# Geheugenmodellen voor parallelle en gedistribueerde systemen

Bart Van Assche

Promotor: Prof. dr. ir. E. D'Hollander
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Toegepaste Wetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: Prof. dr. ir. J. Van Campenhout
Faculteit Toegepaste Wetenschappen
Academiejaar 1999–2000

# Voorwoord

Dit doctoraal proefschrift kon slechts tot stand komen door de steun van meerdere personen. Mijn dank gaat hierbij in het bijzonder naar:

- prof. Erik D'Hollander, voor het waarnemen van het promotorschap en voor de talrijke aanwijzingen bij het schrijven van gefundeerde wetenschappelijke teksten;
- de kritische nalezers van dit proefschrift voor hun talrijke constructieve opmerkingen: prof. Erik D'Hollander, prof. Jan Van Campenhout, prof. Wilfried Philips en prof. Koen De Bosschere;
- dr. Noemie Slaats, voor de vertaling van de eigenschappen van geheugenmodellen naar VDM en de bewijsvoering in VDM van enkele van deze eigenschappen;
- de onderzoekers Kourosh Gharachorloo en Gil Neiger voor de verhelderende schriftelijke discussies over geheugenmodellen;
- de leden van de PARIS/MEDISIP reviewgroep voor hun constructieve inhoudelijke opmerkingen;
- de taalkundige Wouter Kongs voor het nauwgezet nalezen van dit proefschrift op spellings- en andere taalfouten – eventueel overgebleven taalonvolkomendheden blijven uiteraard mijn verantwoordelijkheid;
- de collega's van het labo voor de collegialiteit en de sfeer;
- het IWT voor de financiering van het gevoerde onderzoek;
- en tenslotte wil ik ook iedereen bedanken die morele steun verleende tijdens de realisatie van dit doctoraat.

Bart Van Assche
Gent, 24 april 2000.

# Notatie

## Predikatenlogica

| | |
|---|---|
| $=$ | Is gelijk aan. |
| $\triangleq$ | Wordt gedefinieerd als. |
| $\neq$ | Verschilt van. |
| $\square$ | Eind van een bewijs. |
| $P \wedge Q$ | Zowel $P$ als $Q$ gelden. |
| $P \vee Q$ | $P$, $Q$ of beide gelden. |
| $\neg P$ | $P$ geldt niet. |
| $P \Longrightarrow Q$ | Als $P$ geldt, dan geldt ook $Q$. |
| $\exists x : P(x)$ | Er bestaat een $x$ waarvoor $P(x)$ geldt. |
| $\exists\, x \in A : P(x)$ | Er bestaat een $x$ in verzameling $A$ waarvoor $P(x)$ geldt. |
| $\exists!\, x \in A : P(x)$ | Er bestaat exact één $x$ in verzameling $A$ waarvoor $P(x)$ geldt. |
| $\forall\, x \in A : P(x)$ | Voor elke $x$ in verzameling $A$ geldt $P(x)$. |

## Verzamelingen

| | |
|---|---|
| $\mathbb{N}$ | De verzameling van de natuurlijke getallen. |
| $\in$ | Is een element van. |
| $\notin$ | Is geen element van. |
| $\{\}$ | De lege verzameling. |
| $\{a\}$ | De singletonverzameling met als enig element $a$. |
| $\{a, b, c\}$ | De verzameling met de elementen $a$, $b$ en $c$. |
| $\#A$ | Het aantal elementen in de verzameling $A$. |
| $\{x \mid P(x)\}$ | De verzameling met alle elementen $x$ waarvoor predikaat $P(x)$ geldt. |

| | |
|---|---|
| $\{x \in A \mid P(x)\}$ | De verzameling met alle elementen $x$ uit $A$ waarvoor $P(x)$ geldt. |
| $A \cup B$ | De unie van $A$ en $B$. |
| $A \cap B$ | De doorsnede van $A$ en $B$. |
| $A \setminus B$ | De verzameling $A$ behalve de elementen uit $B$. |
| $A \subset B, A \subseteq B$ | Verzameling $A$ is een deelverzameling van verzameling $B$. |
| $\exists\, B \subset A : P$ | Er bestaat een deelverzameling $B$ van $A$ waarvoor $P$ geldt. |
| $\forall\, B \subset A : P$ | Voor elke deelverzameling $B$ van $A$ geldt $P$. |
| $A \times B$ | De productverzameling $\{(a, b) \mid a \in A \wedge b \in B\}$. |
| $A^2$ | De verzameling paren van elementen van $A$: $A^2 = A \times A = \{(a, a) \mid a \in A\}$. |
| $\bigcup_{i=0}^{n} A_i$ | De unie van de verzamelingen $A_0 \ldots A_n$. |
| $\bigcap_{i=0}^{n} A_i$ | De doorsnede van de verzamelingen $A_0 \ldots A_n$. |

## Relaties

| | |
|---|---|
| $R$ | Relatie $R$. |
| $aRb$ of $(a, b) \in R$ | De relatie $R$ geldt tussen $a$ en $b$. |
| $R_1 \cup R_2$ | Unie van de relaties $R_1$ en $R_2$. |
| $R_1 \cap R_2$ | Doorsnede van de relaties $R_1$ en $R_2$. |
| $dom(R)$ | Domein van de relatie $R$. |
| $R_1; R_2$ | Samenstelling van de relaties $R_1$ en $R_2$. |
| $R^0 = dom(R)^2$ | Nulvoudige samenstelling van $R$ met zichzelf. |
| $R^{n+1} = (R^n; R)$ | $(n+1)$-voudige samenstelling van de relatie $R$ met zichzelf. |

## Geheugenmodellen en berichtencommunicatie

| | |
|---|---|
| $Op$ | Verzameling van uitgevoerde opdrachten. |
| $\xrightarrow{\text{po}}$ | Programmaordening van opdrachten. |
| $\xrightarrow{\text{mo}}$ | Geheugenordening van opdrachten. |
| $(Op, \xrightarrow{\text{po}})$ | Uitvoering van een programma. |
| $(Op, \xrightarrow{\text{po}}, \xrightarrow{\text{mo}})$ | Uitvoering van een programma volgens de geheugenordeningsrelatie $\xrightarrow{\text{mo}}$. |
| $\mapsto$ | Schrijft-in relatie. |

| | |
|---|---|
| $C_m(op)$ | Aantal zendopdrachten via kanaal $m$ voorafgaand aan $op$. |
| $s$ SR $l$ | Het bericht verstuurd met opdracht $s$ werd ontvangen met opdracht $l$. |

# Afkortingen

| Afkorting | Betekenis |
|---|---|
| AM | Asynchronous Messages of asynchrone berichtenordening. |
| CC | Causal Consistency of causale consistentie. |
| CM | Causal Messages of causaal geordende berichten. |
| CISC | Complex Instruction Set Computer. |
| DSM | Distributed Shared Memory. |
| FIFO | First-In First-Out. |
| FM | FIFO-Ordered Messages. |
| MIMD | Multiple Instruction Streams, Multiple Data Streams. |
| MISD | Multiple Instruction Streams, Single Data Stream. |
| MMX | Multi-Media Extensions. |
| RISC | Reduced Instruction Set Computer. |
| PC | Processor Consistency of processorconsistentie. |
| PSO | Partial Store Ordering. |
| RC | Release Consistency. |
| s.c. | Sequentiële consistentie van relaties. |
| SC | Sequentiële consistentie van een geheugensysteem. |
| SIMD | Single Instruction Stream, Multiple Data Streams. |
| SISD | Single Instruction Stream, Single Data Stream. |
| SM | Synchronous Messages of synchroon geordende berichten. |
| t.o. | Totale ordening van relaties. |
| TSO | Total Store Ordering. |
| VLIW | Very Long Instruction Word. |
| VSM | Virtual Shared Memory. |
| WO | Weak Ordering. |

# Inhoudsopgave

# Lijst van tabellen

# Lijst van figuren

# Hoofdstuk 1

# Inleiding

## 1.1 Gedistribueerd rekenen

De benaming gedistribueerd rekenen duidt op de uitvoering van een computerprogramma op een gedistribueerd systeem. Een gedistribueerd systeem is een systeem dat bestaat uit twee of meer actieve onderdelen die onderling via één of meerdere communicatiekanalen verbonden zijn. Indien een informatiesysteem op geografisch verspreide locaties in communicatie met het systeem zelf voorziet, dan is dat informatiesysteem noodzakelijkerwijs gedistribueerd. Een voorbeeld hiervan is een netwerk van geldautomaten. Een andere toepassing van gedistribueerde systemen is de creatie van een systeem dat over aanzienlijk meer rekenkracht beschikt dan een enkel computersysteem. Recent werden bijvoorbeeld duizenden via het Internet verbonden computers ingezet in een geslaagde poging om geëncrypteerde boodschappen te kraken.

In dit doctoraat worden toepassingen van gedistribueerd rekenen beschouwd waarbij de toepassing uitgevoerd wordt op een gedistribueerd systeem omwille van de nood aan extra rekenkracht. Voor deze klasse van toepassingen bestaat er naast de gedistribueerde systemen een tweede klasse van systemen die een hoge rekenkracht leveren, nl. de multiprocessors. Terwijl gedistribueerde systemen eenvoudiger uitbreidbaar zijn, een lagere kostprijs en een hogere schaalbaarheid hebben dan vergelijkbare multiprocessors, hebben de multiprocessors als voordeel dat de interconnectie tussen de processors sneller is. Dit betekent dat bij toepassingen bestemd voor een gedistribueerd systeem meer aandacht zal moeten worden besteed aan het communicatiede-

biet tussen processors.

Twee veelgebruikte samenwerkingsmodellen voor toepassingen op gedistribueerde systemen zijn gemeenschappelijk geheugen en berichtendoorgave. Het eerste samenwerkingsmodel is gebaseerd op een gemeenschappelijke adresruimte voor de samenwerkende processen en impliciete communicatie. Bij berichtendoorgave daarentegen heeft elk proces een eigen adresruimte en is de interprocescommunicatie expliciet in de verschillende processen aanwezig. In beide gevallen wordt het verband tussen de (statische) programmatekst en de (dynamische) uitvoering van een programma mee bepaald door het systeem waarop het programma wordt uitgevoerd. Dit verband heeft op een systeem met gemeenschappelijk geheugen de naam geheugenmodel, en voor een systeem met berichtendoorgave de naam berichtencommunicatiemodel.

