïmplementeerd en geëvalueerd. Afhankelijk van de manier waarop er geparallelliseerd werd bekwam men versnellingen die varieerden van een factor 5.7 op een multiprocessor met gemeenschappelijk geheugen (16 processoren) en een factor 12 op een multiprocessor met gespreid geheugen (128 processoren). Deze versnellingen lijken niet zeer bemoedigend, maar men moet hierbij bedenken dat een bord in een multi-expertsysteem sterker gestructureerd is en dat er in het algoritme van de inferentiemotor ook nog sequentële stukken voorkomen. De versnellingen van de twee implementaties mogen ook niet vergeleken worden omdat er gebruik gemaakt werd van totaal verschillende algoritmen. Dit verklaart waarom de implementatie op de multiprocessor met gemeenschappelijk geheugen minder goed presteert. Bovendien blijkt uit de metingen voor POLYGON dat de versnelling sterk afhankelijk is van de gebruikte



Afbeelding 4.32 Blokdiagram van de simulatie.

gegevens (de toepassing dus). Dit wordt later in dit hoofdstuk bevestigd door de metingen op het simulatiemodel voor MULTI-PROLOG. Zowel CAGE als POLYGON halen nauwelijks een versnelling van een factor twee bij vier processoren. In vergelijking hiermee kan de implementatie van MULTI-PROLOG dan wel efficënt genoemd worden.

4.6.3 Metingen op het Simulatiemodel

Het simulatiemodel van het bord werd ontwikkeld om na te kunnen gaan in hoeverre het bordmodel kan uitgebreid worden met betrekking tot het aantal processoren. Het model werd opgesteld in SIMSCRIPT [Rus83], een simulatietaal voor discrete systemen. De taal SIMSCRIPT is een volwaardige programmeertaal met de klassieke gegevenstypes en controlestructuren. Daarenboven is er in de taal ook nog een aantal types en operatoren die louter voor simulatie gebruikt worden. Het bijhouden van statistische gegevens over de variabelen gebeurt min of meer automatisch.

Er werd een model gemaakt van één enkel deelbord. Deelborden interageren immers weinig met elkaar en worden daarom als onafhankelijk beschouwd in de simulatie. Voor het overige wordt het gespreide bordmodel gemodelleerd. Een blokdiagram van het simulatiemodel is weergegeven in afbeelding 4.32. Een generator G van bordgebeurtenissen genereert gebeurtenissen met een exponentiëel verdeelde tussenaankomsttijd. De gebeurtenissen worden uniform verdeeld over schrijfgebeurtenissen en leesgebeurtenissen. Gemiddeld zullen dus de ene helft van de gebeurtenissen leesgebeurtenissen zijn en de andere helft schrijfgebeurtenissen. Dit garandeert dat de lengte van de beide bordlijsten ook bij lange simulaties min of meer in evenwicht zullen blijven. Elke gebeurtenis krijgt bij zijn ontstaan een zogenaamde complexiteit mee. Deze complexiteit is de tijd die nodig is om met deze gebeurtenis te kunnen unificeren. Samenvattend heeft een gebeurtenis dus een type (lezen of schrijven) en een complexiteit.

Na zijn ontstaan zal een gebeurtenis in functie van zijn type de termlijst of de taaklijst aflopen. Deze taaklijst en termlijst bestaan uit gebeurtenissen waaraan voordien niet voldaan kon worden. Deze gebeurtenissen hebben dus ook een type (ligt vast per lijst) en een complexiteit. Bij de unificatie van een gebeurtenis met een element uit een lijst wordt er eerst gekeken naar de complexiteit van zowel de gebeurtenis als het lijstelement. De kleinste van de twee complexiteiten wordt gebruikt om de unificatie met het lijstelement door te voeren. Dit is in overeenstemming met het unificatiealgoritme. De kleinste van de twee termen zal bepalend zijn voor de complexiteit van de unificatie.

De complexiteit geeft aan hoe lang de unificatie van de twee termen duurt. Het zegt echter niets over het resultaat van de unificatie. In de simulatie nemen we aan dat twee termen unificeren met een kans die we β noemen. Dan is de kans dat er niet geünificeerd wordt gelijk aan $1-\beta$. Dit is meteen de kans dat er moet geünificeerd worden met het volgende element uit de lijst. De kans dat de unificatie daar slaagt is dan $(1-\beta)\cdot\beta$. Samen is de kans dat er met één van beide elementen met succes geünificeerd wordt $\beta+(1-\beta)\cdot\beta$. Voor een lijst met n elementen geeft dit

$$\beta+(1-\beta)\cdot\beta+\cdots+(1-\beta)^{n-1}\cdot\beta$$

hetgeen gelijk is aan

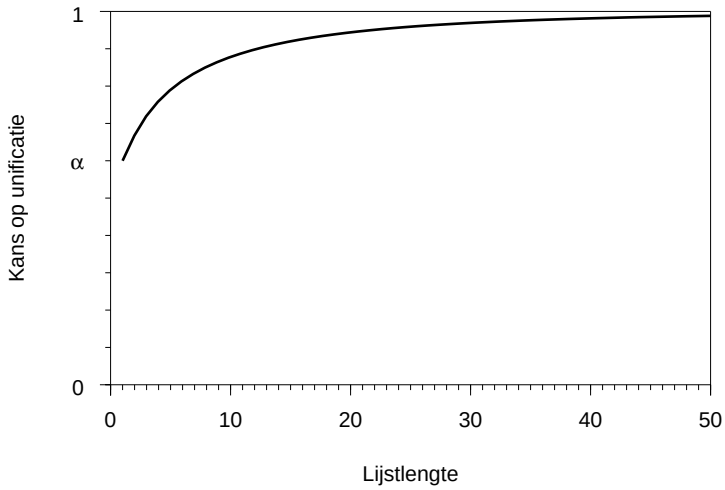
$$1-(1-\beta)^n.$$

Bij een vaste waarde voor β zal de absolute kans dat er met het laatste element van een lijst geünificeerd wordt sterk afnemen met de lengte van de lijst. Bij een β van 0.25 zal er met 95% kans geünificeerd worden met één van de tien eerste elementen van de lijst, en met 99% kans metéén van de 16 eerste elementen van de lijst. Bij lange lijsten wordt het bijna onmogelijk om met een element uit de staart van de lijst te unificeren. Daarom wordt er voorgesteld om β te laten afnemen met de lengte van de lijst. Uiteindelijk werd er gekozen voor $\beta=\frac{\alpha}{\sqrt{n}}$. Hierbij is α de kans dat er geünificeerd wordt met het enige element uit een lijst met lengte $n=1$. Hieruit volgt dat de parameter α ergens tussen 0 en 1 moet liggen. De parameter α is bepalend voor de gemiddelde lijstlengte in regime. Hoe lager de waarde van α des te groter de gemiddelde lijstlengte zal zijn.

De kans dat er met een element uit een lijst met lengte n geünificeerd wordt is dan

$$1-(1-\frac{\alpha}{\sqrt{n}})^n.$$

Deze uitdrukking gaat asymptotisch naar 1 voor stijgende n, d.w.z. voor stijgende lijstlengte (zie afbeelding 4.33). Hoe langer de lijst wordt, des te groter de kans dat een geschikte term zal gevonden worden. Voor een lijst met lengte 0 kan er niet geünificeerd worden, en voor een lijst met lengte 1 is de kans dat

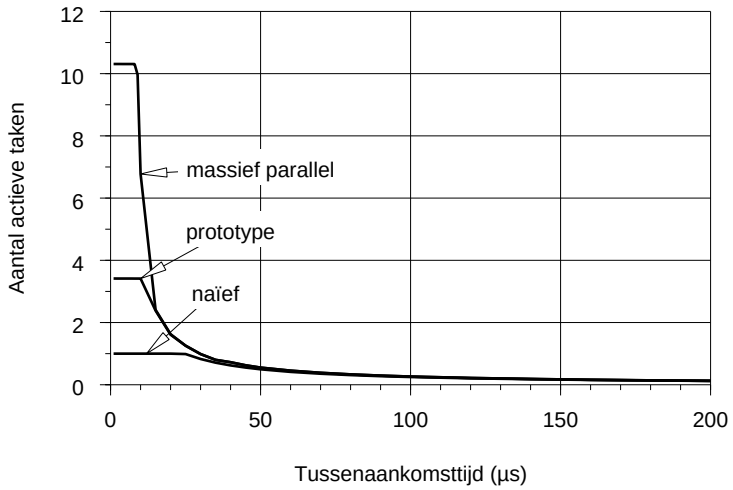


Afbeelding 4.33 Kans dat een gebeurtenis unificeert met een element uit een lijst met een gegeven lengte.

er geünificeerd kan worden gelijk aan α. In de simulaties die verder volgen werd $\alpha = 0.6$ gekozen. Deze waarde is zo gekozen dat zij aanleiding geeft tot bordlijsten met een gemiddelde lengte van ongeveer 20.

In wat volgt worden de resultaten van de simulatie voorgesteld. Alle resultaten worden uitgedrukt in functie van de gemiddelde tussenaankomsttijd. Deze gemiddelde tussenaankomsttijd is een maat voor de belasting van het systeem. Aangezien het simulatiemodel geen rekening houdt met de oorsprong van een bordgebeurtenis, vormt het een model voor zowel de intensiteit waarmeeéén taak communiceert, als voor het aantal taken dat communiceert (hoe meer taken, des te intensiever er gecommuniceerd zal worden).

In afbeelding 4.34 wordt het aantal taken dat simultaan het bord gebruikt uitgezet in functie van de gemiddelde tussenaankomsttijd. De grafiek bevat drie reeksen gegevens. De lijn die gemarkeerd is met naïef geeft het gemiddelde aantal taken weer indien het naïeve bordmodel gebruikt wordt. De lijst met markering prototype geeft het resultaat weer voor de implementatie (maximaal vier taken kunnen simultaan actief zijn op het bord gezien er slechts vier processoren zijn). De lijn met markering massief parallel geeft het aantal taken aan dat bij een onbeperkt aantal vrije processoren simultaan het bord zal gebruiken. Daar waar het aantal taken in het naïeve model beperkt wordt door het synchronisatieschema en in het prototype door het aantal beschikbare processoren, wordt het aantal taken in het massief parallelle geval beperkt door de lengte van de lijsten. Hoe langer de lijsten zijn, des te groter het parallellisme dat uitgebuit kan worden. We zien hier dat bij een gemiddelde

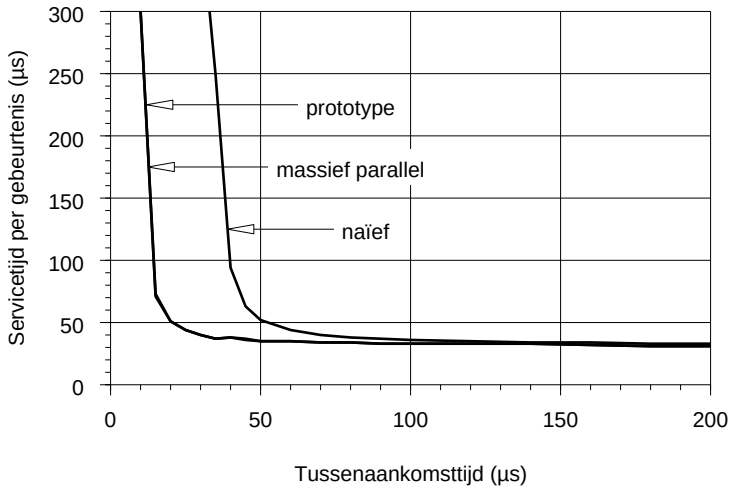


Afbeelding 4.34 Aantal simultaan actieve taken op het bord i.f.v. de gemiddelde tussenaankomsttijd.

lijstlengte van 20, er tot tien taken simultaan de lijst kunnen gebruiken.