Deze modellen, die de semantiek van een systeem weergeven, bepalen in sterke mate de prestaties die met een systeem gehaald kunnen worden. Voor de meeste toepassingen voor gedistribueerde systemen vormt de communicatiesnelheid de beperkende factor voor de uitvoeringstijd. Vandaar dat er onderzoek wordt verricht naar methodes om de communicatie zo efficiënt mogelijk te laten verlopen. Multicast is een dergelijke techniek, en houdt in dat identieke informatie die naar meerdere bestemmingen verstuurd moet worden slechts één keer via het communicatiemedium verstuurd wordt.

In dit proefschrift worden twee facetten van het verband tussen communicatie in gedistribueerde systemen versus berichtendoorgave en gemeenschappelijk geheugen uitgewerkt. Het eerste facet is een systematische studie van geheugenmodellen voor gedistribueerde gemeenschappelijk-geheugensystemen. Een geheugenmodel bepaalt zowel de programmeringswijze van het systeem als de hoeveelheid gegenereerde communicatie. Er wordt een beschrijvingsmethode voorgesteld voor geheugenmodellen die de definitie en de vergelijking van geheugenmodellen eenvoudiger maakt dan wat mogelijk is met bestaande methodes.

Het tweede facet bestaat uit de uiteenzetting van de implementatie van multicast in een berichtencommunicatiesysteem als toepassing van geheugenmodellen; daarnaast wordt de invloed van multicast op de prestaties van gedistribueerde programma's gebaseerd op berichtencommunicatie nagegaan.

## 1.2 Overzicht van dit proefschrift

Dit proefschrift behandelt aspecten van gedistribueerde systemen, een domein dat behoort tot de parallelle systemen. Daarom geven we in hoofdstuk 2 een overzicht van de bestaande parallelle organisaties en architecturen en behandelen we ook enkele basiseigenschappen van parallelle programma's.

Hoofdstuk 3 handelt over de beschrijving van de wisselwerking tussen een parallel programma en een al of niet gedistribueerd parallel systeem. Voor programma's met gemeenschappelijk geheugen wordt deze wisselwerking beschreven door het zogenaamd geheugenmodel, en voor programma's gebaseerd op berichtencommunicatie door het zogenaamd berichtencommunicatiemodel. In dit hoofdstuk wordt een formalisme voorgesteld waarmee geheugenmodellen van bestaande parallelle systemen nauwkeurig kunnen worden beschreven. Bovendien wordt een duidelijk onderscheid gemaakt tussen algemeen geldende en modelspecifieke eigenschappen. Verder worden verschillende geheugenmodellen vergeleken op basis van het ingevoerde formalisme, en wordt ook het verband gelegd met de efficiëntie van de onderliggende implementatie.

In hoofdstuk 4 wordt de invloed nagegaan van de toepassing van de techniek met de naam *multicast* op de prestaties van gedistribueerde programma's gebaseerd op berichtendoorgave.

In hoofdstuk 5 volgen de besluiten van het in dit proefschrift beschreven onderzoek.

Na de bibliografie volgen de appendices met daarin de begrippen over wiskundige relaties die in het hoofdstuk over geheugenmodellen worden gebruikt (appendix A), en ook de bewijzen van enkele stellingen over de geheugenmodellen (appendix B).

## 1.3 Publicaties

Tijdens het doctoraatsonderzoek werd onderzoek verricht in volgende domeinen:

- Gedistribueerd gemeenschappelijk geheugen onder het Mach besturingssysteem.
- Graafvoorstelling van een parallel programma.

- Prestatievergelijking van parallelle programma's op systemen gebaseerd op gedistribueerd gemeenschappelijk geheugen versus berichtencommunicatie.
- Geheugenmodellen en de formele behandeling ervan met VDM.
- Toepassing van multicast in PVM voor snellere communicatie.
- Generatie van parallelle code via de parallelliserende compiler FPT (Fortran Parallel Transformer), ontwikkeld in ELIS.
- Parallellisatie van een simulatieprogramma voor gietprocessen.

In dit doctoraat zijn de onderwerpen geheugenmodellen en multicast opgenomen omdat voor deze onderwerpen de beste resultaten werden behaald.

Hieronder volgt een overzicht van de publicaties waaraan ik zelf heb bijgedragen, met de nadruk op de bijdragen die onderwerpen uit deze scriptie beschrijven. Andere bijdragen worden kort vermeld.

De originele bijdragen die werden voorgesteld in dit proefschrift zijn:

- In de publicaties [SVAH98, VAD00] wordt de formele voorstelling van geheugenmodellen behandeld. In de eerst vermelde bijdrage ([SVAH98]), verschenen als hoofdstuk in het boek *Proof in VDM: Case studies*, wordt de vertaling van de basiseigenschappen van geheugenmodellen naar de formele bewijsvoeringstaal VDM (*Vienna Development Method*) behandeld. Ook wordt de geschiktheid aangetoond van de geformuleerde eigenschappen voor automatische bewijsvoering. In de publicatie [VAD00] wordt aangetoond dat er duidelijk onderscheid gemaakt kan worden tussen basiseigenschappen en modelspecifieke eigenschappen voor een geheugenmodel, en ook dat het ingevoerde formalisme toelaat geheugenmodellen beknopt te formuleren.
- De implementatie van multicast in de PVM-daemon en het onderzoek naar de invloed van multicastcommunicatie op parallelle applicaties wordt ook beschreven in [VA97].

Naast bovenstaande bijdragen leverde ik ook bijdragen die niet in dit proefschrift werden opgenomen:

- De voor- en nadelen van berichtengebaseerde samenwerking versus gedistribueerd gemeenschappelijk geheugen voor handmatig resp. door een compiler geparallelliseerde programma's worden beschreven in [VAD96a, VAD96b, VAD97].

- In het afstudeerwerk [VA95] wordt de implementatie van een sequentieel consistent gedistribueerd gemeenschappelijk-geheugensysteem voor het Unix-achtige besturingssysteem Mach beschreven. Een kwantitatieve analyse van de verdeling van uitvoeringstijden over de verschillende softwarelagen van het besturingssysteem, pagineringssoftware en toepassingssoftware samen werd gepubliceerd in [VA96].

# Hoofdstuk 2

# Parallelle computerarchitecturen en programma's

In dit hoofdstuk worden drie indelingen voor computersystemen beschouwd. Een eerste indeling is volgens de architectuur – d.w.z. volgens de interface tussen de software van het laagste niveau en de hardware. Een tweede indeling van parallelle computersystemen is volgens organisatie. De term organisatie duidt op de wijze waarop een computer is opgebouwd uit onderdelen en hoe deze onderdelen samenwerken. De derde indeling is volgens het programmeermodel dat wordt toegepast voor interprocescommunicatie. Deze verschillende indelingen worden besproken in de respectieve paragrafen 2.1, 2.2 en 2.3. Vervolgens wordt in paragraaf 2.4 ingegaan op het mogelijk niet-deterministisch karakter van parallelle programma's.

Het onderscheid tussen architectuur en organisatie is nodig om in hoofdstuk 3 het begrip geheugenmodel te kunnen invoeren. Ook het begrip niet-determinisme is nodig in hoofdstuk 3.

## 2.1 Indeling volgens architectuur

Flynn beschouwt vier mogelijke architecturen voor computersystemen, al of niet met gemeenschappelijk geheugen [Fly66, Fly72]. Deze klassen verschillen van elkaar door het aantal instructie- en het aantal datastromen van en naar de verwerkingseenheden. Daarbij is een instructie-

stroom een sequentie van instructies die door een processor wordt uitgevoerd, en bestaat een datastroom uit de operandi en resultaten van de instructiestroom. De verschillende klassen worden aangeduid met de afkortingen SISD, SIMD, MISD en MIMD.

De eenvoudigste klasse is SISD (*single instruction stream, single data stream*). In een architectuur uit deze klasse worden één instructiestroom en één datastroom verwerkt. De meeste oudere monoprocessorsystemen vallen in deze categorie.

In een architectuur uit de SIMD-klasse (*single instruction stream, multiple data streams*) worden één instructiestroom en meerdere datastromen verwerkt. Deze klasse architecturen is ook gekend onder de benaming vectorcomputers. Moderne processors met multimedia-extensies, zoals de Pentium-processorfamilie met de MMX-instructieset, behoren in principe tot de SIMD-klasse. De meerderheid van de bestaande programma's maakt echter geen gebruik van deze extensies: deze programma's gebruiken alleen de SISD-deelverzameling van de Pentium-instructieset.

In de MISD-klasse (*multiple instruction streams, single data stream*) worden meerdere bewerkingen tegelijk toegepast op dezelfde data. Wegens hun beperkte toepasbaarheid bestaan er weinig MISD-computers. Een voorbeeld is de Multiflow Trace [SS93, CL94].

De MIMD-klasse (*multiple instruction streams, multiple data streams*) is de meest algemene parallelle architectuur. Elke processor verwerkt een eigen instructie- en datastroom. De interactie tussen de processors bestaat uit data-uitwisseling of synchronisatie.

Een bijzondere klasse van architecturen zijn de VLIW-systemen (*very long instruction word*). Een VLIW-systeem is een systeem met meerdere datastromen maar waarbij het controleverloop gemeenschappelijk is voor alle processors en waarbij de processors synchroon werken.

Verder in dit proefschrift worden MIMD-systemen bestudeerd waarbij elke processor een eigen controleverloop heeft, en waarbij deze systemen al of niet met gemeenschappelijk geheugen werken.

## 2.2 Indeling volgens organisatie

In de organisatie van een multiprocessor bestaat het geheugen voor een deel uit lokaal geheugen, uitsluitend toegankelijk voor een wel-

bepaalde processor, en voor een deel uit gemeenschappelijk geheugen, toegankelijk voor alle processors. Het volstaat als één van deze twee componenten aanwezig is en de mogelijkheid tot interprocessorcommunicatie voorzien is.

Beide organisaties kunnen als programmeermodel gemeenschappelijk geheugen of gedistribueerd geheugen hebben.

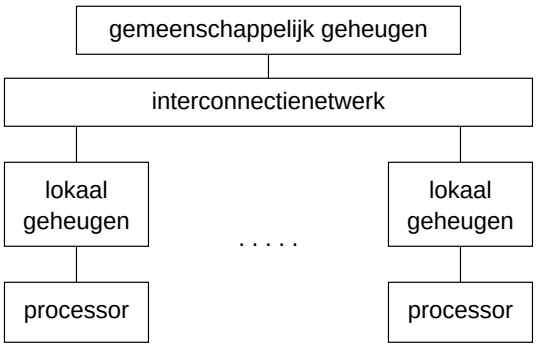
We onderscheiden volgende organisaties: zowel lokaal als gemeenschappelijk geheugen, alleen lokaal geheugen met ondersteuning voor het programmeermodel gemeenschappelijk geheugen en lokaal geheugen zonder ondersteuning voor het programmeermodel gemeenschappelijk geheugen.

### 2.2.1 Gemeenschappelijk geheugen

In figuur 2.1 is een systeem voorgesteld met gemeenschappelijk geheugen, waarbij de gemeenschappelijke data in het gemeenschappelijk geheugen worden bijgehouden. Meestal zijn in deze organisatievorm de lokale geheugens kleiner en sneller dan het gemeenschappelijk geheugen, en ze worden in dat geval als cache op het gemeenschappelijk geheugen ingezet. Deze organisatie wordt o.m. toegepast in multiprocessors. Multiprocessors hebben typisch een snelle verbinding tussen processors, lokale geheugens en gemeenschappelijk geheugen. Er geldt dat multiprocessors met deze organisatie minder schaalbaar zijn dan multiprocessors met de organisatie gedistribueerd gemeenschappelijk geheugen. Verder hebben multiprocessors een relatief hoge prijs per processor vergeleken met gedistribueerde systemen. Daar staat tegenover dat multiprocessors met de organisatie gemeenschappelijk geheugen een lagere prijs hebben dan multiprocessors met als organisatie gedistribueerd gemeenschappelijk geheugen, en meestal beter presteren dan gedistribueerde systemen met hetzelfde aantal processors.

### 2.2.2 Gedistribueerd gemeenschappelijk geheugen

Met de naam gedistribueerd gemeenschappelijk geheugen wordt hier de organisatie aangeduid waarin enkel lokaal en dus geen gemeenschappelijk geheugen aanwezig is en waarbij hardwareondersteuning voor het programmeermodel gemeenschappelijk geheugen aanwezig is. Een voorbeeld van een dergelijke organisatie staat in figuur 2.2. Alhoewel er door de organisatie dezelfde data in meerdere lokale geheu-



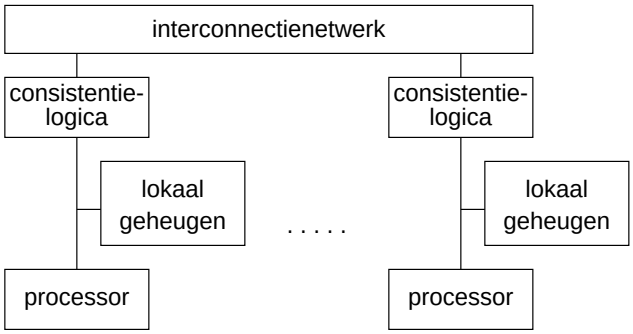

**Figuur 2.1:** Organisatie van een systeem met zowel lokaal als gemeenschappelijk geheugen.

gens aanwezig kunnen zijn gedraagt het geheel van lokale geheugens, consistentielogica en interconnectienetwerk zich als één gemeenschappelijk geheugen.

De essentie van het programmeermodel gemeenschappelijk geheugen is dat de samenwerkende processen één adresruimte gebruiken. Communicatie tussen processen vindt plaats doordat deze processen data naar één zelfde adres schrijven of eruit lezen. Als bovendien data voor een gegeven adres fysisch op meer dan één plaats worden bijgehouden, dan dienen alle kopieën corresponderend met dat adres dezelfde waarde te bevatten. Deze eigenschap wordt aangeduid met de benamingen *coherentie* of ook *consistentie*.

De consistentie tussen de verschillende lokale geheugens wordt in stand gehouden door de consistentielogica. Deze logica observeert de bus die een processor met zijn lokaal geheugen verbindt, en kan indien nodig de controle over de bus overnemen. Indien een processor leest uit of schrijft naar het lokale geheugen, zal de consistentielogica zonodig gegevens over het interconnectienetwerk kopiëren of invalideren. Naargelang de fysische afstand tussen de processors en het aantal te verbinden processors variëren de topologie en de snelheid van het interconnectienetwerk. Het interconnectienetwerk is meestal een bus, een hyperkubus (*hypercube*) of een meerwegskruising (*crossbar*).

Als nadeel van deze architectuur geldt dat de consistentie-eenheid typisch aanzienlijk groter is dan één enkele locatie uit het gemeenschappelijk geheugen, wat voor een aantal toepassingen het optreden van *false sharing* als gevolg heeft. Het fenomeen *false sharing* treedt op



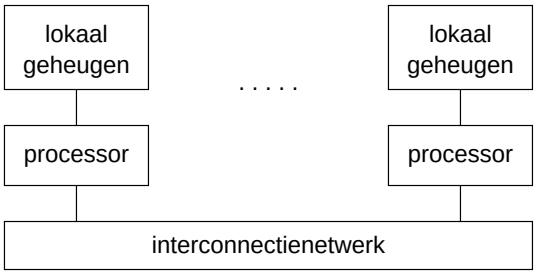

**Figuur 2.2:** Gedistribueerd gemeenschappelijk geheugen: systeem met als organisatie enkel lokaal geheugen en met ondersteuning voor het programmeermodel gemeenschappelijk geheugen.

als minstens twee processors data op verschillende adressen wijzigen, en deze data tot dezelfde consistentie-eenheid behoren. Dit heeft als gevolg dat minstens één van de betreffende processors veel frequenter beroep moet doen op interprocessorcommunicatie dan in het geval waarbij de data van elke processor in hun eigen lokaal geheugen geplaatst zijn.

Voorbeelden van deze organisatie zijn de Stanford DASH [LLG+92], de MIT Alewife [ABC+99], de Convex Exemplar [SMS96], Suns S3.mp [PBA+98], de SICS Data Diffusion Machine [HH89] en de Kendall Square Research 1 systemen [BHW+93].

### 2.2.3 Gedistribueerd geheugen

Met de naam gedistribueerd geheugen wordt hier de organisatie aangeduid waarin geen gemeenschappelijk geheugen aanwezig is en ook geen hardwareondersteuning voorzien is voor het programmeermodel gemeenschappelijk geheugen – zie ook figuur 2.3. Alle communicatie tussen de lokale geheugens gebeurt doordat een van de processen in uitvoering data via berichtencommunicatie (*Message Passing* of MP) naar een ander proces verstuurt, waardoor de data van het ene lokale geheugen naar een ander lokaal geheugen worden verstuurd. Terwijl op deze manier volledige controle mogelijk is over de verstuurde data en redundante communicatie kan worden geëlimineerd, vraagt het van een programmeur meer inspanning om programma's te schrijven voor een organisatie zonder gemeenschappelijk geheugen. Voorbeelden van




**Figuur 2.3:** Systeem met als organisatie enkel lokaal geheugen en zonder ondersteuning van het programmeermodel gemeenschappelijk geheugen.

deze organisatie zijn bijvoorbeeld een netwerk van werkstations of een systeem als de IBM SP2. In hoofdstuk 4 volgen nog enkele voorbeelden van deze organisatie.

## 2.3 Indeling parallelle computers volgens programmeermodel

Onder de programma's bestemd voor uitvoering op een MIMD-architectuur bestaan er zowel expliciet als impliciet parallelle programma's. Een programma is expliciet parallel als het bestaat uit meerdere sequentiële uniprocessorprogramma's. Een programma is impliciet parallel als het gaat om één sequentieel programma met daaraan directieven toegevoegd over de manier waarop er een expliciet parallel programma uit kan worden gegenereerd.

### 2.3.1 Expliciet parallellisme

Twee belangrijke programmeermodellen voor parallelle programma's zijn gemeenschappelijk geheugen en berichtencommunicatie. In het eerste model is er voor alle processen een gemeenschappelijke adresruimte, en worden data passief tussen processen uitgewisseld door deze data op een afgesproken adres te plaatsen. Bij berichtencommunicatie wordt er actief een bericht verstuurd telkens een proces gegevens naar een ander proces wil doorsturen. De primitieven voor synchronisatie zijn bij beide programmeermodellen van een zelfde abstractieniveau. Beide ondersteunen semafoor- en barrièresynchronisatie, en bij

gemeenschappelijk geheugen worden bovendien ook conditievariabelen ondersteund.

**Gemeenschappelijk geheugen** In expliciet parallelle programma's wordt in het programma zelf gespecificeerd welk proces welke instructiestroom uitvoert. In expliciet parallelle programma's moet kunnen worden uitgedrukt dat een nieuw proces gestart of beëindigd moet worden, hoe de scheduling van de processen onderling moet gebeuren, en waar en wanneer synchronisatie nodig is. Het hoofdkenmerk van gemeenschappelijk geheugen, nl. een gemeenschappelijke adresruimte, is niet direct zichtbaar in een expliciet parallel programma.

Programmeertalen zoals Modula-2, Ada en Java ondersteunen het schrijven van expliciet parallelle programma's met gemeenschappelijk geheugen. Voor programmeertalen zonder die ondersteuning bestaan er bijvoorbeeld de PARMACS-macro's [BBD+87] en de POSIX-draden (IEEE standaard 1003.1c, 1995) om expliciet parallelle programma's met gemeenschappelijk geheugen te ondersteunen.

**Berichtencommunicatie** Bij programma's gebaseerd op berichtencommunicatie is er functionaliteit nodig om processen te starten en te stoppen, om berichten samen te stellen uit meerdere afzonderlijke variabelen en ook om de vertaling uit te voeren tussen datarepresentaties van verschillende platformen. De programmeertalen Occam en Ada bijvoorbeeld voorzien in de mogelijkheid om programma's te baseren op berichtencommunicatie. Voor programmeertalen waarvan berichtencommunicatie geen standaardonderdeel is bestaan er softwarepakketten die in deze functionaliteit voorzien. Twee voorbeelden zijn PVM [SGDM94] (*Parallel Virtual Machine*) en MPI [DOSW96] (*Message Passing Interface*).

### 2.3.2 Impliciet parallellisme

In een impliciet parallel programma wordt met directieven o.m. aangegeven welke delen van het programma geparallelliseerd kunnen worden en welke de meest gunstige distributie is van de datastructuren over de verschillende lokale geheugens. Het expliciet parallel programma dat met een geschikt vertaalprogramma uit een impliciet parallel programma wordt gegenereerd kan ofwel gebaseerd zijn op het pro-

grammeermodel gemeenschappelijk geheugen, ofwel op het programmeermodel berichtencommunicatie.

Impliciet parallelle programma's worden geschreven als sequentiële programma's met directieven die aanduiden waar parallellisme mag worden toegepast. In impliciet parallelle programma's moet kunnen worden uitgedrukt welke lussen parallel mogen worden uitgevoerd en hoe de datadistributie dient te gebeuren. De OpenMP-directieven [DM98] en HPF-programmeertaal [Lov93, MRZ98] voorzien hierin. Vertaalsoftware zorgt dan voor de omzetting van de impliciet naar de expliciet parallelle vorm.