In principe moet het aantal actieve taken nog vermenigvuldigd worden met het aantal actieve deelborden. Gezien we geen echt harde controle kunnen uitoefenen op de verdeling van de gebeurtenissen over de deelborden (zij zijn onder meer afhankelijk van de toepassing), moeten we wel voorzichtig zijn met deze extrapolatie. Met zekerheid kan echter wel gezegd worden dat het bord eerder een flessehals zal zijn voor toepassingen met korte bordlijsten dan voor toepassingen met grote bordlijsten. Dit is eigenlijk geen toeval. Het voorkomen van kleine lijsten kan wijzen op een gebrek aan parallellisme in de toepassing (een aantal taken dat staat te wachten op een lege lijst wijst erop dat de producent van gegevens niet snel genoeg kan produceren). Hoe meer elementen er zich op het bord bevinden, des te meer parallellisme er gebruikt zal kunnen worden. De hoeveelheid taakparallellisme wordt vooral bepaald door de toepassing zelf. Dit wordt bevestigd door gelijkaardige resultaten voor het bordgebaseerde multi-expertsysteem POLYGON [RAN89] waarvan de versnellingen varieerden tussen 3.6 en 11.5 voor hetzelfde programma, maar met verschillende data.

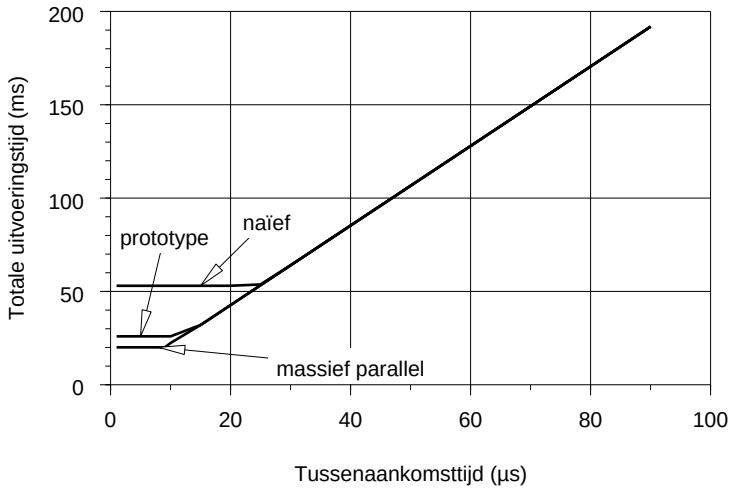
In afbeelding 4.35 staat de totale tijd die verloopt tussen het ontstaan van een gebeurtenis, en het einde van zijn verwerking in functie van de gemiddelde tussenaankomsttijd. Deze tijd bestaat uit drie componenten: (i) de wachttijd om de semafoor van het eerste element van de lijst te vergrendelen, (ii) unificatietijden met de elementen, samen met de wachttijden om naar een volgend element over te gaan, en (iii) de tijd nodig om de gebeurtenis ten uit-



Afbeelding 4.35 Verwerkingstijd per gebeurtenis i.f.v. de gemiddelde tussenaankomsttijd.

voer te brengen (invoegen in de taak- of termlijst of effectieve communicatie). Opnieuw werden de tijden uitgezet in drie gevallen. Het verschil tussen het naïeve bordmodel en het massief parallelle model is indrukwekkend. De tijden blijven beperkt tot 30 μs voor gemiddelde tussenaankomsttijden groter dan 100 μs voor het naïeve model. Voor het massief parallelle model daarentegen blijven de tijden beperkt tot 30 μs voor gemiddelde tussenaankomsttijden groter dan 30 μs. De verwerkingscapaciteit is dus heel wat toegenomen. Het prototype blijkt nagenoeg hetzelfde gedrag te vertonen als het massief parallelle model voor dit geval. Dit toont nogmaals aan dat het gespreide bordmodel wel degelijk zijn effect op de doorvoercapaciteit van het bord heeft.

Een derde en laatste grafiek schetst de totale uitvoeringstijd van het simulatieprogramma. Dit is de tijd nodig om 2000 gebeurtenissen te laten verwerken door het simulatieprogramma. Zoals blijkt uit afbeelding 4.36 is de uitvoeringstijd bij lage belasting (grote gemiddelde tussenaankomsttijd) onafhankelijk van het bordmodel en van het aantal beschikbare processoren. Doordat er zoveel tijd verstrijkt tussen twee gebeurtenissen kunnen ze gemakkelijk sequentieel verwerkt worden zonder de andere gebeurtenissen hierbij al te veel te hinderen. De totale uitvoeringstijd wordt niet in de eerste plaats bepaald door de verwerkingstijden van de gebeurtenissen, maar door de tijd die verstrijkt tussen twee gebeurtenissen. De totale uitvoeringstijd neemt dan ook lineair af met de gemiddelde tussenaankomsttijd. Het is pas als de gemiddelde tussenaankomsttijd klein wordt dat de gebeurtenissen elkaar beginnen te beïnvloeden. Het naïeve bordmodel blijft hierbij steken op ongeveer 53 ms,



Afbeelding 4.36 Totale uitvoeringstijd i.f.v. de gemiddelde tussenaankomsttijd.

dit is de tijd nodig om 2000 gebeurtenissen te verwerken aan 26.5 μs per stuk. Het gespreide en het massaal parallelle geval gaan echter nog verder tot een uitvoeringstijd van 20 ms, d.w.z. 10 μs per gebeurtenis. Dit komt natuurlijk omdat verscheidene gebeurtenissen nagenoeg simultaan kunnen verwerkt worden.

4.6.4 Uitbreiding van het Prototype

Uit de simulatieresultaten volgt dat het aantal simultane bewerkingen dat door het bord kan verwerkt worden groeit met de lengte van de bordlijsten. In de praktijk zal deze regel echter niet onbeperkt blijven opgaan omdat er effecten zullen optreden waarmee er in de simulatie geen rekening gehouden werd. De voornaamste beperking wordt gevormd door de globale bus.

Deze bus wordt in ons geval gebruikt voor het transport van zowel instructies als van gegevens. Experimenteel werd vastgesteld dat de tussengeheugens afhankelijk van het programma tussen de 90% en de 99.8% van de trafiek lokaal kunnen afhandelen (instructiedebiet en gegevensdebiet samen).

Als we ervan uitgaan dat de processoren tegen 10 MIPS werken, dan vereist dit per processor een instructiedebiet van 40 MB/s gezien de instructies 4 bytes lang zijn. Op de M-bus zal hiervan (in regime, dank zij de tussengeheugens) maximaal 10% te zien zijn, d.w.z. 4 MB/s. De M-bus heeft een maximaal debiet van 80 MB/s hetgeen overeenstemt met 20 processoren. In de praktijk zijn er dan ook tot 10 processoren geen dramatische prestatieverliezen te verwachten.

Experimenteel werd inderdaad vastgesteld dat de busbelasting voor vier

processoren kan variëren tussen 300 KB/s en 13 MB/s en dat deze busbelasting in belangrijke mate beïnvloed wordt door de hoeveelheid bordcommunicatie. De reden hiervoor ligt voor de hand: alle bordcommunicatie gaat rechtstreeks naar het geheugen, en wordt slechts een korte tijd in het tussengeheugen bijgehouden. Van zodra een taak op een andere processor een bordlijst overloopt, moeten de data eerst via de globale bus naar het geheugen geschreven worden, alvorens ze door de andere processoren kunnen gelezen worden. Dit brengt een aanzienlijke hoeveelheid extra communicatie over de M-bus met zich mee. Een voorbeeld kan dit verduidelijken.

Bij een viertal parallelle `schrijf_lees`-taken treden er vier schrijfbewerkingen en vier leesbewerkingen op per cyclus van 102 μs (zie afbeelding 4.22). Dit correspondeert met een gemiddelde tussenaankomsttijd van 13 μs of omgerekend 77 000 communicaties per seconde. Bij een busbelasting van 13 MB/s geeft dit ruim 160 bytes per communicatie. Eén communicatie vereist het vergrendelen van minstens 5 semaforen. Het vergrendelen van een semafoor bestaat uit het atomair lezen en schrijven van de nieuwe waarde van de semafoor. Bovendien wordt er per lijn geschreven. Samen geeft dit alleen al 40 bytes trafiek op de M-bus. Dit is reeds 25% van de totale M-bustrafiek per communicatie. Daarnaast komt nog de eigenlijke communicatie en de overlast die door de ware-tijdskern veroorzaakt wordt.

Gelukkig communiceren niet alle programma's even intensief waardoor er nog wat ruimte voor uitbreiding blijft bestaan. Een uitbreiding tot 10 processoren zou voor wat de prestatie betreft in principe geen probleem mogen zijn.

4.7 De Uitbreidingen

In dit hoofdstuk zijn tot nu toe enkel het schrijfpredicaat en het blokkerende niet-terugzoekende destructieve leespredicaat aan de orde geweest. In hoofdstuk 2 werden echter ook nog tal van andere predicaten voorgesteld. De aanpassingen aan het bordmodel die nodig zijn om deze predicaten ook te ondersteunen worden nu besproken.

Hierbij is het nuttig te vermelden dat nu volgende leesvarianten wel degelijk geïmplementeerd werden gezien zij tot de kerntaal behoren, maar dat noch de voorwaardelijke bordbewerkingen, noch de meervoudige borden geïmplementeerd werden in het prototype.

4.7.1 De Leesvarianten

Naast het blokkerende niet-terugzoekende destructieve leespredicaat werden de zeven andere varianten van het leespredicaat ook geïmplementeerd. De implementatie ervan vereist een aantal uitbreidingen. De acht leesbewerkin-

gen werden in het hoofdstuk over taal (zie blz. 25) besproken aan de hand van drie criteria: destructief tegenover niet-destructief, blokkerend tegenover niet-blokkerend en terugzoekend tegenover niet-terugzoekend. Wij zullen hier dezelfde criteria aanhouden om de nodige uitbreidingen te bespreken. De acht primitieven gedragen zich identisch tijdens het zoeken naar een geschikt lijst-element. Pas nadat al dan niet een element gevonden is treden de verschillen op.

Destructief tegenover Niet-destructief

Nadat een element gevonden is zal het uit de lijst verwijderd worden bij een destructieve bewerking en niet verwijderd worden bij een niet-destructieve. Beide bewerkingen eindigen nadien op precies dezelfde manier.

De schrijfbewerking moet wel licht gewijzigd worden om niet-destructief lezen toe te laten. Tot nu toe overliep de taak die de schrijfbewerking uitvoert alle wachtende taken om na te gaan of er geen stonden te wachten op de aangeboden term. Dit moet nog steeds gebeuren, maar indien een niet-destructieve leesbewerking staat te wachten, moet de term gecommuniceerd worden met de wachtende taak en nadien de taaklijst gewoon verder afgelopen worden omdat de term niet echt geconsumeerd geweest is. Pas nadat de schrijfbewerking met succes geünificeerd werd met een destructieve leesbewerking, of nadat de term opgenomen is in de termlijst, is de schrijfbewerking echt ten einde. Bovendien mogen de niet-destructieve leesbewerkingen pas echt verdergezet worden op het einde van de schrijfbewerking, nadat de term mogelijk op de termlijst geplaatst werd. De taak die de niet-destructieve leesbewerking uitvoert kan er immers van uitgaan dat na het slagen van de leesbewerking de term nog steeds op het bord zal staan. Dit kan enkel gegarandeerd worden indien de schrijfbewerking volledig uitgevoerd is.

Blokkerend tegenover Niet-blokkerend

Dit criterium geeft enkel een verschil indien er geen term gevonden wordt. Bij een niet-blokkerende variant wordt er in plaats van een wachtende taak in de taaklijst toe te voegen gewoon gefaald. Er blijven dus geen sporen over van deze niet-verwezenlijkte vraag.