### 2.3.3 Programmeermodel versus organisatie

Alhoewel elk van de in paragraaf 2.2 beschreven organisatievormen een natuurlijk programmeermodel heeft, is dit niet noodzakelijk het enige mogelijk toepasbare programmeermodel voor die specifieke organisatie. Via software is het namelijk mogelijk berichtencommunicatie op een gemeenschappelijk-geheugenarchitectuur te implementeren, of gemeenschappelijk geheugen op een architectuur zonder gemeenschappelijk geheugen. De software die gemeenschappelijk geheugen mogelijk maakt op een systeem zonder hardwareondersteuning voor gemeenschappelijk geheugen staat bekend onder de naam DSM- of VSM-software (*Distributed Shared Memory* resp. *Virtual Shared Memory*). Een bekend voorbeeld is de TreadMarks-software [KDCZ94, ACD+96].

Er bestaan studies die de prestaties van berichtengebaseerde en DSM-systemen op dezelfde hardware, een netwerk van werkstations, met elkaar vergelijken – zie [LDCZ95, LDCZ97, VAD96b]. De door Lu handmatig geparallelliseerde DSM-programma's blijken gemiddeld 15% trager dan de equivalente MP-programma's. Zelf heb ik de uitvoeringstijden van de door een parallelliserende compiler gegenereerde DSM- en MP-programma's voor een zelfde sequentieel bronprogramma vergeleken. Uit de resultaten volgt dat de MP-variant het snelst is als het programma eenvoudig parallelliseerbaar is, en het traagst als de parallelliserende compiler suboptimale MP-code genereert.

## 2.4 Niet-determinisme bij gemeenschappelijk-geheugensystemen

Beschouw een expliciet parallel programma, bestaande uit meerdere processen, dat wordt uitgevoerd op een gemeenschappelijk-geheugensysteem. De interactie van elk proces met het gemeenschappelijk geheugen bestaat daarbij uit lees-, schrijf- en synchronisatieopdrachten. Onder **niet-determinisme** wordt dan verstaan dat uit de invoer naar het programma en uit de broncode van het programma zelf niet het volledige verloop van het programma kan worden afgeleid.

Als twee processen ongecoördineerd lezen uit of schrijven naar dezelfde locatie in het gemeenschappelijk geheugen, en minstens één van deze processen de data op die locatie wijzigt, dan doet er zich een **race** voor. Races in een programma zijn meestal ongewenst. Netzer deelt races in in twee klassen: **data-races** en **algemene races** [NM92]. Een data-race doet zich voor telkens wanneer twee processen ongecoördineerd een opdracht op dezelfde locatie uitvoeren, en minstens één van deze processen de data op die locatie wijzigt. Ongecoördineerd betekent in deze context dat er uit de broncode van de processen geen volgorde kan worden afgeleid. Een algemene race vindt plaats telkens zich een data-race voordoet, en als voor de twee processen die een opdracht op dezelfde locatie uitvoeren, deze opdrachten ofwel voor ofwel na elkaar worden uitgevoerd, maar er uit de programmatekst niet volgt in welke van de twee mogelijke volgordes deze opdrachten worden uitgevoerd. Data-races worden beschouwd als ongewenst omdat door data-races het gedrag van het programma niet-deterministisch wordt.

Er bestaat een verband tussen het al of niet optreden van data-races en het deterministisch zijn van een programma. Algemene races komen niet voor in deterministische programma's. Alhoewel meestal deterministisch gedrag verlangd wordt van een programma, bestaan er programma's die nuttig gebruik maken van algemene races. De programma's gebaseerd op chaotische relaxatie (*chaotic relaxation*, [CM69, Str97, SBKK98]) bijvoorbeeld zijn moeilijk even efficiënt te implementeren zonder gebruik te maken van algemene races. Er wordt daarbij ondersteld dat schrijfopdrachten in het geheugen, uitgevoerd door een processor binnen een eindige tijd, aan de andere processors worden bekendgemaakt. Hoe sneller dit gebeurt hoe beter dit is voor de convergentie van algoritmen gesteund op chaotische relaxatie. De techniek van chaotische relaxatie wordt o.m. nuttig toegepast door Kurreck bij

het numeriek oplossen van stromingsvergelijkingen [KW97].

Vertrekkend van het gedrag van een programma kunnen we drie klassen programma's onderscheiden: intern deterministische, extern deterministische en niet-deterministische programma's. **Intern deterministische** programma's zijn programma's die altijd resulteren in dezelfde uitgevoerde opdrachten in dezelfde volgorde. Deze programma's produceren dus steeds hetzelfde resultaat. **Extern deterministische programma's** geven altijd hetzelfde resultaat, maar dit resultaat mag bereikt worden zonder dat het verloop van het programma steeds hetzelfde is. Parallelle programma's die *load-balancing* toepassen behoren tot de klasse van extern deterministische programma's. Het resultaat van **niet-deterministische programma's** daarentegen kan verschillen van uitvoering tot uitvoering.

Een eigenschap van programma's zonder algemene races is dat deze programma's intern deterministisch worden uitgevoerd. Vandaar het belang van race-vrije programma's. Er werd en wordt nog steeds onderzoek verricht naar technieken om data-races op te sporen [Ron99].

# Hoofdstuk 3

# Modellering van gemeenschappelijk geheugen

## 3.1 Inleiding

Gemeenschappelijk geheugen is een abstractie die het mogelijk maakt dat meerdere processen in dezelfde adresruimte samenwerken als onderdeel van een parallelle applicatie. Daartoe voorziet het gemeenschappelijk geheugen in een aantal locaties waarin waarden kunnen worden opgeslagen, en is het mogelijk minstens lees- en schrijfopdrachten uit te voeren in het gemeenschappelijk geheugen.

Het gemeenschappelijk geheugen in een multiprocessor zorgt voor de uitwisseling van resultaten tussen de verschillende processen. In een uniprocessorsysteem zonder cachegeheugen is het reeds zo dat bij de uitvoering van een proces het communicatiekanaal tussen de processor en het hoofdgeheugen permanent zou worden aangesproken wanneer code en data rechtstreeks uit dit geheugen worden gehaald. Dit betekent dat als twee processors hetzelfde geheugen gebruiken, elke processor slechts aan halve snelheid kan werken. Bovendien is de evolutie van processor- en geheugentechnologie zo dat voor de nabije toekomst zeker geen verbetering wordt verwacht in deze situatie, integendeel. Opdat een multiprocessor sneller zou kunnen werken dan een uniprocessor wordt daarom bij elke processor lokaal geheugen aangebracht met daarin een kopie van een gedeelte van het gemeenschappe-

lijk geheugen, de zgn. cache. Systeemsoftware of daartoe ontworpen hardware zorgt er dan voor dat een wijziging van een kopie van een locatie transparant ook op de andere kopieën van die locatie wordt toegepast, m.a.w. dat de caches consistent blijven. Het komt er hierbij op aan minder communicatie te genereren tussen de lokale geheugens dan er in een systeem zonder lokale geheugens gegenereerd wordt tussen de verschillende processors en het gemeenschappelijk geheugen.

Naast de gemeenschappelijk-geheugenabstractie bestaat ook de berichtendoorgaveabstractie. Berichtdoorgave houdt in dat elke processor enkel lokaal geheugen heeft, en dat uitwisseling van data gebeurt door expliciete opdrachten in de verschillende processen. Deze data worden dan via een gemeenschappelijk communicatiekanaal, door de processors of door andere hardware die de consistentie beheert, tussen de lokale geheugens uitgewisseld. Berichtendoorgave heeft als voordelen op gemeenschappelijk geheugen dat het eenvoudiger implementeerbaar is en ook dat het toelaat efficiëntere applicaties te schrijven als het communicatiepatroon eenvoudig voorspelbaar is. Gemeenschappelijk geheugen daarentegen heeft het grote voordeel eenvoudiger programmeerbaar te zijn, en is daarnaast ook beter geschikt voor applicaties met een moeilijker te voorspellen communicatiepatroon. Omwille van de extra communicatie die het gebruik van gemeenschappelijk geheugen kan meebrengen t.o.v. berichtendoorgave, werd reeds uitgebreid onderzoek verricht naar technieken om de gegenereerde communicatie tussen de lokale geheugens te verminderen. Daartoe werden uiteenlopende technieken toegepast: de reeds vermelde cachegeheugens, schrijfbuffers, herordening van lees- en schrijfopdrachten door de processors en het verzwakken van de consistentiegaranties. Om de nodige vrijheid te behouden bij de implementatie van deze technieken en ook omwille van de portabiliteit wordt naar de programmeur toe niet de implementatie van een geheugensysteem maar wel het gedrag ervan gespecificeerd. Deze gedragsbeschrijving krijgt gewoonlijk de naam geheugenmodel.

Binnen een computerarchitectuur kunnen op verschillende abstractieniveaus geheugenmodellen worden gedefinieerd. Als in een geheugenmodel wordt gewerkt met deelopdrachten van lees- en schrijfopdrachten, dan spreekt met van een hardwaregericht (*hardware-centric*) geheugenmodel. Een deelopdracht verschilt van een opdracht doordat een deelopdracht een enkel lokaal geheugen adresseert, terwijl een opdracht gericht is naar het gemeenschappelijk geheugen als één systeem. Deelopdrachten zijn niet zichtbaar in een architectuur, opdrachten wel.

Wordt er in tegenstelling tot bij de *hardware-centric* geheugenmodellen met de lees- en schrijfopdrachten zelf gewerkt, dan is er sprake van een programmeurgerichte (*programmer-centric*) aanpak. In dit hoofdstuk wordt de laatste aanpak gevolgd.

Er bestaat een grote verscheidenheid niet alleen in de beschrijvingswijzen van geheugenmodellen maar ook in de bouwstenen die gebruikt worden in deze geheugenmodellen. Om geheugenmodellen met elkaar te kunnen vergelijken is er nood aan een set gemeenschappelijke bouwstenen en een uniforme beschrijvingswijze. In dit hoofdstuk wordt een dergelijke beschrijvingswijze voor geheugenmodellen voorgesteld.

Geheugenmodellen hebben een ruimer toepassingsgebied dan alleen multiprocessors. Deze zijn ook toepasbaar op meerdradige programma's die worden uitgevoerd door één of meer processors, op programma's die gebruik maken van gedistribueerd gemeenschappelijk geheugen op een netwerk van werkstations (*distributed shared memory* of DSM), op gedistribueerde databanken en op netwerkbestandssystemen. Bovendien kan het voorgestelde formalisme ook toegepast worden om berichtencommunicatiemodellen te beschrijven.