Terugzoekend tegenover Niet-terugzoekend

Een terugzoekend predicaat heeft nood aan een soort van keuzepunt om bij te houden welk het laatst geselecteerde element was. Het bijhouden van dit laatste element is echter delicaat. Een wijzer naar het laatst gelezen element volstaat niet omdat dit element kan verwijderd worden door een destructieve leesbewerking nog voordat terugzoeken optreedt. In dit geval zou er dus een

corrupte wijzer ontstaan. Daarom wordt er aan alle bordtermen een uniek sequentieel nummer gegeven. Dit doet dienst als een soort van tijdstempel. In het keuzepunt wordt dan gewoon bijgehouden wat de tijdstempel van de laatst geselecteerde term was. Bij terugzoeken wordt de lijst opnieuw overlopen (unificatie is nu niet meer nodig) totdat het eerste element gevonden wordt dat recenter is. Dan kan de unificatie opnieuw beginnen. Merk op dat de tijdstempel niet ingevoerd werd om unificatie met reeds gecontroleerde elementen te vermijden, maar louter om te weten hoever men gekomen was, en om dus niet tweemaal dezelfde term te selecteren. Het feit dat er niet meer geünificeerd moet worden is natuurlijk meegenomen. Het neemt echter niet weg dat de volledige lijst opnieuw moet doorlopen worden vanaf het begin, en dat alle tussenliggende termen één na één moeten vergrendeld en weer moeten vrijgegeven worden.

Het terugzoeken veroorzaakt een lichte overlast in de bordprogrammatuur omdat bij het invoegen in de termlijst een term nu steeds een tijdstempel zal moeten krijgen. Men kan immers niet voorspellen of er al dan niet een terugzoekende leesbewerking zal gebruikt worden. Nadeel van deze tijdstempel is dat hij na gebruik geïncrementeerd moet worden[13] en dit kritisch moet gebeuren gezien verscheidene taken dit terzelfder tijd kunnen doen. Dit zou betekenen dat er nog een extra semafoor toegevoegd wordt aan het bord. Doordat de deelborden echter onafhankelijk zijn kan het bijhouden van de tijdstempel lokaal per deelbord gebeuren. Dit betekent dat men de tijdstempel kan laten beschermen door de semafoor van het eerste lijstelement. Deze semafoor wordt steeds als eerste vergrendeld (zelfs voor een lege lijst) en kan dus gebruikt worden zonder extra synchronisatiekost. Merk op dat zelfs indien een term niet fysisch in de termlijst opgenomen wordt omdat er een geschikte destructieve leesbewerking staat te wachten, er toch een tijdstempel moet toegekend worden omdat eventuele terugzoekende leesbewerkingen de tijdstempel van de term steeds nodig hebben.

Wij vervolgen nu met een overzicht van de moeilijkheden die te verwachten zijn bij de implementatie van de voorwaardelijke bordbewerkingen. Deze moeilijkheden hebben vooral te maken met de efficëntie waarmee er gecommuniceerd moet kunnen worden. Het hoofdprobleem is telkens weer dat tijdens de uitvoering van een bordbewerking dat deel van het bord dat effectief gebruikt wordt stabiel moet zijn. De gemakkelijkste oplossing, met name het vergrendelen van het totale bord, is jammer genoeg de minst efficënte.

[13] Dit kan niet oneindig lang blijven doorgaan. Wel kan het aantal bits van de teller groot genoeg gekozen worden opdat de teller geruime tijd kan blijven gebruikt worden. Zoniet moeten op een bepaald ogenblik de tijdstempels op het bord aangepast worden (b.v. door te hernummeren te beginnen bij 1).

```
!X:V :- V, !X.
```

Programmafragment 4.9 Simulatie van een voorwaardelijke schrijfbewerking.

4.7.2 De Voorwaardelijke Schrijfbewerking

De voorwaardelijke schrijfbewerking is moeilijk efficiënt te implementeren in het gespreide bordmodel. Een correcte implementatie vereist dat men het bord volledig bevriest tijdens de uitvoering van de voorwaardelijke schrijfbewerking. De evaluatie van de voorwaarde en de uitvoering van de eigenlijke schrijfbewerking moeten immers ononderbroken kunnen gebeuren. Zoniet kan men de voorwaardelijke schrijfbewerking beter implementeren door een conjunctie zoals in programmafragment 4.9. Deze implementatie is enkel correct indien de evaluatie van de voorwaarde geen communicatie met het bord veroorzaakt.

In het geval dat er wel bordcommunicatie optreedt tijdens de evaluatie van de voorwaarde V, moet in principe het complete bord (alle deelborden) bevroren worden. Dit is echter een bijzonder tijdrovende bewerking[14]. Indien tijdens de compilatie kan uitgemaakt worden welke deelborden er gebruikt zullen worden tijdens de uitvoering van de voorwaarde, dan kan men zich wel beperken tot enkel het bevriezen van die deelborden. Een andere mogelijkheid is om enkel vlakke voorwaarden toe te laten (enkel ingebouwde predicaten, en hiertoe behoren ook de bordpredicaten). De uitdrukkingskracht wordt hierdoor beperkt, maar de implementatie kan wel efficiënter gebeuren omdat in bepaalde gevallen enkel die deelborden die echt gebruikt zullen worden bevroren kunnen worden. Het bevriezen van alle deelborden die dit vereisen moet vóór de uitvoering van schrijfbewerking gebeuren om vastlopen te vermijden.

4.7.3 De Voorwaardelijke Leesbewerking

Analoog met de voorwaardelijke schrijfbewerking is ook hier de implementatie het hoofdprobleem. Tijdens de uitvoering van een voorwaardelijke leesbewerking moet in principe opnieuw het totale bord bevroren worden. De implementatie van de voorwaardelijke leesbewerking is zo mogelijk nog moeilijker dan de voorwaardelijke schrijfbewerking omdat zelfs de meeste eenvoudige gevallen niet in de kerntaal van MULTI-PROLOG kunnen gesimuleerd worden.

Net zoals bij de voorwaardelijke schrijfbewerking dienen er zich twee mogelijkheden aan. Indien men algemene voorwaarden toelaat moet het volledige bord (alle deelborden) bevroren worden alvorens de bewerking uit te voeren omdat het bord tijdens de uitvoering van de bewerking gegarandeerd stabiel moet zijn. Indien men vlakheid oplegt aan de voorwaarde, dan kan men zich

[14] Zie ook de discussie over de implementatie van ?X op blz. 108.

in bepaalde gevallen beperken tot het bevriezen van enkel die deelborden die gebruikt zullen worden tijdens de communicatie.

4.7.4 Meervoudige Borden

De implementatie van de meervoudige borden schept geen echte problemen voor de prestaties van de bordbewerkingen. De prestaties zullen enkel beïnvloed worden door het feit dat een extra parameter nodig is bij elke bewerking en door het feit dat er nu telkens moet gecontroleerd worden of een bord wel bestaat (dit is nu niet meer vanzelfsprekend).

Een belangrijk implementatieprobleem is de verdeling van het geheugen over al de borden. Ideaal zou elk bord zijn eigen geheugenbeheer moeten kunnen doen. Het centraliseren van het geheugenbeheer zou anders wel eens een flessehals kunnen vormen. Het decentraliseren van het geheugenbeheer vereist ook dat van in het begin voldoende geheugen aan de nieuwe borden wordt gegeven. Doordat alle borden gelijkwaardig zijn zouden ze ook allemaal even groot moeten zijn als het moederbord. Deze verspilling van geheugen is echter niet te verantwoorden. Twee oplossingen dienen zich aan. De eerste oplossing bestaat erin het geheugenbeheer centraal te bewaren, eventueel hiërarchisch. Voor een kleine configuratie met vier processoren zoals in de prototype-implementatie zal dat geen groot prestatieverlies veroorzaken. Een tweede oplossing is meer pragmatisch, maar minder elegant. Bij de creatie van een bord zou men een indicatie kunnen meegeven over de maximale grootte van dit bord (b.v. uitgedrukt in het aantal termen). Deze oplossing zal wel aanleiding geven tot een krachtiger implementatie.

4.7.5 Uitbreiding naar Andere Platformen

In dit hoofdstuk werd aangetoond dat de kerntaal van MULTI-PROLOG efficiënt geïmplementeerd kan worden op een multiprocessor met gemeenschappelijk geheugen, en dat de volledige taal ook implementeerbaar is, zij het minder efficiënt.

In deze sectie gaan wij na of MULTI-PROLOG ook op andere platformen kan geïmplementeerd worden, en wat de gevolgen hiervan zijn. Twee platformen zullen bekeken worden: we beginnen met een multiprocessor met gespreid geheugen en gaan verder met UNIX. Er zij terloops op gewezen dat de andere bordgebaseerde logische programmeertalen meteen op dergelijke platformen gerealiseerd werden.

Multiprocessor met Gespreid Geheugen

We beperken ons hier tot de implementatie van het bord, en bekommeren ons niet over de manier waarop taken gecreëerd en over het netwerk van proces-

soren verdeeld worden. We nemen aan dat er daarvoor voldoende faciliteiten op de multiprocessor aanwezig zijn.

Wat de implementatie van lokale borden betreft is er geen probleem. Zij interageren niet rechtstreeks en kunnen onafhankelijk van elkaar afgebeeld worden op de verschillende processoren. In het geheugenbeheer zal dan wel lokaal moeten voorzien worden. De fysische plaats van een bord kan bij de creatie samen met de identiteit van het bord opgeslagen en gecommuniceerd worden.

Indien het bord dan gemanipuleerd moet worden door taken die niet lokaal door dezelfde processor uitgevoerd worden zullen de bordbewerkingen die eigenlijk procedureoproepen zijn RPC-oproepen worden. Om deze RPC-oproepen goed te kunnen verwerken moet het bord verrijkt worden met een RPC-bediener. Deze bedienertaak kan dan de gewone bordmanipulatieroutines gebruiken en zal bij voorkeur meerdradig geïmplementeerd worden om meerdere aanvragen simultaan te kunnen verwerken en om toe te laten dat bepaalde aanvragen blokkeren op het bord. De klassieke bordmanipulatieroutines kunnen blijven gebruikt worden door alle taken die lokaal uitgevoerd worden. In [Pin91a] wordt een dergelijke oplossing gesuggereerd voor LINDA. De bedienertaak wordt een *veeltallenbediener* genoemd en aanvragen worden via het XDR-protocol tussen platformen overdraagbaar gemaakt.

Het verdelen van deelborden over een aantal processoren is een groter probleem omdat één logisch bord dan opgesplitst wordt in verscheidene fysisch gescheiden deelborden die onzichtbaar moeten blijven in de taal. De identificatie van het bord volstaat in dat geval niet om te weten waar een bepaald deelbord zich bevindt. Een mogelijke oplossing is het gebruik van een bordbediener die zich naar de toepassing toe gedraagt als een (virtueel) bord en die intern de verdeling van de borden over het netwerk van processoren bijhoudt en bewerkingen gewoon doorstuurt naar de processor waar een bepaald deelbord gelegen is (dit kan natuurlijk ook lokaal zijn). Dit schema wordt ook in Linda-implementaties gebruikt [Pin91a] in de vorm van een zogenaamde typebediener. Deze typebediener gaat autonoom op zoek naar een niet gekende veeltallenruimte en houdt het adres van deze ruimte bij voor gebruik in de toekomst. Een alternatieve implementatie (X-LINDA) wordt beschreven in [Faa91]. Hier worden de veeltallenruimten niet éénmaal, maar meermaals opgeslagen in een netwerk van processoren. De schaalbaarheid van dit schema is goed, maar wel ten koste van een aanzienlijk toegenomen communicatiekost.

Uit dit overzicht blijkt dat het verdelen van borden over een netwerk van processoren niet echt een probleem schept op voorwaarde dat de borden voorzien worden van een bedienertaak die bordbewerkingen die van niet-lokale processoren afkomstig zijn kan verwerken en dat er 'virtuele borden' gecreëerd worden op alle processoren. Het spreekt natuurlijk voor zich dat de prestatie

van de bordbewerkingen hieronder zal lijden, maar dit is de prijs die moet betaald worden om een gespreide implementatie te realiseren. Bovendien zal de overlast vaak kunnen beperkt worden door taken die intensief met elkaar communiceren door dezelfde processor te laten uitvoeren en het bord op dezelfde processor af te beelden. Hierdoor wordt een groot deel van de communicatie lokaal.

De implementatie van de voorwaardelijke bordbewerkingen is minder kritisch op een multiprocessor met gespreid geheugen dan op een multiprocessor met gemeenschappelijk geheugen. De reden hiervoor is dat de communicatie met een bord dat op een niet-lokale processor afgebeeld is toch al minder efficiënt gebeurt en dat de relatieve kost van het vergrendelen van het bord hierdoor kleiner wordt. Anderzijds moet men er wel voor zorgen dat de processor die de voorwaardelijke bewerking uitvoert wel degelijk toegang heeft tot de systeemmiddelen die nodig zijn om de voorwaarde te kunnen evalueren (de bepalingen, en eventueel ook de borden waarmee er gecommuniceerd moet worden). Indien een voorwaardelijke bewerking communiceert met verscheidene borden die op verschillende processoren zijn afgebeeld, dan valt er wel een aanzienlijk prestatieverlies te verwachten.

Unix

De huidige implementatie is gebaseerd op de ware-tijdskern MTOS-UX omdat deze voldeed aan onze noden, en omdat deze ter beschikking was. Van in het begin was echter duidelijk dat deze keuze de overdraagbaarheid naar andere machines fel zou beperken.

De voor de hand liggende oplossing voor dit probleem van overdraagbaarheid is de keuze voor een meerdradig uitvoeringspakket dat lichtgewicht taken aanbiedt zoals μSYSTEM [BS90] of MACH [TRG$^+$]. Een dergelijk pakket voorziet in 'lichtgewichtparallellisme' binnenin een UNIX-taak. Deze lichtgewichttaken worden door een speciale werkverdeler (niet de werkverdeler van UNIX) behandeld en zijn efficiënter dan de UNIX-taken omdat de context van de taak beperkter is. Bovendien heeft men een betere controle over de soort van strategie die door de werkverdeler gebruikt wordt.

Samenvatting

In dit hoofdstuk werd een overzicht gegeven van de belangrijkste aspecten die bij de implementatie van de taal MULTI-PROLOG zijn komen kijken. Drie bordmodellen werden voorgesteld en geïmplementeerd. Enkel het naïeve model was in staat de volledige semantiek van de taal te ondersteunen. Het gepartitioneerde bordmodel vereiste min of meer dat de communicatie van individuele vrije variabelen moest uitgesloten worden. Dit werd evenwel niet als een groot

verlies ervaren gezien de voordelen qua efficëntie die uit dit compromis voortvloeiden. Het gespreide bordmodel is het meest complexe, maar tevens ook het meest krachtige. Uit simulatieresultaten volgt dat het bord in staat is een stuk mee te groeien met de toepassing. Hieruit kan besloten worden dat bordcommunicatie op realistische multiprocessoren met gemeenschappelijk geheugen niet echt de beperkende factor zal zijn. Bovendien kan bordcommunicatie ook geïmplementeerd worden op andere platformen.

5 Toepassingen

Beware bugs in the above code.
I have only proved it correct, not tried it.

— DONALD KNUTH

In dit hoofdstuk wordt een aantal standaardvoorbeelden uitgewerkt. De meeste van deze voorbeelden zijn ontleend aan de literatuur over parallelle logische programmeertalen. Dit overzicht is echter verre van volledig. Er wordt begonnen met een aantal elementaire voorbeelden zoals de implementatie van een semafoor, en er wordt geëindigd met een paar meer uitgewerkte voorbeelden zoals een vluchtreservatiesysteem en het welbekende probleem van de dinerende filosofen. Belangrijk om hier te vermelden is wel dat de hier beschreven programma's uitgevoerd werden op de prototype implementatie van de kerntaal. Dit vereist dat de bordgebeurtenissen ?X *en* !X *niet gebruikt worden*[1]*, evenmin als de voorwaardelijke primitieven. Dit werd echter niet als hinderlijk ervaren.*

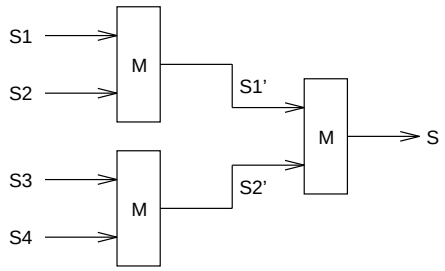
5.1 Semafoor

Semaforen zijn belangrijke en efficiënte synchronisatieprimitieven om een kritische sectie af te bakenen. Zij kunnen eenvoudig geïmplementeerd worden in MULTI-PROLOG zoals geschetst in programmafragment 5.1. Een semafoor is als een sleutel die toegang geeft tot de kritische sectie. Normaal bevindt de sleutel zich op het bord (initieel daar geplaatst door de `initialiseer` bepaling). Een p-bewerking zal de sleutel daar weghalen, indien aanwezig. Indien hij niet aanwezig is, dan zal de p-bewerking blokkeren totdat hij beschikbaar komt. De v-bewerking plaatst de sleutel terug op het bord zodat hij door iemand anders kan gebruikt worden. Het is duidelijk dat het aantal sleutels niet totéén

[1] Dit in tegenstelling met de voorbeelden die voorkomen in het hoofdstuk over de taaldefinitie die wel gebruik maken van ?X en !X.

```
p(S) :- ?S.
v(S) :- !S.
initialiseer(S) :- !S.
```

Programmafragment 5.1 Semafoor.



Afbeelding 5.1 Netwerk van mengertaken.

beperkt hoeft te zijn. Hoe meer sleutels, des te meer taken de kritische sectie simultaan kunnen betreden.

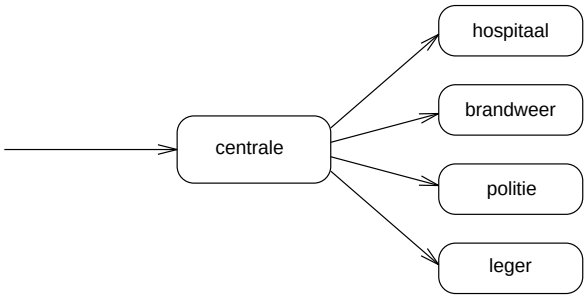
5.2 Menger

In de stroomgebaseerde parallelle logische programmeertalen gebruikt men vaak een netwerk van mengertaken om stromen samen te voegen [Sha87b] zoals geschetst in afbeelding 5.1. Dit mechanisme wordt algemeen als noodzakelijk aanzien, maar is niet elegant omdat (i) het als dusdanig niets met het probleem te maken heeft (het wordt enkel noodzakelijk gemaakt omdat die talen in tegenstelling tot MULTI-PROLOG geen ingebouwde mengfaciliteit bevatten), en (ii) het gedrag van een mengertaak kan afhangen van de gebruikte werkverdeler [New90]. Deze afhankelijkheid van de werkverdeler kan als volgt uitgelegd worden.

Als gevolg van de gebruikte implementatietechnieken zijn de meeste parallelle Prologvertolkers doorgaans vrij stabiel [Sha87b]. Dit wil zeggen dat de meeste vertolkers als gevolg van de strategie van de werkverdeler de eerste (van boven naar beneden) bepaling die met succes kan geëvalueerd worden, zal selecteren. Indien een knip of geëngageerde keuze gebruikt wordt om alternatieven te elimineren heeft dit als gevolg dat de mengerroutine eerst alle elementen uit de eerste stroom zal lezen, en slechts indien de eerste stroom geen elementen meer bevat, de tweede stroom zal beginnen te verwerken. Dit gedrag is niet billijk omdat de eerste stroom duidelijk een hogere prioriteit heeft. Dit probleem kan opgelost worden door de mengerroutine intern de prioriteiten van de inkomende stromen geregeld te laten omwisselen.

```
producent :- genereer(X), !oplossing(X), fail.

consument :- ?oplossing(X), verwerk(X), consument.

?- producent&, producent&, producent&, producent&,
   consument.
```

Programmafragment 5.2 Menger.



Afbeelding 5.2 Centrale sequentiële demultiplexer.

In MULTI-PROLOG stelt dit probleem zich niet omdat het bord automatisch de bordgebeurtenissen van alle taken samenvoegt totéén logische stroom en zich hierdoor als menger gedraagt. De volgorde is de volgorde van aankomst en deze is automatisch billijk. Programmafragment 5.2 mengt vier informatiestromen.

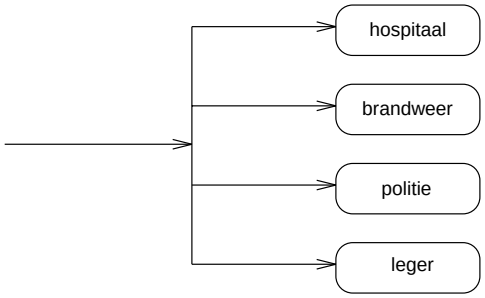
5.3 Demultiplexer

Dit is het duale probleem van de menger. Het wordt zelden behandeld in de literatuur, maar is wel interessant. Een binnenkomende stroom gegevens wordt centraal ontvangen en doorgestuurd naar de desbetreffende taken zoals geschetst in afbeelding 5.2. Het programmafragment 5.3 dat hiermee correspondeert heeft inderdaad een `centrale` en start taken op voor de verwerking van de aanvragen.

In MULTI-PROLOG bestaat er een efficiëntere oplossing. De diensten die instaan voor noodhulp kunnen ook zelf luisteren of er een nood te lenigen is, en autonoom ingrijpen wanneer dit nodig mocht blijken. Dit is schematisch voorgesteld in afbeelding 5.3. In tegenstelling met programmafragment 5.3 waar alle noodoproepen verwerkt worden door een centrale taak, worden in programmafragment 5.4 de noodoproepen door alle taken simultaan verwerkt. Dit vormt een illustratie van wat in [Gel89] een gespreide gegevensstructuur genoemd wordt.

```
centrale :-
  ?noodoproep(Type,X),
  verwerk(Type,X),
  centrale.

verwerk(ongeval,X) :- hospitaal(X)&.
verwerk(brand,X) :- brandweer(X)&.
verwerk(overval,X) :- politie(X)&.
verwerk(ramp,X) :- leger(X)&.
```

Programmafragment 5.3 Centrale demultiplexer.



Afbeelding 5.3 Gedecentraliseerde demultiplexer.

Het bijbehorende programmafragment 5.4 weerspiegelt nagenoeg exact dit schema. Eventueel moet het aantal simultane hulpdiensttaken beperkt worden met een telsemafoor zoals geschetst in programmafragment 5.5. Hoe meer `hulpdienst`-constanten er zich initieel op het bord bevinden, des te meer `hulpdienst`-taken er parallel kunnen uitgevoerd worden. In MULTI-PROLOG kan men dergelijke beperkingen op elegante en eenvoudige manier inbouwen.

5.4 Buffer met Beperkte Capaciteit

Een buffer met beperkte capaciteit is een noodzakelijk kwaad in realistische asynchrone systemen [TF85]. Er moet immers verhinderd kunnen worden dat een op hol geslagen taak het complete bord in een minimum van tijd zou volschrijven en hierdoor problemen zou veroorzaken voor de andere taken. Een dergelijk buffer zorgt ervoor dat een taak ook tijdens het schrijven van een gegeven op het bord kan blokkeren.