Naast de beschrijving van geheugenmodellen en berichtencommunicatiemodellen en de studie van hun eigenschappen, moet het verband tussen een model en de implementatie ervan kunnen worden gelegd. Omwille van de complexiteit van bestaande implementaties van geheugensystemen en berichtencommunicatiesystemen dient dit bij voorkeur op formele wijze te gebeuren. De in dit hoofdstuk voorgestelde aanpak is bij uitstek geschikt voor behandeling in een bewijsverificatiesysteem zoals Mural [BR91, BFL$^+$94, Bic98, JJLM91] of PVS [ORS92]. Voor de uitwerking van de in dit hoofdstuk voorgestelde aanpak met Mural, zie [Sla96, SVAH98].

Dit hoofdstuk is als volgt ingedeeld. In paragrafen 3.2 en 3.3 worden de ordeningsrelaties ingevoerd die als basis zullen dienen voor het beschrijven van geheugenmodellen. Bij geheugenmodellen met zwakke garanties qua ordening is het nodig per processor expliciet een ordening te kunnen opleggen. Daarvoor dienen etiketten, die besproken worden in paragraaf 3.4. De notatie die gehanteerd zal worden voor de verschillende soorten opdrachten wordt vastgelegd in paragraaf 3.5. Er zijn een aantal eigenschappen die voor alle geheugenmodellen gemeenschappelijk zijn, en ook een aantal eigenschappen die bij meerdere maar niet bij alle geheugenmodellen voorkomen. Deze worden be-

sproken in paragrafen 3.6 resp. 3.7. Het ingevoerde formalisme wordt in paragraaf 3.8 toegepast om een aantal bestaande geheugenmodellen te beschrijven. In de volgende paragraaf, nl. 3.9, wordt aangetoond dat het voorgestelde formalisme niet alleen geschikt is om geheugenmodellen te beschrijven maar ook om berichtencommunicatiemodellen te beschrijven. In paragraaf 3.10 worden de ingevoerde geheugenmodellen onderling gerangschikt. In de volgende paragraaf, 3.10, wordt via enkele voorbeelden het verband behandeld tussen geheugenmodel en implementatie, en in paragraaf 3.12 wordt verwant onderzoek rond geheugenmodellen besproken. Tenslotte volgt in paragraaf 3.13 het besluit van dit hoofdstuk.

## 3.2 Sequentiële programma's en opdrachten

De uitvoering van een **sequentieel programma** geschreven in een procedurale programmeertaal gebeurt ofwel door het programma eerst te vertalen via een compiler naar een uitvoerbare vorm, ofwel door het programma met een interpreter uit te voeren. Tijdens de uitvoering van een sequentieel programma worden berekeningen uitgevoerd, worden data uit het geheugen gelezen en er naar geschreven, en worden ook beslissingen genomen over welk deel van het programma als volgende wordt uitgevoerd. Voorlopig onderstellen we dat de interactie met het geheugen alleen bestaat uit lees- en schrijfbewerkingen. Een essentiële eigenschap van sequentiële programma's is dat uit een opdracht, voor een leesopdracht in combinatie met de uit het geheugen gelezen waarde, ondubbelzinnig volgt welke opdracht als volgende zal worden uitgevoerd.

Het **geheugen** waarmee een sequentieel programma interageert bestaat uit een eindig aantal locaties. Elke locatie kan een eindig aantal waarden aannemen. Een variabele heeft een naam en wordt met één locatie geassocieerd. Er wordt ondersteld dat de inhoud van het geheugen alleen wordt veranderd door de schrijfopdrachten uit het programma. Het resultaat van een leesopdracht is de waarde geschreven naar dezelfde locatie door de meest recente schrijfopdracht. De betekenis van *recent* wordt verder nauwkeurig vastgelegd via de programmaordeningsrelatie $\xrightarrow{\text{po}}$.

In dit hoofdstuk worden enkel programma's beschouwd die onveranderd blijven tijdens hun uitvoering, m.a.w. *self-modifying-code* wordt niet beschouwd.

De interactie van een sequentieel programma met het geheugen tijdens een particuliere uitvoering van dat programma kan worden samengevat als een lijst van viertallen (opdrachttype, volgnummer, adres, waarde), waarbij het opdrachttype ofwel *leesopdracht* ofwel *schrijfopdracht* is. Zo'n viertal krijgt de naam **opdracht**. Voor deze viertallen gebruiken we de notatie $l_i(m, v_l)$ en $s_i(m, v_s)$. Daarin staat $l$ voor leesopdracht, $s$ voor schrijfopdracht, $m$ voor locatie, $v_l$ voor de door een leesopdracht gelezen waarde, $v_s$ voor de door een schrijfopdracht geschreven waarde, en is $i$ het volgnummer. Een uitvoering van een sequentieel programma wordt ondubbelzinnig vastgelegd door de lijst van uitgevoerde opdrachten. De verzameling uitgevoerde opdrachten wordt aangeduid met *Op*. De opdrachtenvolgorde wordt voorgesteld met de programmaordeningsrelatie $\xrightarrow{\text{po}}$, die dus een totale ordening is in *Op*. Een uitvoering van een programma kan dus ondubbelzinnig voorgesteld worden via het tweetal $E = (Op, \xrightarrow{\text{po}})$. Tenzij anders wordt vermeld zijn alle relaties reflexief, anti-symmetrisch en transitief.

De sequentie bestaande uit alle waarden $v_l$ uit de leesopdrachten in volgorde van uitvoering is de **sequentie van gelezen waarden**. We beschouwen alleen programma's waarvoor de sequentie van gelezen waarden de uitvoering van een programma $E = (Op, \xrightarrow{\text{po}})$ volledig vastlegt. Voor deze programma's ligt de uitvoering van elke één-ingang, één-uitgangstructuur binnen een programma vast als de sequentie van gelezen waarden voor deze deelstructuur gekend is. Structuren met één ingang en één uitgang worden ook aangeduid met de naam D-structuren, waarbij D staat voor Dijkstra [BS72].

Voor sequentiële programma's geschreven in de assembleertaal van een RISC-processor met hoogstens één geheugenoperand per instructie legt de sequentie van gelezen waarden de programma-uitvoering eenduidig vast. De voorwaarde dat de sequentie van gelezen waarden de programma-uitvoering volledig vastlegt wordt echter niet door alle sequentiële programma's vervuld. Voor programma's in een hogere programmeertaal kan elk gebruik of verandering van de waarde van een variabele een geheugentoegang veroorzaken, maar kan dat gebruik of die verandering evengoed een toegang naar een register veroorzaken. Bovendien wordt in de definitie van een programmeertaal doorgaans niet vastgelegd in welke volgorde de operandi van commutatieve bewerkingen moeten worden geëvalueerd. Om te kunnen afleiden welke lees- en schrijfopdrachten worden gegenereerd moet dus gekend zijn welke geheugenopdrachten door optimalisatie werden geëlimineerd,

en in welke volgorde operandi van commutatieve bewerkingen worden geëvalueerd. In bepaalde programmeertalen kunnen optimalisaties van geheugenopdrachten per variabele uitgeschakeld worden. In de talen C, C++ en Java bijvoorbeeld is daartoe het sleutelwoord volatile beschikbaar. In de voorbeelden in dit hoofdstuk zal worden ondersteld dat bij commutatieve bewerkingen operandi van links naar rechts worden geëvalueerd, en dat geen lees- of schrijfopdrachten werden geëlimineerd.

Voor D-structuren waarbij alle gelezen variabelen ook binnen de D-structuur worden geïnitialiseerd volgt de sequentie van gelezen waarden uit de D-structuur zelf. Voor niet-geïnitialiseerde variabelen bevat de sequentie van gelezen waarden de beginwaarde van deze variabelen. De uitvoering van een programma $E = (Op, \xrightarrow{\text{po}})$ volgt dus uit de combinatie van de gelezen waarden en het programma zelf.

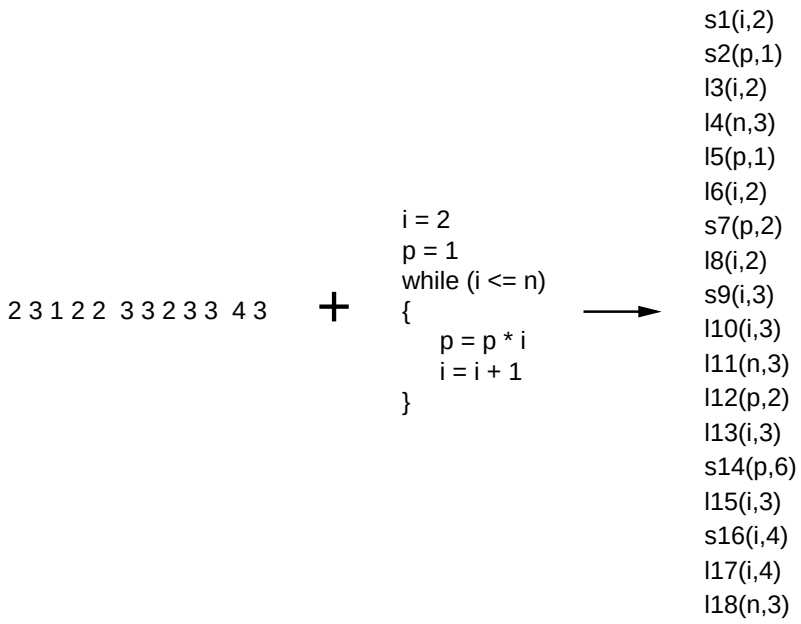
Het tot stand komen van de sequentie opdrachten $E = (Op, \xrightarrow{\text{po}})$ door een sequentieel programma uit te voeren geven we de naam **proces**. Een **processor** is een in hardware uitgevoerde interpreter. Doorgaans bestaat een proces uit de uitvoering van een sequentieel programma door een processor.

Een voorbeeld van een sequentieel programma, de uit het geheugen gelezen waarden en de resulterende opdrachten staat in figuur 3.1. Het programma berekent voor een getal $n$ de waarde $n! = 1 \times 2 \ldots \times n$. Er werd ondersteld dat de variabele $n$ vooraf werd geïnitialiseerd met de waarde 3. Omwille van de leesbaarheid staan in de sequentie gelezen waarden bredere spaties tussen de laatste waarde gelezen in een iteratie en de eerste waarde gelezen in de volgende iteratie.

## 3.3 Parallelle programma's en geheugenordening

Met de benaming **gemeenschappelijk-geheugensysteem** duiden we een systeem aan dat een gegeven expliciet parallel programma kan uitvoeren. Een parallel programma bestaat uit een vast aantal $n$ sequentiële programma's die dezelfde adresruimte gebruiken. Er wordt ondersteld dat het gemeenschappelijk-geheugensysteem alleen informatie uit de buitenwereld kan opnemen en informatie naar buiten kan brengen via geheugenopdrachten. Dit mechanisme staat ook bekend onder de naam *memory-mapped* in- en uitvoer.