In CONCLOG [Jac91] wordt er een stroomgebaseerde oplossing gesuggereerd. Het gedrag van het buffer wordt beschreven aan de hand van een procedure bestaande uit zes bepalingen met acht argumenten. Het is merkwaardig tot welke complexe programmatekst deze toch relatief eenvoudige functionaliteit aanleiding geeft.

```
hospitaal :-
  ?noodoproep(ongeval,X),
  hospitaal(X)&,
  hospitaal.

brandweer :-
  ?noodoproep(brand,X),
  brandweer(X)&,
  brandweer.

politie :-
  ?noodoproep(overval,X),
  politie(X)&,
  politie.

leger :-
  ?noodoproep(ramp,X),
  leger(X)&,
  leger.
```

Programmafragment 5.4 Gespreid demultiplexerprogramma.

```
hulpdienst(X) :-
  ?hulpdienst,
  roep_hulpdienst(X),
  !hulpdienst.
```

Programmafragment 5.5 Hulpdienst met beperkingen.

In de RELATIONAL LANGUAGE [CG81] wordt een alternatieve methode voorgesteld, gebaseerd op de uitbreiding van de syntaxis van de taal. Naast de modedeclaraties ↑ voor de vrije variabelen en ? voor de niet-vrije variabelen, voorziet men ook in een constructie $\uparrow n$ om een communicatiekanaal met maximale lengte n te maken. Dit is wellicht een pragmatische oplossing, maar niet elegant gezien er een speciale constructie aan te pas komt.

In PARLOG [CG86b] en GHC [Ued85] maakt men gebruik van een buffertaak die het aantal vrije buffertjes bijhoudt en alle verkeer tussen de zender en de ontvanger regelt. Deze oplossing is eleganter te programmeren dan de beide vorige, maar vereist steeds de interventie van een speciale taak en dus van de werkverdeler.

De oplossing in MULTI-PROLOG zoals geschetst in programmafragment 5.6 is niet ideaal, maar wel eenvoudig te begrijpen en te implementeren. Een buffer is een rij van termen op het bord. Bij voorkeur zal ervoor gekozen worden deze termen een gemeenschappelijke structuur te geven zoals bijvoorbeeld `naam/1`. De capaciteit van het buffer wordt vastgelegd door een aantal *pasjes* op het bord te plaatsen. Vooraleer een term in het buffer te stoppen (d.w.z. op het bord te zetten), wordt er eerst een pasje van het bord gehaald. Enkel indien dit mogelijk is, wordt het gegeven in het buffer gestopt. Nadat een element

```
maakbuffer(0).
maakbuffer(N) :-
  !pasje,
  N1 is N-1,
  maakbuffer(N1).

schrijfbuffer(X) :-
  ?pasje,
  !buffer(X).

leesbuffer(X) :-
  ?buffer(X),
  !pasje.
```

Programmafragment 5.6 Buffer met beperkte capaciteit.

uit het buffer geconsumeerd werd, wordt het pasje teruggegeven. Van zodra de pasjes opgebruikt zijn, zal het `schrijfbuffer`-predicaat moeten wachten totdat er terug een pasje beschikbaar komt. Deze oplossing vereist twee communicaties per te communiceren gegeven. Speciale primitieven kunnen dit zeker versnellen, maar hebben andere nadelen. Ofwel creëert men primitieven die enkel voor dit soort buffers gebruikt worden, waardoor de taal complexer wordt, ofwel geeft men aan alle communicatiestromen een maximale lengte. Communicatiestromen zonder beperking krijgen dan een speciale waarde (b.v. -1). Nadeel van deze aanpak is dat de communicatie onnodig vertraagd wordt in het geval er geen capaciteitsbeperking is. De Multi-Prologaanpak is een compromis: geen extra primitieven en ook geen vertraging van de standaard communicatieprimitieven.

5.5 Synchrone Communicatie

Bordcommunicatie is asynchroon. Dit impliceert dat gegevens een tijdlang op het bord kunnen verblijven alvorens geconsumeerd te worden. Een buffer met lengte 0 zal ervoor zorgen dat een gegeven niet kan gecommuniceerd worden [CG81] tenzij de ontvanger klaar staat. Het buffer heeft immers geen capaciteit. Een dergelijke synchrone communicatie of *afspraak* behoeft opnieuw geen speciale primitieven, maar kan geïmplementeerd worden aan de hand van de standaardprimitieven. De zender in programmafragment 5.7 zal de gegevens niet op het bord kunnen plaatsen zolang de ontvanger niet in staat is ze te ontvangen. De beschikbaarheid van de ontvanger wordt aan de zender gesignaleerd door middel van de constante `klaar` op het bord. Deze constante doet dienst als het enig pasje uit het vorige voorbeeld.

```
zend(X) :-
  ?klaar,
  !gegeven(X).

ontvang(X) :-
  !klaar,
  ?gegeven(X).
```

Programmafragment 5.7 Synchrone communicatie.

```
even(N) :-
   ?nieuw_getal,
   !getal(N),
   N1 is N + 2,
   even(N1).

?- even(0)&.
```

Programmafragment 5.8 Uitgestelde evaluatie.

5.6 Uitgestelde Evaluatie

Uitgestelde evaluatie betekent dat een producent niet zomaar gegevens genereert in het wilde weg, maar op aanvraag [CG86b]. Die vraag naar gegevens wordt gegenereerd door één van de consumenten. Merk op dat de producent niet echt hoeft te wachten op de aanvraag om een gegeven te produceren. Het gegeven kan reeds geproduceerd worden, maar er wordt gewacht met de communicatie. Van zodra er een vraag naar een dergelijk gegeven komt, hoeft het enkel maar gecommuniceerd te worden en kan het volgende gegeven geproduceerd worden. Een voorbeeld zoals dat van programmafragment 5.8 met een even-getallengenerator kan dit verduidelijken. Een getal zal pas op het bord geplaatst worden nadat een constante `nieuw_getal` op het bord verschenen is. Merk op dat de getallen op een synchrone manier gecommuniceerd worden en dat verscheidene taken simultaan naar een getal kunnen vragen.

5.7 Vluchtreservatiesysteem

Dit voorbeeld werd ontleend aan [Sha87b, BC91, Jac91]. Het probleem wordt als volgt geformuleerd.

Een vluchtreservatiesysteem moet de informatie over een aantal vluchten bevatten. Het moet aanvragen voor de reservatie van een aantal zetels kunnen verwerken. Aanvragen worden door terminals gegenereerd op willekeurige ogenblikken. Het reservatiesysteem behandelt de aanvragen in volgorde van aankomst.

In zowel CONCURRENT PROLOG als in CONCLOG wordt alle informatie in verband met de vluchten in een centrale gegevensbank bijgehouden. Alle aan-

vragen voor reservaties en dergelijke worden dan sequentieel door deze gegevensbank verwerkt. Deze oplossing creëert evenwel een onnodige flessehals. In de Multi-Prologoplossing wordt de informatie over een bepaalde vlucht bijgehouden in een vluchttaak. Deze taak verwerkt enkel de aanvragen vooréén specifieke vlucht. Dit verhoogt duidelijk het aantal aanvragen dat simultaan verwerkt kan worden (cfr. de demultiplexer op blz. 155). Initieel wordt er per beschikbare vlucht een taak gecreëerd en een term met de naam van de vlucht op het bord geplaatst (`vlucht/1`). Deze laatste wordt door de terminaltaak gebruikt om te verifiëren of een bepaalde vlucht wel bestaat.

Elke individuele terminal in programmafragment 5.9 is een taak die een aanvraag ontvangt, doorstuurt naar de betreffende vlucht, en het antwoord afwacht. Elke vlucht is een taak die bijhoudt hoeveel vrije plaatsen er nog beschikbaar zijn, en de aanvragen van de terminals beantwoordt in volgorde van ontvangst. Een vlucht kan zich in twee toestanden bevinden `inscheping` en `vertrokken`. De overgang van de eerste naar de tweede toestand gebeurt niet als gevolg van een commando van een terminal, maar zal veroorzaakt worden door een commando om te vertrekken van bijvoorbeeld de controletoren. Dit commando kan mogelijk ook door een geassocieerde wekkertaak gegenereerd worden.

De oplossing in SHARED PROLOG [BC89] plaatst de informatie over de vluchten als termen op het bord. De inhoud van deze termen wordt aangepast door een centrale taak. Er is slechts één dergelijke taak voorhanden. De oplossing in MULTI-PROLOG is flexibeler omdat de aanvragen verwerkt worden door de vluchten zelf (een centrale taak kan een flessehals vormen).

Het opvoeren van het aantal centrale taken in SHARED PROLOG is mogelijk, maar vereist dan wel dat deze taken onderling gesynchroniseerd worden omdat ze dezelfde gegevens manipuleren. Het gebruiken vanéén taak per vlucht is moeilijk in SHARED PROLOG omdat er geen mogelijkheid bestaat om dynamisch taken te creëren. Door statisch een aantal (lege) vluchttaken te creëren en nadien de werkelijke vluchten hieraan toe te kennen kan dit gedrag gesimuleerd worden, maar echt elegant is dit niet.

5.8 Dinerende Filosofen

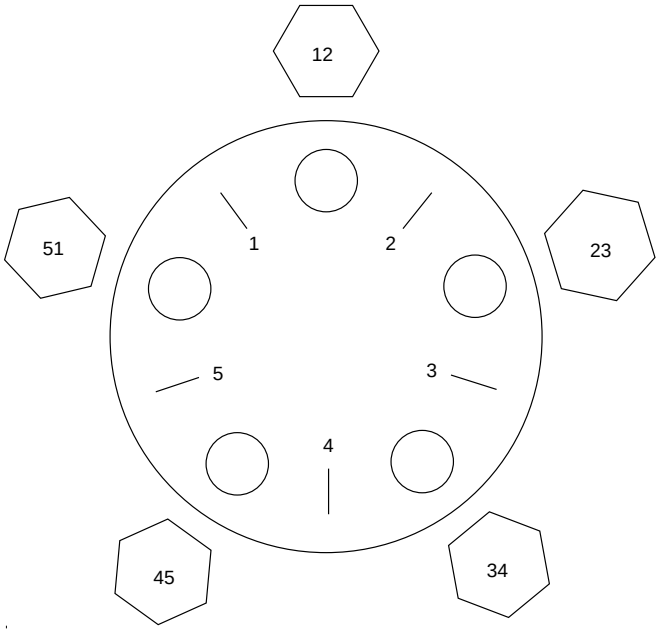
Dit welgekend voorbeeld werd ontleend aan [Cia89b, Jac91]. In de implementatie van programmafragment 5.10 denken en dineren de filosofen afwisselend. Om te kunnen eten moeten ze eerst een stoel kunnen bemachtigen, en zodra ze aan tafel zitten moeten ze proberen twee vorken te bemachtigen (zie afbeelding 5.4). Om vastlopen te vermijden mag een filosoof nooitéén vork voor onbeperkte tijd vasthouden. Indien hij de tweede vork niet meteen na de eerste kan grijpen, dan moet hij de eerste meteen weer vrijgeven.

```
start(V,T) :-
   vluchtgegevensbank(V),
   maak_terminals(T).

vluchtgegevensbank(0).
vluchtgegevensbank(V) :-
   zetels(V,Zetels),
   vlucht(V,Zetels,inladen)&,   % vluchttaak
   !vlucht(V),
   V1 is V - 1,
   vluchtgegevensbank(V1).

maak_terminals(0).
maak_terminals(T) :-
   terminal(T)&,
   T1 is T-1,
   maak_terminals(T1).

terminal(T) :-
   gebruikersinvoer(T,Vlucht,Aantal),
   ??!vlucht(Vlucht),   % bestaat vlucht ?
   !reservatie(Vlucht,reserveer,T,Aantal),
   ?bevestiging(T,Vlucht,Aantal,Toestand),
   gebruikersuitvoer(T,Vlucht,Aantal,Toestand),
   terminal(T).

vlucht(Vlucht, Vrij, Toestand) :-
   ?reservatie(Vlucht, Vraag, T, Aantal),
   vlucht(Vraag, Vlucht, T, Aantal, Vrij, Toestand).

vlucht(reserveer, Vlucht, T, Aantal, Vrij, inscheping) :-
   reserveer_zetels(Aantal, Vrij, Over, Toestand), !,
   !bevestiging(T, Vlucht, Aantal, Toestand),
   vlucht(Vlucht, Over, inscheping).
vlucht(vertrek, Vlucht, _, _, Vrij, inscheping) :-
   ?vlucht(Vlucht),
   vlucht(Vlucht, Vrij, vertrokken).
vlucht(reserveer,Vlucht,T,Aantal,Vrij,vertrokken) :-
   !bevestiging(T, Vlucht, Aantal, vertrokken),
   vlucht(Vlucht, Vrij, vertrokken).

reserveer_zetels(Aantal, Vrij, Over, aanvaard) :-
   Vrij >= Aantal, !,  Over is Vrij - Aantal.
reserveer_zetels(_, Vrij, Vrij, geweigerd).
```