Het geheugen in een gemeenschappelijk-geheugensysteem bestaat




**Figuur 3.1:** Verband tussen de begrippen sequentie van gelezen waarden, sequentieel programma en uitgevoerde opdrachten.

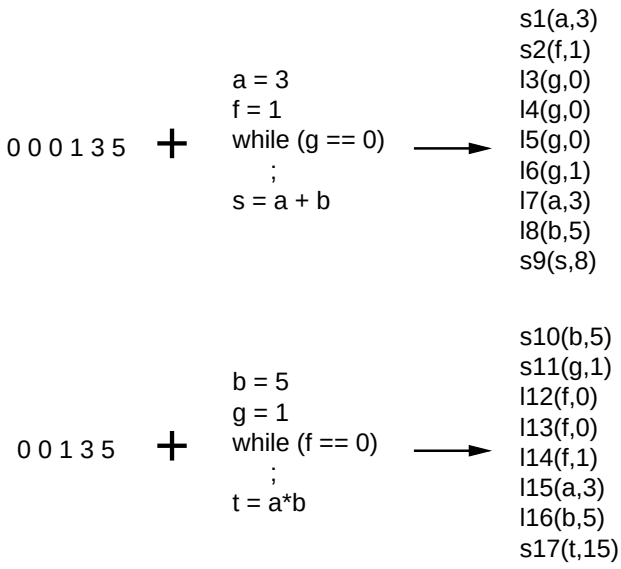
uit een aantal locaties, en op elke locatie kan een lees- of schrijfbewerking worden uitgevoerd. Het precieze verband tussen het resultaat van een leesopdracht en de waarden geschreven door schrijfopdrachten naar dezelfde locatie hangt af van het geheugenmodel.

De uitvoering van een parallel programma bestaat uit één proces per sequentieel programma dat deel uitmaakt van het parallelle programma. Met elk sequentieel programma $p$, waarbij $1 \leq p \leq n$, worden een eigen sequentie gelezen waarden, een eigen opdrachtenstroom $Op_p$ en een eigen programmaordening $\xrightarrow{\text{po}\,p}$ geassocieerd. De opdrachtenverzameling $Op$ voor een parallel programma is de unie van de opdrachtenverzamelingen $Op_p$ van alle processen. De programmaordening $\xrightarrow{\text{po}}$ voor een parallel programma is de unie van de afzonderlijke programmaordeningen $\xrightarrow{\text{po}\,p}$ per proces. Het begrip programmaordening zoals hier gedefinieerd werd reeds eerder ingevoerd door Shasha [SS88].

Voor processen met in- en/of uitvoer modelleren we het effect van de in- en uitvoerapparaten op het gemeenschappelijk geheugen als schrijf- resp. leesbewerkingen in een afzonderlijk proces van het parallelle programma.

Een **gemeenschappelijk geheugen** is een geheugen zoals gedefinieerd in de vorige paragraaf, waarbij alle processen toegang hebben tot de beschikbare locaties. Het resultaat van een leesopdracht door proces $p$ is nog steeds de waarde geschreven naar dezelfde locatie door de meest recente schrijfopdracht. De betekenis van *recent* wordt bepaald door de **geheugenordeningsrelaties** $\xrightarrow{\text{mo}\,p}$. Elke geheugenordeningsrelatie $\xrightarrow{\text{mo}\,p}$ is een partiële ordening van alle opdrachten.

Stel dat voor een parallel programma per proces de sequenties van gelezen waarden gekend zijn. Terwijl tijdens de uitvoering van een sequentieel programma dat programma zelf de enige oorzaak is van veranderingen in het geheugen, wordt tijdens de uitvoering van een proces als onderdeel van een parallel programma de inhoud van het geheugen ook veranderd door de andere processen. Toch geldt de eigenschap dat de stromen van gelezen waarden voor een parallel programma de uitvoering van het volledige parallelle programma $E = (Op, \xrightarrow{\text{po}})$ eenduidig bepalen. Voor een parallel programma bestaan er dus afhankelijkheden tussen de per-proces sequenties van gelezen waarden. Veranderingen van gemeenschappelijke variabelen door één proces kunnen terug te vinden zijn in de sequentie met gelezen waar-




**Figuur 3.2:** Verband tussen de begrippen sequenties van gelezen waarden, parallel programma en uitgevoerde opdrachten.

den van een ander proces. Dit is trouwens de essentie van gemeenschappelijk geheugen.

In figuur 3.2 is een voorbeeld van een parallel programma opgenomen. In de twee processen die samen de uitvoering van het parallel programma realiseren wordt eerst aan de variabelen $a$ en $b$ een waarde gegeven. Zoals uit de sequentie gelezen waarden van beide processen blijkt wordt er ondersteld dat de variabelen $f$ en $g$ op nul geïnitialiseerd werden. Beide processen veranderen elk één van deze variabelen en wachten dan tot de verandering van de andere variabele door het andere proces wordt opgemerkt. De twee lussen zorgen dus voor synchronisatie. Daarna worden de waarden van variabelen $a$ en $b$ gelezen en toegepast in verdere berekeningen.

Voor een sequentieel programma volstaat één totale ordening, nl. $\xrightarrow{\text{po}}$, om de resultaten van leesopdrachten te verklaren. Bij een parallel programma is dit niet langer het geval. Om de resultaten van de leesopdrachten van een gegeven proces $p$ te verklaren moet naast de schrijfopdrachten door het eigen proces ook met de schrijfopdrachten van alle andere processen rekening worden gehouden. Een relatie die de resultaten van de leesopdrachten van proces $p$ verklaart door de schrijfop-

drachten van alle processen te ordenen geven we de naam **geheugenordening**, en duiden we aan met het symbool $\xrightarrow{\text{mo}\,p}$. Per geheugenmodel worden aan de geheugenordeningsrelaties bijkomende eigenschappen opgelegd. In het algemeen is voor een gegeven programma-uitvoering $E = (Op, \xrightarrow{\text{po}})$ zo'n $\xrightarrow{\text{mo}\,p}$-relatie niet uniek. Daar $E$ alle informatie bevat over een programma-uitvoering die op architecturaal niveau waarneembaar is, zijn de geheugenordeningsrelaties dus niet noodzakelijk waarneembaar. Het begrip architectuur wordt hier in dezelfde betekenis gebruikt als in de eerste paragraaf van hoofdstuk 2. Meestal zijn deze geheugenordeningsrelaties sterk verwant aan de implementatie van het geheugenmodel. In paragraaf 3.11 worden enkele voorbeelden gegeven van implementaties van geheugensystemen en het verband met de geheugenordeningsrelaties voor die systemen.

Uit paragraaf 2.4 volgt dat een parallel programma zonder algemene races intern deterministisch wordt uitgevoerd. Dat betekent dat voor een dergelijk programma de uitvoering $E = (Op, \xrightarrow{\text{po}})$ volledig door het programma zelf wordt bepaald.

Alhoewel de volledige geheugenordeningsrelaties op het architecturale niveau niet waarneembaar zijn, kunnen we uit een uitvoering $E$ wel een deel van de geheugenordeningen afleiden. Uit de regel dat zowel voor relaties $\xrightarrow{\text{mo}\,1}$ als $\xrightarrow{\text{mo}\,2}$ een leesopdracht de waarde van de meest recente schrijfopdracht moet retourneren kunnen we voor voorbeeld 3.2 afleiden dat deze volgordes zeker moeten gelden: $l_3 \xrightarrow{\text{mo}\,1} s_{11}$, $l_4 \xrightarrow{\text{mo}\,1} s_{11}$, $l_5 \xrightarrow{\text{mo}\,1} s_{11}$, $s_{11} \xrightarrow{\text{mo}\,1} l_6$ voor $g$, $s_{10} \xrightarrow{\text{mo}\,1} l_8$ voor $b$, $l_{12} \xrightarrow{\text{mo}\,2} s_2$, $l_{13} \xrightarrow{\text{mo}\,2} s_2$, $s_2 \xrightarrow{\text{mo}\,2} l_14$ voor $f$ en $s_1 \xrightarrow{\text{mo}\,2} l_{15}$ voor $a$.

## 3.4 Etiketten en synchronisatie

Een processor heeft afhankelijk van het geheugenmodel een zekere vrijheid in het herordenen van lees- en schrijfopdrachten. In bepaalde omstandigheden echter is deze herordening ongewenst en is het nodig deze te verhinderen. Dit is de bestaansreden van etiketten. Gharachorloo voerde samen met het geheugenmodel release-consistentie volgende etiketten (*labels*) in: *ordinary*, *nsync*, *acquire* en *release* [GLL+90]. Het etiket *ordinary* geeft geen garanties qua behoud van volgorde, terwijl de andere drie etiketten respectievelijk het behoud garanderen van de programmaordening t.o.v. alle andere lees- en schrijfopdrachten uitgevoerd door dezelfde processor, enkel t.o.v. de opdrachten na of enkel

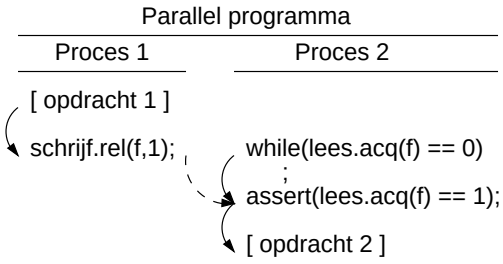
t.o.v. de opdrachten vóór de opdracht voorzien van het etiket. De aanduidingen *vóór* en *na* verwijzen naar alle opdrachten die in programmaordening vóór resp. na de beschouwde opdracht komen, tot aan het begin resp. einde van het beschouwde proces. In deze tekst worden de vier etiketten aangeduid met de benamingen $ord_L$, $sync_L$, $acq_L$ en $rel_L$, met dezelfde betekenis als bij Gharachorloo. De index $L$ is de eerste letter van *label*.

Naast lees- en schrijfopdrachten beschouwen we ook opdrachten van het type lees/wijzig/schrijf-opdrachten. Elke lees- of schrijfopdracht heeft één etiket, terwijl bij elke lees/wijzig/schrijf-opdracht twee etiketten horen. Bij een lees/wijzig/schrijf-opdracht wordt een eerste etiket met de leesopdracht ervan geassocieerd en een tweede etiket met de schrijfopdracht ervan. Er zijn geen andere opdrachten waaraan een etiket wordt toegekend. Lees- en schrijfopdrachten met het etiket $ord_L$ en ook de lees/wijzig/schrijf-opdrachten met twee etiketten $ord_L$ behoren tot de *gewone opdrachten* (*ordinary accesses*), terwijl opdrachten met minstens één etiket uit $sync_L$, $acq_L$ of $rel_L$ behoren tot de *bijzondere opdrachten* (*special accesses*).