Programmafragment 5.9 Vluchtreservatiesysteem.

```
start(S,F) :-
   tafel(S),
   filosofen(F).

tafel(S) :- stoelen(S,S).

stoelen(0,S) :- !.
stoelen(V1,S) :-
   V2 is (V1 mod S) + 1,
   !stoel(V1, V2),
   !vork(V1),
   V3 is V1-1,
   stoelen(V3,S).

filosofen(0).
filosofen(F) :-
   filosoof(F)&,
   F1 is F-1,
   filosofen(F1).

filosoof(F) :-
   denk(F),
   ?stoel(V1,V2),
   neem_vorken(V1,V2),
   eet(F),
   geef_vorken(V1,V2),
   !stoel(V1,V2),
   filosoof(F).


neem_vorken(V1,V2) :-
   ?vork(V1),
   ??vork(V2),
   !.
neem_vorken(V1,V2) :-
   !vork(V1),
   neem_vorken(V2,V1).

geef_vorken(V1,V2) :-
   !vork(V1),
   !vork(V2).
```

Programmafragment 5.10 Dinerende filosofen.



Afbeelding 5.4 Dinerende filosofen.

Elke filosoof wordt voorgesteld door een taak die afwisselend nadenkt en eet. Het aantal filosofen kan groter zijn dan het aantal zitplaatsen. Een filosoof probeert een stoel te pakken te krijgen met `?stoel(V1,V2)`. De stoel bepaalt welke vorken hij later moet proberen in handen te krijgen alvorens te kunnen beginnen eten. De routine `neem_vorken/2` lijkt op het eerste zicht een beetje gecompliceerd. Eerst wordt vork `V1` genomen. Indien deze niet beschikbaar is, dan wordt er gewacht. Van zodra vork `V1` genomen is, wordt vork `V2` genomen. Indien deze beschikbaar is kan de filosoof beginnen eten. Indien deze niet beschikbaar is, faalt deze bepaling en zal in de tweede bepaling de vork `V1` teruggegeven worden door `!vork(V1)`, en `neem_vorken` recursief opgeroepen worden, zij het nu met gecommuteerde argumenten. Vork `V2` wordt nu eerst genomen, en doordat deze net tevoren niet beschikbaar was, zal er nu gewacht worden, tenzij hij inmiddels vrijgegeven zou zijn. Er is dus voor gezorgd dat er enkel gewacht kan worden op de eerste vork, nooit op de tweede, dit om vastlopen te vermijden. Het gebruik van een voorwaardelijke leesbewerking zou dit stukje programma veel eenvoudiger kunnen maken, maar is niet beschikbaar in de kerntaal. Merk terloops op dat dit programma zich eerder ongewoon gedraagt. De voorgrondtaak termineert na de initialisatiefase. De achtergrondtaken blijven echter doorgaan en vormen het eigenlijke

```
filosoof(F) :-
   denk(F),
   ?pasje,
   ?stoel(V1,V2),
   neem_vorken(V1,V2),
   eet(F),
   geef_vorken(V1,V2),
   !stoel(V1,V2),
   !pasje,
   filosoof(F).
```

Programmafragment 5.11 Dinerende filosofen zonder uitsluiting.

programma.

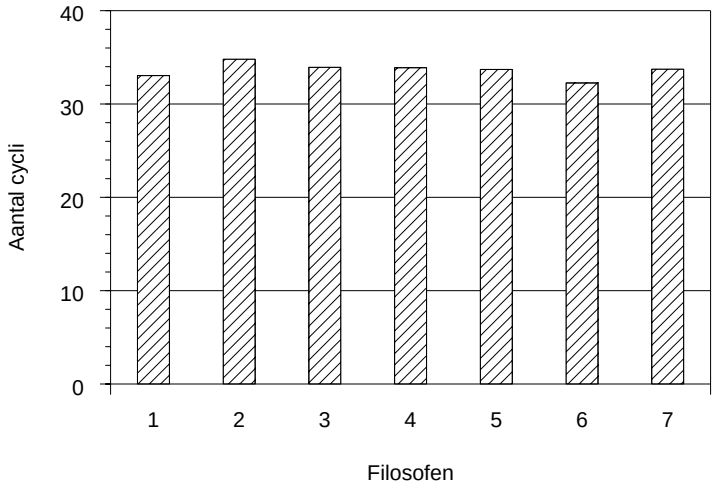
Louter op basis van programmafragment 5.10 kan evenwel niet gegarandeerd worden dat de hoeveelheid voedsel min of meer eerlijk onder de filosofen verdeeld zal worden. De filosofen kunnen samenzweren waardoor het mogelijk is dat een filosoof oneindig lang moet wachten vooraleer twee vorken te kunnen bemachtigen en hierdoor zal verhongeren. Dit probleem kan eenvoudig vermeden worden door steeds één stoel vrij te laten. Hierdoor zal elke filosoof uiteindelijk steeds aan zijn trekken komen [Rin88].

De wellicht eenvoudigste manier om dit te realiseren in MULTI-PROLOG is om een aantal pasjes op het bord te plaatsen (één minder dan het aantal stoelen). Een filosoof moet dan eerst in het bezit zijn van een pasje vooraleer een stoel te kunnen nemen. De filosooftaak wordt hierdoor nauwelijks ingewikkelder zoals blijkt uit programmafragment 5.11.

Het feit dat het bovenstaande programma naar behoren werkt mag blijken uit afbeelding 5.5 die het aantal keren dat de filosofen eten gedurende een periode van 1 s uitzet. Hieruit blijkt dat alle filosofen voldoende keren aan bod komen en dus zeker niet van honger zullen omkomen.

Samenvatting

Dit handvol voorbeelden toont aan dat MULTI-PROLOG zeker voldoende expressieve kracht bezit om een hele reeks van courante problemen zonder problemen en op een vrij directe manier op te lossen. Dit toont nogmaals aan dat een bordgebaseerde taal zeker niet moet onderdoen voor de stroomgebaseerde talen. Wij zijn er zelfs van overtuigd dat de hier gepresenteerde programma's een eenvoudiger structuur hebben, en daardoor een stuk gemakkelijker te begrijpen zijn dan hun stroomgebaseerde equivalenten.



Afbeelding 5.5 Verdeling van het aantal cycli van de filosooftaak voor zeven filosofen en vijf plaatsen.

6 Besluit

In dit proefschrift worden het ontwerp, een operationele en een declaratieve semantiek, en een prototype-implementatie van een parallelle logische programmeertaal voorgesteld. Het geheel wordt doorspekt met voorbeelden en vergelijkingspunten met de literatuur over andere parallelle logische programmeertalen.

Teruggrijpend naar de thesis van dit proefschrift op blz. 15 kan er als besluit geformuleerd worden dat taakparallellisme en expliciete bordcommunicatie een *elegante, logisch verantwoorde, efficiënte en algemeen bruikbare uitbreiding* is voor een sequentiële logische programmeertaal. MULTI-PROLOG is daarvan het bewijs.

- MULTI-PROLOG is *elegant* gezien het geringe aantal uitbreidingen, en de eenvoud van het bordmodel. Doordat het bord zelf zorg draagt voor zijn integriteit, wordt de gebruiker verlost van tal van synchronisatieproblemen. Het bord is voor iedereen zichtbaar en door iedereen bruikbaar. Zijn globale en statische natuur maakt het vrij gemakkelijk te beheersen. Dit alles volgt uit hoofdstuk 2.

- MULTI-PROLOG vormt een logisch verantwoorde uitbreiding van PROLOG. Bordcommunicatie is een neveneffect, maar weléén dat vrij goed past in het kader van het logisch programmeren. Gegevens op het bord gedragen zich als een soort van logische variabele. Bovendien kan de semantiek van een Multi-Prologvraag nog steeds op de klassieke manier beschreven worden. Dit wil zeggen dat er uitwendig weinig of geen verschil is met PROLOG. Deze resultaten volgen uit hoofdstuk 3.

- MULTI-PROLOG is efficiënt. Door het bord te implementeren op een multiprocessor met gemeenschappelijk geheugen kunnen de bordbewerkingen bijzonder snel gemaakt worden. Doordat alle uitbreidingen in de vorm van ingebouwde predicaten kunnen gerealiseerd worden kan er gebruik

gemaakt worden van een snelle sequentiële Prologimplementatie als onderbouw. De communicatiesnelheid is hoofdzakelijk afhankelijk van de kwaliteit van de bordimplementatie en in veel mindere mate van de toepassing zelf. De efficiëntie blijkt uit de prestatieresultaten in hoofdstuk 4.

- MULTI-PROLOG is algemeen bruikbaar. Een hele reeks toepassingen kan op eenvoudige en efficiënte manier geprogrammeerd worden. Een Multi-Prologprogramma bestaat uit sequentiële Prologprogramma's doorspekt met bordprimitieven. Dit maakt de drempel om de taal te gaan gebruiken klein. De sequentiële taken kunnen nagenoeg onafhankelijk van elkaar ontwikkeld worden. De specificatie van de communicatie met het bord volstaat. Dit wordt geïllustreerd in hoofdstuk 5.

In dit proefschrift wordt de huidige stand van zaken in het Multi-Prologonderzoek weergegeven. Er blijven echter nog tal van interessante aspecten te onderzoeken.

(i) De implementatie van de voorwaardelijke communicatieprimitieven.

(ii) De implementatie op een standaardplatform onder UNIX, hierbij gebruik makend van een meerdradig uitvoeringspakket.

(iii) De implementatie op een multiprocessor met gespreid geheugen.

(iv) De uitbreiding van de semantiek zodat alle primitieven behandeld worden.

(v) Het verder uitwerken van een programmeermethodologie voor de bordgebaseerde talen.

Een aantal van deze aspecten is echter voldoende complex om een volledig doctoraat aan te wijden.

7 Bibliografie

L'exactitude de citer,
c'est un talent beaucoup plus rare que l'on pense.

— BAYLE, *Dict.*, art. *Sanchez.*

[AAF+91] J. Almgren, S. Andersson, L. Flood, C. Frisk, H. Nilsson, en J. Sundberg. SICStus Prolog Library Manual. SICS technical report T91:12B, SICS, Kista, Sweden, 1991.

[ACCD89] V. Ambriola, P. Ciancarini, A. Corradini, en M. Danelutto. Shell: A Shell Hierarchical Environment Based on a Logic Language. Technical Report TR-31/89, dipartimento di informatica, università di pisa, augustus 1989.

[ACD90] V. Ambriola, P. Ciancarini, en M. Danelutto. Design and Distributed Implementation of the Parallel Logic Language Shared Prolog. *SIGPLAN Notices*, 25(3):40–49, maart 1990.

[ACG86] S. Ahuja, N. Carriero, en D. Gelernter. Linda and Friends. *IEEE Computer*, 19(8):26–34, augustus 1986.

[ACHS88] K. Appleby, M. Carlsson, S. Haridi, en D. Sahlin. Garbage Collection for Prolog Based on WAM. *Communications of the ACM*, 31(6):719–741, juni 1988.

[AR90] Khayri A.M. Ali en Karlsson Roland. The Muse Approach to Or-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, april 1990.

[AS88] L. Alkalaj en E. Shapiro. An Architectural Model for a Flat Concurrent Prolog Processor. In Kowalski en Bowen [KB88].

[Bai87] M.G. Bailey. Spreadsheets and Databases - Alternatives to Programming for Non-Computer Science Majors. *SIGCSE*, 19(1):499–503, februari 1987.

[Bak85] Raymond Bakker, editor. *Kluwer's Woordenboek Informatica Nederlands-Engels Engels-Nederlands*. Kluwer Technische Boeken, Antwerpen, eerste druk, 1985.

[Bar85] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North Holland, Amsterdam, revised edition, 1985.

[BC89] A. Brogi en P. Ciancarini. The Concurrent Language Shared Prolog. Technical Report TR-10/89, dipartimento di informatica, università di pisa, februari 1989.

[BC91] Antonio Brogi en Paolo Ciancarini. The Concurrent Language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, januari 1991.

[BHR84] S.D. Brookes, C.A.R. Hoare, en A.W. Roscoe. A Theory of Communicating Sequential Processes. *JACM*, 31(3):560–599, juli 1984.

[Bie88] Henk Biemond. *Woordenboek Automatisering*. Koninklijke PBNA, derde druk, 1988.

[BS90] P.A. Buhr en R.A. Stroobosscher. The μSystem: Providing Light-weight Concurrency on Shared Memory Multiprocessor Computers Running Unix. *Software - Practice and Experience*, 20(9):929–964, september 1990.

[Car87] Mats Carlsson. Internals of SICStus Prolog version 0.6, Logic Programming Systems. Technical report, SICS, Zweden, 1987.

[CC91] T. Catagnetti en P. Ciancarini. Static Analysis of a Parallel Logic Language Based on the Blackboard Model. *Journal of Parallel and Distributed Computing*, 13:412–423, 1991.

[CFL88] J. Cunha, M. Ferreira, en Pereira L. Programming in Delta Prolog. Technical report, Departamento de Informática, Universidade Nova de Lisboa, november 1988.

[CG81] Keith Clark en Steve Gregory. A Relational Language for Parallel Programming. In *Proceedings of the ACM Conference on Functional Programmming languages and Computer Architecture*, blzn 171–178, oktober 1981.

[CG86a] N. Carriero en D. Gelernter. The S/Net's Linda Kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, mei 1986.

[CG86b] Keith Clark en Steve Gregory. PARLOG: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.

[CGL86] N. Cariero, D. Gelernter, en J. Leichter. Distributed Data Structures in Linda. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, blzn 236–242. ACM, januari 1986.

[Cia89a] P. Ciancarini. Blackboard Programming in Shared Prolog. In *Proceedings of te 2nd Workshop on Parallel Languages and Compilers*, blzn 1–17, augustus 1989.

[Cia89b] P. Ciancarini. Coordination Languages for Open System Design. Technical Report TR-41/89, dipartimento di informatica, università di pisa, 1989.

[Cia89c] P. Ciancarini. System Programming with Logic Languages. Technical Report TR-30/89, dipartimento di informatica, università di pisa, augustus 1989.

[Cia91] P. Ciancarini. Parallel Logic Programming using the Linda model of Computation. In Goos en Hartmanis [GH91], blzn 110–125.

[CM81] W.F. Clocksin en C.S. Mellish. *Programming in Prolog*. Springer Verlag, Berlin, 1981.

[CM87] W.F. Clocksin en C.S. Mellish. *Prolog, beschrijving van de standaard*. Kluwer Technische Boeken, eerste druk, 1987. Vertaling van [CM81].

[CMCP92] J.C. Cunha, P.D. Medeiros, M.B. Carvalhosa, en L.M. Pereira. Delta-Prolog: A Distributed Logic Programming Language and its Implementation on Distributed Memory Multiprocessors. In Kacsuk en Wise [KW92], blzn 335–356.

[Con87] J.S. Conery. *Parallel Execution of Logic Programs*. Parallel Processing and Fifth Generation Computing. Kluwer Academic Publishers, Boston, 1987.

[Cor89] D.D. Corkill. Advanced Architectures: Concurrency and Parallelism. In Jagannathan et al. [JDB89], blzn 77–83.

[CWA+91] M. Carlsson, J. Widén, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, en T. Sjöland. SICStus Prolog User's Manual. SICS technical report T91:11B, SICS, Kista, Sweden, 1991.

[DB86] K. De Bosschere. Performantieverhoging door Coprocessoren in Multi-Interpreteromgeving. Ingenieursthesis, Faculteit van de Toegepaste Wetenschappen, RU Gent, juni 1986.

[DB87] K. De Bosschere. EDULAN, een Didactisch Softwarehulpmiddel bij de Studie van Systeemsoftware. Licentiaatsthesis, Faculteit van de Toegepaste Wetenschappen, RU Gent, juni 1987.

[DB89a] K. De Bosschere. Multi-Prolog, a Process-Oriented Prolog. In *Proceedings of the Second International Conference on Software Engineering for Real-time Systems*, blzn 6–10, Cirencester, UK, september 1989.

[DB89b] K. De Bosschere. Multi-Prolog, Another Approach for Parallelizing Prolog. In D.J. Evans, G.R. Joubert, en F.J. Peters, editors, *Proceedings of Parallel Computing 89*, blzn 443–448. Elsevier North-Holland, augustus 1989.

[DB89c] K. De Bosschere. Parallelism in Logic Programming. LEM Technical Report DG 89-02, Laboratorium voor Elektronica en Meettechniek, RU Gent, 1989.

[DB89d] K. De Bosschere. Parallellisme in Prolog-achtige Talen. *Technology Transfer Express*, 63:9–11, maart 1989.

[DB89e] K. De Bosschere. Predicate Calculus and Natural Deduction as a Tool for Logic Inference. *CCAI, The Journal for the Integrated Study of Artificial Intelligence, Cognitive Science, and Applied Epistemology*, 6(2/3):91–110, november 1989.

[DB90a] K. De Bosschere. Semantics of Communicating Horn Clauses. LEM Technical Report DG 90-08, Laboratorium voor Elektronica en Meettechniek, RU Gent, november 1990.

[DB90b] K. De Bosschere. The Semantics of Horn Clause Logic. LEM Technical Report DG 90-06, Laboratorium voor Elektronica en Meettechniek, RU Gent, november 1990.

[DB90c] K. De Bosschere. The Use of Clause Forms and Resolution for Proving Validity. *CCAI, The Journal for the Integrated Study of Artificial Intelligence, Cognitive Science, and Applied Epistemology*, 7(1):101–131, 1990.

[DB91a] K. De Bosschere. Blackboard Communication in Prolog. In *Parallel Execution of Logic Programs, ICLP'91 Pre-Conference Workshop*, volume 569 of *Lecture Notes in Computer Science*, blzn 159–172, Paris, juni 1991. Springer Verlag.

[DB91b] K. De Bosschere. Prolog, and its Relation to Clausal Logic. *CCAI, The Journal for the Integrated Study of Artificial Intelligence, Cognitive Science, and Applied Epistemology*, 7(3-4):279–290, 1991.

[DBJ92] K. De Bosschere en J.-M. Jacquet. Comparative Semantics of μLog. In D. Etienne en J.-C. Syre, editors, *Proceedings of the PARLE'92 Conference*, volume 605 of *Lecture Notes in Computer Science*, blzn 911–926, Paris, juni 1992. Springer Verlag.

[DBW90] K. De Bosschere en L. Wulteputte. Experiments with Prolog Execution Mechanisms. LEM Technical Report DG 90-03, Laboratorium voor Elektronica en Meettechniek, RU Gent, maart 1990.

[DBW91] K. De Bosschere en L. Wulteputte. Multi-Prolog: Implementation on an 88000 Shared Memory Multiprocessor. LEM Technical Report DG 91-19, Laboratorium voor Elektronica en Meettechniek, RU Gent, december 1991.

[DBW92] K. De Bosschere en L. Wulteputte. Multi-Prolog, A Blackboard Based Parallel Prolog. LEM Technical Report DG 92-09, Laboratorium voor Elektronica en Meettechniek, RU Gent, juni 1992.

[Deg87] Doug Degroot. Restricted And-Parallelism and Side Effects. In *Proceedings of the 1987 Symposium on Logic Programming*, blzn 80–89, augustus 1987.

[Den85] P.J. Denning. The Evolution of Parallel Processing. *American Scientist*, 73:414–415, 1985.

[DKM84] C. Dwork, P.C. Kanellakis, en J.C. Mitchell. On the Sequential Nature of Unification. *Journal on Logic Programming*, 1(1):35–50, 1984.

[DVC90] E. Debaere en J. Van Campenhout. *Interpretation and Instruction Path Coprocessing*. Computer Systems Series. The MIT Press, 1990.

[Eli92] A. Eliëns. *DLP, A Language for Distributed Logic Programming*. Series in Parallel Computing. Wiley, 1992.

[EM88] R. Engelmore en T. Morgan, editors. *Blackboard Systems*. The insight series in Artificial Intelligence. Addison-Wesley, Wokingham, England, 1988.

[Faa91] C. Faasen. Intermediate Uniformly Distributed Tuple Space on Transputer Meshes. In Goos en Hartmanis [GH91], blzn 157–173.

[FF92] Sz. Ferenczi en I. Futó. CS-Prolog, A Communicating Sequential Prolog. In Kacsuk en Wise [KW92], blzn 357–378.

[FK89] I. Futo en P. Kacsuk. CS-Prolog on multitransputer systems. *Microprocessors and Microsystems*, 13(2):103–112, maart 1989.

[FL77] R.D. Fennell en V.R. Lesser. Parallelism in Artificial Intelligence Solving: A Case Study of Hearsay-II. *IEEE Transactions on Computers*, 26(2):98–111, februari 1977.

[GDBD92] D. Gudeman, K. De Bosschere, en S. Debray. `jc`: An Efficient and Portable Sequential Implementation of Janus. In Krzysztof Apt, editor, *Joint International Conference and Symposium on Logic Programming*, blzn 399–413, Washington, november 1992. MIT press.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, januari 1985.

[Gel89] David Gelernter. Multiple Tuple Spaces in Linda. In *Proceedings of the PARLE'89 Conference*, volume 365 of *Lecture notes on Computer Science*, blzn 20–27, 1989.

[GH86] G. Goos en J. Hartmanis, editors. volume 225 of *Lecture Notes in Computer Science*, London, juli 1986. Springer Verlag.

[GH91] G. Goos en J. Hartmanis, editors. *Proceedings of Research Directions in High-Level Parallel Programming Languages*. Springer Verlag, Mont Saint Michel, France, juni 1991.

[GJ90] D. Gelernter en S. Jagannathan. *Programming Linguistics*. The MIT Press, Cambridge, Massachusetts, 1990.

[GL88] J. Goossenaerts en J. Lewi. Object-Oriented Programming : Concepts. Technical report, Dept. Computerwetenschappen, Katholieke Universiteit Leuven, april 1988.

[GM] J.A. Goguen en J. Messenguer. Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. Personal communication.

[Gre87] S. Gregory. *Parallel Logic Programming in Parlog*. International Series in Logic Programming. Addison Wesley, Wokingham, 1987.

[HB88] S. Haridi en P. Brand. Andorra Prolog – An Integration of Prolog and Committed Choice Languages. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, blzn 745–754, Tokyo, Japan, december 1988.

[HCF84] H. Hirakawa, T. Chikayama, en K. Furukawa. Eager and Lazy Enumerations in Concurrent Prolog. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Logic Programming Conference*, Uppsala, juli 1984.

[Her86] M.V. Hermenegildo. An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs. In Goos en Hartmanis [GH86].