Alhoewel deze etiketten slechts de uitvoeringsvolgorde op één processor vastleggen, is het toch mogelijk met deze opdrachten synchronisatie tussen processen te realiseren. Een voorbeeld hiervan staat in figuur 3.3: uit het voorgestelde programma volgt dat de leesopdracht met het acquire-etiket na de schrijfopdracht met het release-etiket zal worden uitgevoerd. Het is daarvoor essentieel dat de leesopdracht met het acquire-etiket in een lus is geplaatst: op die manier bestaat er zekerheid over de geheugenordening van de opdrachten met release- en acquire-etiketten. Dit voorbeeld werkt zonder aanpassing op alle verder beschouwde geheugenmodellen.

De naam van het geheugenmodel *release-consistentie* verwijst naar een mogelijke implementatie van dit geheugenmodel, nl. dat veranderingen in de lokale geheugens pas aan de andere processors worden bekendgemaakt op het ogenblik dat een opdracht voorzien van een $rel_L$-etiket wordt uitgevoerd.

Daar de $acq_L$- en $rel_L$-etiketten behoud van programmaordening in één richting garanderen, en het $sync_L$-etiket behoud van programmaordening in beide richtingen garandeert, garandeert het $sync_L$-etiket meer dan de $acq_L$- en $rel_L$-etiketten. Voor intern deterministische programma's blijft het resultaat gelijk indien $acq_L$- en $rel_L$-etiketten door $sync_L$-etiketten worden vervangen. Zie ook paragraaf 2.4.




**Figuur 3.3:** Voorbeeld van synchronisatie tussen draden m.b.v. lees- en schrijfopdrachten voorzien van etiketten. De pijlen in volle lijn stellen volgordes voor opgelegd door de etiketten $acq_L$ en $rel_L$, en de pijl in streeplijn stelt de volgorde voor opgelegd door het wachten op de waardeverandering van f. Er wordt ondersteld dat variabele f initieel nul is.

Verder hebben bij een leesopdracht alleen de acquire- en sync-etiketten zin, en hebben bij een schrijfopdracht alleen de release- en sync-etiketten zin. Voor correct werkende uniprocessors is het effect van etiketten pas waarneembaar op architecturaal niveau als etiketten worden toegepast om een volgorde tussen opdrachten van verschillende processen op te leggen. Op basis van het resultaat van een leesopdracht kan worden nagegaan of een schrijfopdracht op een andere processor reeds werd uitgevoerd, zoals voorgesteld in figuur 3.3. Als die leesopdracht een acquire- of sync-etiket heeft, dan zijn ook de opdrachten die in programmaordening na de leesopdracht komen wegens transitiviteit geordend t.o.v. de schrijfopdracht. Heeft de leesopdracht daarentegen een release-etiket, dan zijn er geen garanties voor de geheugenordening van de schrijfopdracht voorafgaand aan de leesopdracht en de opdrachten die in programmaordening volgen op de leesopdracht. Vandaar dat alleen acquire- en sync-etiketten zin hebben bij een leesopdracht. De redenering voor schrijfopdrachten is analoog.

In tegenstelling tot lees- en schrijfopdrachten zijn er met elke lees/wijzig/schrijf-opdracht twee etiketten geassocieerd: een eerste etiket dat hoort bij de leesopdracht, en een tweede etiket dat hoort bij de schrijfopdracht. Opdat een lees/wijzig/schrijf-opdracht zinvol zou zijn moet de leesopdracht vóór de schrijfopdracht uitgevoerd worden. Afhankelijk van de gewenste volgorde kan dus aan de lees/wijzig/schrijf-opdracht een combinatie van twee van de etiketten $sync_L$, $acq_L$ en $rel_L$ toegekend worden.

De synchronisatie uit het programma uit figuur 3.3 zorgt ervoor dat de code aangeduid met [ opdracht 2 ] uitgevoerd wordt na [ opdracht 1 ]. Deze synchronisatie werd bereikt door in een lus leesopdrachten uit te voeren. In het algemeen geldt dat indien enkel lees- en schrijfopdrachten en geen synchronisatieopdrachten beschikbaar zijn, dan een lus met leesopdrachten noodzakelijk is om een volgorde tussen opdrachten op verschillende processors te kunnen opleggen. Tijdens een dergelijke lus met leesopdrachten is er dus continu communicatie tussen de betreffende processor en het geheugen, wat zijn invloed heeft op de prestaties van andere processors. Om dergelijke aanhoudende communicatie met het geheugen te kunnen vermijden tijdens interprocessorsynchronisatie, werden in verschillende geheugenmodellen expliciete synchronisatieopdrachten opgenomen. Geheugenmodellen voor DSM-systemen bijvoorbeeld voorzien hierin. Gewoonlijk zijn deze synchronisatieopdrachten equivalenten van Dijkstra's P- en V- opdrachten op semaforen [Dij65], samen met barrière-synchronisatie over meerdere processen.

## 3.5 Gehanteerde notatie

De opdrachttypes lees-, schrijf- en lees/wijzig/schrijf-opdrachten werden reeds eerder ingevoerd. We breiden deze opdrachttypes uit met lock-, unlock- en grensopdrachten. Deze **opdrachten** worden aangeduid met de respectieve symbolen $l$, $s$, $f$, *lock*, *unlock* en *bar*. Daarbij staat $l$ voor leesopdracht of *load*, $s$ voor schrijfopdracht of *store* en is $f$ de eerste letter van *fetch-and-add*, één van de mogelijke lees/wijzig/schrijfopdrachten. De afkorting *bar* staat voor *barrier* of grensopdracht. De laatste drie opdrachttypes, nl. *lock*, *unlock* en *bar* vormen gezamenlijk de **synchronisatieopdrachten**. De betekenis van de synchronisatieopdrachten lock en unlock volgt in paragraaf 3.6.2.

Een leesopdracht leest een waarde uit een locatie in het gemeenschappelijk geheugen, en een schrijfopdracht schrijft een waarde naar een locatie in het gemeenschappelijk geheugen. Een lees/wijzig/schrijfopdracht leest een waarde uit een locatie in het geheugen, wijzigt deze waarde, en schrijft die gewijzigde waarde naar dezelfde locatie. Andere lees- of schrijfopdrachten naar dezelfde locatie zijn niet toegelaten tussen de lees- en schrijfopdracht van een lees/wijzig/schrijf-opdracht.

Elke lees- en schrijfopdracht heeft één geassocieerd **etiket** $\lambda_1$ dat specificeert welke soort herordening van de opdracht toegelaten is. El-

ke lees/wijzig/schrijf-opdracht heeft twee geassocieerde etiketten $\lambda_1$ en $\lambda_2$. Een etiket is één van de vier mogelijke etiketten $ord_L$, $acq_L$, $rel_L$ of $sync_L$. De precieze betekenis van deze etiketten werd vastgelegd in paragraaf 3.6.

De opdrachten aangeduid met *bar* zijn de **grensopdrachten**. Deze opdrachten ordenen ofwel lees- ofwel schrijfopdrachten voor de grensopdracht met ofwel de lees- ofwel de schrijfopdrachten na de grensopdracht. Op basis hiervan worden vier types grensopdrachten onderscheiden: $bar_{ll}$, $bar_{ls}$, $bar_{sl}$ en $bar_{ss}$. De twee indices verwijzen naar het type opdracht dat door de grensopdracht wordt geordend: l staat voor leesopdracht, s voor schrijfopdracht. De eerste index verwijst naar de opdrachten vóór de grensopdracht, de tweede index naar de opdrachten na de grensopdracht. Een verdere besprekring van grensopdrachten volgt in paragraaf 3.6.3.

Het aantal processors wordt aangeduid met $n \in \mathbb{N}$, en het processornummer met $p \in P$. Het symbool $P$ staat voor de verzameling van alle processornummers, met $P = \{1 \ldots n\}$. Elke opdracht heeft een **opdrachtnummer** $i \in \mathbb{N}$, dat toeneemt voor opdrachten die later komen in programmaordening. Aan de volgorde van opdrachtnummers van opdrachten uitgevoerd door verschillende processen wordt geen betekenis gehecht. Synchronisatieopdrachten hebben bovendien een **identificatienummer** $j \in \mathbb{N}$, dat bij elkaar horende synchronisatieopdrachten groepeert. Het **geheugen** bestaat uit een aantal **locaties** $m \in Mem$, waarbij *Mem* de verzameling van alle locaties is. Een verzameling locaties wordt aangeduid met $M \subset Mem$. Elke locatie bevat een **waarde** $v \in Val$, met *Val* de eindige verzameling van alle mogelijke waarden. Een overzicht van de notatie gehanteerd voor de verschillende soorten opdrachten staat in tabel 3.1. De hier ingevoerde lees/wijzig/schrijfopdracht zal dienen als basis om andere ondeelbare opdrachten te modelleren. De *fetch-and-increment* en *test-and-set* opdrachten bijvoorbeeld worden gemodelleerd door de respectieve opdrachten $f(m, v_l, v_l+1)$ en $f(m, v_l, 1)$. Daarbij is $v_l$ de waarde die zich op locatie $m$ bevond vóór de opdracht werd uitgevoerd.