[HJ90] Seif Haridi en Sverker Janson. Kernel Andorra Prolog and its Computational Model. In *Proceedings of the Seventh International Conference on Logic Programming*, blzn 31–46, Jerusalem, Israël, juni 1990.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, augustus 1978.

[Hoa85] CAR Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HW88] F.G. Hilgevoord en H.K.U. Wind. *Inleiding in Prolog*. Samson, Alphen aan den Rijn, 1988.

[Ind] Inducom Systems. *PMA-030 88000/68K Real-Time Program Analyzer User's Manual*, firmware revision 1.6.

[Ind89] Industrial Programming, Inc, Jericho, New York. *MTOS-UX User's Guide*, 1989.

[Jac91] J.-M. Jacquet. *Conclog: A Methodological Approach to Concurrent Logic Programming*, volume 556 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[JDB89] V. Jagannathan, R. Dodhiawala, en L.S. Baum, editors. *Blackboard Architectures and Applications*. Perspectives in Artificial Intelligence. Academic Press, 1989.

[JM91a] J.-M. Jacquet en L. Monteiro. Extended Horn Clauses: the Framework and its Semantics. In J.C.M. Baeten en J.F. Groote, editors, *Proceedings of the Concur'91 Conference*, volume 527 of *Lecture Notes on Computer Science*, blzn 281–297, Amsterdam, 1991. Springer Verlag.

[JM91b] Jean-Marie Jacquet en Luís Monteiro. Comparative Semantics of Generalized Horn Clauses. In K. Furukawa, editor, *Proceedings of the Japanese Logic Programming Conference*, Lecture Notes in Computer Science. Springer Verlag, juli 1991.

[KB88] R.A. Kowalski en K.A. Bowen, editors. *Proceedings of the fifth International Conference and Symposium on Logic Programming*, volume 2 of *MIT Press Series in Logic Programming*, Cambridge, Massachusetts, 1988. MIT Press.

[Kos87] Y. Koseki. Amalgating Multiple Programming Paradigms in Prolog. In J. McDermott, editor, *Proceedings of the Tenth International Joint Conference on Artificial Intelligence 1987*, IJCAI, Inc, Los Altos, California, augustus 1987. Morgan Kaufman Publishers.

[Kow79] R. Kowalski. Algorithms = Logic + Control. *Communications of the ACM*, 22(7):424–436, juli 1979.

[KW92] Peter Kacsuk en Michael J. Wise, editors. *Implementations of Distributed Prolog*. Series in Parallel Computing. Wiley, Chichester, 1992.

[Lam84] L. Lamport. An Axiomatic Semantics of Concurrent Programming Languages. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series F: Computer and System Sciences*, blzn 77–122. Springer Verlag, Berlin, 1984.

[LE77] V.R. Lesser en L.D. Erman. A Retrospective View of the Hearsay-II Architecture. In *Proceedings of the IJCAI-77*, blzn 790–800, 1977.

[Lev86] J. Levy. Shared Memory Execution of Committed-choice Languages. In Goos en Hartmanis [GH86].

[LK88] Y.J. Lin en V. Kumar. AND-parallel Execution of Logic Programs on a Shared Memory Multiprocessor : A Summary of Results. In Kowalski en Bowen [KB88].

[Llo84] J.W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, Berlin, 1984.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, second, 1987.

[Min74] M. Minsky. A Framework for Representing Knowledge. Artificial Intelligence Memo 306, MIT AI Lab, 1974.

[MN86] Paola Mello en Antonio Natali. Programs as Collections of Communicating Prolog Units. In *Proceedings of the ESOP*, volume 213 of *Lecture Notes on Computer Science*, 1986.

[Mon84] L. Monteiro. A Proposal for Distributed Programming in Logic. In J.A. Campbell, editor, *Implementations of Prolog*, Ellis Horwood Series in Artificial Intelligence, blzn 329–340. Ellis Horwood, Chichester, 1984.

[Mon86] L. Monteiro. Distributed Logic. internal report, Universidade Nova de Lisboa, Departamento de Informatica, april 1986.

[Mot88a] Motorola, Inc. *MC88100 RISC Microprocessor User's Manual*, 1988.

[Mot88b] Motorola, Inc. *MC88200 Cache/Memory Management Unit User's Manual*, 1988.

[Mot89a] Motorola, Inc. *MVME 188BUG User's Manual*, oktober 1989.

[Mot89b] Motorola, Inc. *MVME VMEmodule RISC Microprocessor User's Manual*, september 1989.

[Mye90] Brad A. Myers. Taxonomies of Visual Programming and Program Visualisation. *Journal of Visual Languages and Computing*, 1(1):97–123, maart 1990.

[Nai85] L. Naish. μProlog 3.2. Reference Manual. Technical Report 85/11, Department of Computer Science, University of Melbourne, 1985.

[New90] J.D. Newmarch. *Logic Programming: Prolog and Stream Parallel Languages*. Advances in Computer Science. Prentice Hall, 1990.

[NL91] B. Nitzberg en V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, blzn 52–60, augustus 1991.

[OAS91a] OASYS, Lexington, Massachussets. *88000 Assembler/Linker User's Guide*, maart 1991. v.2.04.

[OAS91b] OASYS, Lexington, Massachussets. *Green Hills Compiler Family Cross Development Guide*, januari 1991. Version 1.8.5.

[O'K90] R. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, Massachusetts, 1990.

[Pin91a] James Pinakis. The Design and Implementation of a Distributed Linda Tuple Space, 1991.

[Pin91b] James Pinakis. Providing Directed Communication in Linda, 1991.

[PM91] James Pinakis en Chris McDonald. The Inclusion of the Linda Tuple Space Operations in a Pascal-based Concurrent Language. *Australian Computer Science Communications*, 13(1):45.1–45.11, februari 1991.

[PMCA86] L.M. Pereira, L. Monteiro, J. Cunha, en J.N. Aparício. Delta Prolog : A Distributed Backtracking Extension with Events. In Goos en Hartmanis [GH86].

[PMCA88] L.M. Pereira, L.F. Monteiro, J.C. Cunha, en Aparício. Concurrency and Communication in Delta-Prolog. In *The Design and Application of Parallel Digital Processors*, volume 298, blzn 94–104. IEEE, 1988.

[pro91] ISO/IEC draft for Prolog working draft: 1991 (E), 1991.

[RAN89] J. Rice, N. Aiello, en H. Nii. See How They Run..., The Architecture and Performance of Two Concurrent Blackboard Systems. In Jagannathan et al. [JDB89], blzn 153–178.

[Rev88] G.E. Revesz. *Lambda-Calculus, Combinators, and Functional Programming*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1988.

[Rin88] G.A. Ringwood. Parlog86 and the Dining Logicians. *Communications of the ACM*, 31(1):10–25, januari 1988.

[Rip89] David L. Ripps. *An Implementation Guide to Real-Time Programming*. Yourdon Press, Englewood Cliffs, New Jersey, 1989.

[rpc88] XDR: eXternal Data Representation Protocol and RPC: Remote Procedure Calling Protocol. Technical report, SUN Microsystems, Inc, juni 1988. version 2.

[Rus83] Edward C. Russell. *Building Simulation Models with SIMSCRIPT II.5*. CACI, Inc.-Federal Modelling and Simulation Department, Los Angeles, september 1983.

[Sar89] V. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, 1989.

[Sch87] Henk Schotel. *Programmeren in Prolog*. Coutinho, Muiderberg, 1987.

[Sha86] E. Shapiro. Concurrent Prolog, A Progress Report. *IEEE Computer*, 19(8):44–58, augustus 1986.

[Sha87a] E. Shapiro. Systems Programming in Concurrent Prolog. In Ehud Shapiro, editor, *Concurrent Prolog: collected papers*, volume 2, blzn 6–27. MIT Press, 1987.

[Sha87b] Ehud Shapiro. A Subset of Concurrent Prolog and its Interpreter. In Ehud Shapiro, editor, *Concurrent Prolog: collected papers*, volume 1. MIT Press, 1987.

[Sha89] E.Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[SP91] G. Sutcliffe en J. Pinakis. Prolog-D-Linda: An Embedding of Linda in SICStus Prolog. Technical Report 91/7, Department of Computer Science, University of Western Australia, 1991.

[SS86] E. Shapiro en L. Sterling. *The Art of Prolog*. MIT Press Series in Logic Programming. The MIT Press, Cambridge, 1986.

[Str88a] B. Stroustrup. A Better C ? *BYTE*, 13(8):215–216D, augustus 1988.

[Str88b] B. Stroustrup. What is Object-Oriented Programming? *IEEE Software*, 5(3):10–20, mei 1988.

[Tar92] Paul Tarau. Low-Level Issues in Implementing a High-Performance Continuation Passing Prolog Engine. technical report 92-02, Departement d'Informatique, Université de Moncton, Moncton, 1992.

[Tay89] Stethen Taylor. *Parallel Logic Programming Techniques*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[TF85] Akikazu Takeuchi en Koichi Furukawa. Bounded Buffer Communication in Concurrent Prolog. *Journal of New Generation Computing*, 3(2):145–155, 1985.

[Tic91] E. Tick. *Parallel Logic Programming*. MIT Press, Cambridge, Massachusetts, 1991.

[TRG$^+$] A.Jr. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, en M. Young. Mach Threads and the Unix Kernel: The Battle for Control.

[Ued85] K. Ueda. Guarded Horn Clauses. In E. Wada, editor, *Logic Programmig '85. Proceedings of the Fourth Conference*, volume Lecture Notes in Computer Science 221, blzn 168–179, Tokyo, Japan, 1985. Springer Verlag.

[VC91] J. Van Campenhout. Ordinator-organisatie II. Syllabus 1991-1992, Faculteit van de Toepgepaste Wetenschappen, RU Gent, 1991.

[VCD87] J.M. Van Campenhout en E. Debaere. Language Coprocessor to Support the Interpretation of Modula-2 Programs. *Microprocessors and Microsystems*, 11(6):301–307, augustus 1987.

[VCSD84] J. Van Campenhout, R. Stoop, en H. Decuypere. The Implementation of CHILL on a Multi-Interpreter Architecture. In *Proceedings of the Third CHILL Conference*, blzn 49–54, Cambridge, september 1984.

[VR90] Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, 1990.

[vS91] Hein van Steenis. *Computer Jargon*. SYBEX, Soest, 1991.

[War83] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, Artificial Intelligence Center, SRI International, oktober 1983.

[WDB92] L. Wulteputte en K. De Bosschere. Superlinear Speedup for Partial Solutions of Combinatorial Problems in Multi-Prolog. In H.T. Dorrah, editor, *Proceedings of the second IASTED International Conference on Computer Applications in Industry*, blzn 312–315, Alexandria, Egypt, mei 1992. Acta Press.

[WDBVC92] L. Wulteputte, K. De Bosschere, en J. Van Campenhout. An Instruction Path Coprocessor to Speedup Warren's Abstract Machine. In *Proceedings of the Tenth IASTED International Conference on Applied Informatics*, blzn 98–101, Innsbrück, februari 1992. Acta Press.

[Wir82] N. Wirth. *Programming in Modula-2*. Springer-Verlag, New York, 1982.

[Wis92] M.J. Wise. Message-Brokers and Communicating Prolog Processes. In *Proceedings of PARLE'92: Parallel Architectures and Languages Europe*, volume 605 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[WJH92] M.J. Wise, D.G. Jones, en T. Hintz. PMS-Prolog: A Distributed Coarse-grain-parallel Prolog with Processes, Modules and Streams. In Kacsuk en Wise [KW92], blzn 379–403.

[Wul92] L. Wulteputte. Een Performantiestudie van Multi-Prolog op een Shared Memory Multiprocessor. Licentiaatsthesis, Faculteit van de Toegepaste Wetenschappen, 1992.

[Yan87] R. Yang. *P-Prolog, A Parallel Logic Programming Language*, volume 9 of *World Scientific Series in Computer Science*. World Scientific, Singapore, 1987.