Het resultaat van de functies $type()$, $lbl_1()$, $lbl_2()$, $num()$, $id()$, $proc()$, $mem()$, $val_l()$ en $val_s()$ zijn respectievelijk de waarden opdrachttype, $\lambda_1$, $\lambda_2$, $i$, $j$, $p$, $\{m\}$ of $M$, $v_l$ en $v_s$. De waarde $\lambda_2$ wordt gedefinieerd als $ord_L$ voor opdrachten waaraan slechts één etiket is geassocieerd. De verzamelingen met alle opdrachten van het type $s$, $l$, $f$, *lock*, *unlock*, $bar_{ll}$, $bar_{ls}$, $bar_{sl}$, $bar_{ss}$ voor de beschouwde uitvoering hebben als naam

| Opdrachtnaam | Algemene gedaante | Verkorte notatie |
|---|---|---|
| lees | $(l, \lambda_1, i, p, m, v_l)$ | $\mathrm{l}_i.\lambda_1(m,v_l)$ |
| schrijf | $(s, \lambda_1, i, p, m, v_s)$ | $\mathrm{s}_i.\lambda_1(m,v_s)$ |
| lees/wijzig/schrijf | $(f, \lambda_1, \lambda_2, i, p, m, v_l, v_s)$ | $\mathrm{f}_i.\lambda_1.\lambda_2(m,v_l,v_s)$ |
| lock | $(\mathit{lock}, i, p, M, j)$ | $\mathit{lock}_i(M,j)$ |
| unlock | $(\mathit{unlock}, i, p, M, j)$ | $\mathit{unlock}_i(M,j)$ |
| grensopdracht type $t$ | $(\mathit{bar}_t, i, p, M, j)$ | $\mathit{bar}_{t,i}(M,j)$ |

**Tabel 3.1:** Opdrachtnamen, algemene gedaante, en verkorte notatie voor de gedefinieerde opdrachttypes. Het processornummer is bij de verkorte notatie niet vermeld, maar zal blijken uit de context. Het type $t$ van een grensopdracht is één van $ll$, $ls$, $sl$ of $ss$.

respectievelijk $S$, $L$, $F$, *Lock*, *Unlock*, $\mathit{Bar}_{ll}$, $\mathit{Bar}_{ls}$, $\mathit{Bar}_{sl}$ en $\mathit{Bar}_{ss}$. De definitie van afgeleide verzamelingen die verder zullen worden gebruikt staat in tabel 3.2. Waar restricties opgelegd worden aan de volgorde van lees- of schrijfopdrachten, zullen meestal lees/wijzig/schrijfopdrachten behandeld worden als behorende zowel tot de lees- als tot de schrijfopdrachten. Concreet zal dit gebeuren door naar de verzamelingen $LF = L \cup F$ en $SF = S \cup F$ te verwijzen i.p.v. de verzamelingen $L$ en $S$.

In alle voorbeelden van uitvoeringen in dit hoofdstuk wordt ondersteld dat het geheugen op nul werd geïnitialiseerd. Ook wordt van elke partiële of totale ordening in figuren systematisch de transitieve reductie weergegeven, d.w.z. dat als er een pijl gaat van $a$ naar $b$, van $a$ naar $c$ en van $b$ naar $c$ dat dan de pijl van $a$ naar $c$ wordt weggelaten.

De benaming **uitvoering** van een programma wordt gebruikt om te verwijzen naar de verzameling uitgevoerde opdrachten *Op* geordend door de programmaordening $\xrightarrow{\text{po}}$ al of niet in combinatie met de geheugenordeningen $\xrightarrow{\text{mo}\,p}$. Een uitvoering wordt genoteerd als $E = (\mathit{Op}, \xrightarrow{\text{po}})$ of $E' = (\mathit{Op}, \xrightarrow{\text{po}}, \xrightarrow{\text{mo}\,1}, \ldots, \xrightarrow{\text{mo}\,n})$.

## 3.6 Basiseigenschappen geheugenmodellen

Er zijn zeven eigenschappen die gemeenschappelijk zijn voor elk geheugenmodel: drie eigenschappen die de betekenis van lees- en schrijfopdrachten vastleggen en vier eigenschappen die de betekenis van synchronisatieopdrachten vastleggen. Deze eigenschappen volgen hieron-

| Omschrijving | Formele definitie |
|---|---|
| Opdrachten uitgevoerd door processor p | $Op_p = \{op \in Op \mid proc(op) = p\}$ |
| Alle lees- en lees/wijzig/schrijf-opdrachten | $LF = L \cup F$ |
| Alle schrijf- en lees/wijzig/schrijf-opdrachten | $SF = S \cup F$ |
| Alle lees-, schrijf- en lees/wijzig/schrijf-opdrachten | $LSF = L \cup SF$ |
| Gewone lees-, schrijf- en lees/wijzig/schrijf-opdrachten | $Op_{ord} = \{op \in Op \mid lbl_1(op) = ord_L \wedge lbl_2(op) = ord_L\}$ |
| Bijzondere lees-, schrijf- en lees/wijzig/schrijf-opdrachten | $Op_s = Op \setminus Op_{ord}$ |
| Opdrachten i.v.m. minstens locatie $m$ | $A_m = \{op \in A \mid m \in mem(op)\}$ |
| Opdrachten uitgevoerd door processor $p$ | $A_p = \{op \in A \mid proc(op) = p\}$ |
| Opdrachten i.v.m. minstens locatie $m$ en uitgev. door proc. $p$ | $A_{m,p} = \{op \in A \mid m \in mem(op) \wedge proc(op) = p\}$ |
| Grensopdrachten voor voorafgaande leesopdrachten | $Bar_{l\cdot} = Bar_{ll} \cup Bar_{ls}$ |
| Grensopdrachten voor voorafgaande schrijfopdrachten | $Bar_{s\cdot} = Bar_{sl} \cup Bar_{ss}$ |
| Grensopdrachten voor leesopdrachten verderop | $Bar_{\cdot l} = Bar_{ll} \cup Bar_{sl}$ |
| Grensopdrachten voor schrijfopdrachten verderop | $Bar_{\cdot s} = Bar_{ls} \cup Bar_{ss}$ |
| Alle grensopdrachten | $Bar = Bar_{ll} \cup Bar_{ls} \cup Bar_{sl} \cup Bar_{ss}$ |
| Synchronisatieopdrachten | $Sync = Lock \cup Unlock \cup Bar$ |

**Tabel 3.2:** Afgeleide opdrachtenverzamelingen, waarbij $A$ staat voor een verzameling van opdrachten.

der.

### 3.6.1 Eigenschappen van lees- en schrijfopdrachten

De eigenschap dat de uitvoering van lees- en schrijfopdrachten door processor $p$ ook het resultaat zou kunnen zijn van een correct werkende uniprocessor is equivalent met de eigenschap dat data-afhankelijke opdrachten in programmaordening worden uitgevoerd. Als data-afhankelijke opdrachten door een processor in volgorde worden uitgevoerd, dan voldoet die processor aan **uniprocessorcorrectheid**. Formeel is deze eigenschap als volgt:

$$\begin{aligned}&\forall\, m \in \mathit{Mem} :\ \forall\, (op_1, op_2) \in (\mathit{LSF}_m^2 \setminus L_m^2) :\ \forall\, p \in P :\\ &\quad op_1 \xrightarrow{\text{po}\,p} op_2 \Longrightarrow op_1 \xrightarrow{\text{mo}\,p} op_2\end{aligned} \tag{3.1}$$

Indien een lees- of schrijfopdracht een ander etiket heeft dan $ord_L$, en dus behoort tot de bijzondere opdrachten, of als het gaat over synchronisatieopdrachten zijn er bijkomende restricties wat betreft de toegelaten geheugenordening. Deze restricties zijn aangegeven in tabel 3.3. Deze eigenschap heet de **volgorde van bijzondere en synchronisatieopdrachten**:

$$\begin{aligned}&\forall\, (op_1, op_2) \in \mathit{Op}^2 :\ \forall\, p \in P :\\ &\quad op_1 \xrightarrow{\text{po}} op_2 \wedge \text{in-order}(op_1, op_2) \Longrightarrow op_1 \xrightarrow{\text{mo}\,p} op_2,\\ &\text{waarbij}\\ &\quad \text{in-order}(op_1, op_2)\\ &\quad = (op_1 \in \mathit{Lock}\\ &\qquad \vee op_1 \in \mathit{Unlock}\\ &\qquad \vee op_2 \in \mathit{Lock}\\ &\qquad \vee op_2 \in \mathit{Unlock}\\ &\qquad \vee op_1 \in \mathit{Bar}_{.l} \wedge op_2 \in \mathit{LF}\\ &\qquad \vee op_1 \in \mathit{Bar}_{.s} \wedge op_2 \in \mathit{SF}\\ &\qquad \vee op_1 \in \mathit{LF} \wedge op_2 \in \mathit{Bar}_{l.}\\ &\qquad \vee op_1 \in \mathit{SF} \wedge op_2 \in \mathit{Bar}_{s.}\\ &\qquad \vee op_1 \in \mathit{Bar} \wedge op_2 \in \mathit{Bar}\\ &\qquad \vee \mathit{lbl}_1(op_1) \in \{\mathit{acq}_L, \mathit{sync}_L\} \wedge op_2 \in \mathit{LSF}\\ &\qquad \vee \mathit{lbl}_2(op_1) \in \{\mathit{acq}_L, \mathit{sync}_L\} \wedge op_2 \in \mathit{LSF}\\ &\qquad \vee op_1 \in \mathit{LSF} \wedge \mathit{lbl}_1(op_2) \in \{\mathit{rel}_L, \mathit{sync}_L\}\\ &\qquad \vee op_1 \in \mathit{LSF} \wedge \mathit{lbl}_2(op_2) \in \{\mathit{rel}_L, \mathit{sync}_L\}).\end{aligned} \tag{3.2}$$

Aan deze eigenschap wordt op triviale wijze voldaan als alle lees-

| $op_2 \rightarrow$ $\downarrow op_1$ | $ord_L$ | $acq_L$ | $rel_L$ | $sync_L$ | *bar* | *lock* | *unlock* |
|---|---|---|---|---|---|---|---|
| $ord_L$ | | | + | + | c | + | + |
| $acq_L$ | + | + | + | + | c | + | + |
| $rel_L$ | | | + | + | c | + | + |
| $sync_L$ | + | + | + | + | c | + | + |
| *bar* | c | c | c | c | + | + | + |
| *lock* | + | + | + | + | + | + | + |
| *unlock* | + | + | + | + | + | + | + |

**Tabel 3.3:** Tabelsgewijze voorstelling van de functie in-order(), de functie die weergeeft welke synchronisatieopdrachten en bijzondere lees- en schrijfopdrachten in programmaordening moeten worden uitgevoerd. In de tabel staat **+** voor waar, en een opengelaten vakje betekent onwaar. Het symbool **c** betekent dat het opgelegd zijn van een volgorde afhangt van de opdracht waaraan het etiket werd toegekend – zie ook de formele definitie van in-order(). De volgorde vermeld in deze tabel is van toepassing ongeacht het adres waar een opdracht naar verwijst. Indien twee opdrachten naar hetzelfde adres verwijzen dan legt uniprocessorcorrectheid nog bijkomende beperkingen op.

en schrijfopdrachten gewone opdrachten zijn, omdat het predikaat in-order() voor gewone opdrachten steeds onwaar is.

Het resultaat van een leesopdracht op een bepaalde locatie, uitgevoerd door processor $p$, is de waarde geschreven naar dezelfde locatie door de schrijfopdracht die onmiddellijk aan de leesopdracht voorafgaat. Hierbij is *voorafgaan* vastgelegd door de relatie $\xrightarrow{\text{mo}\,p}$. Er zijn twee ontaarde gevallen: er is geen voorafgaande schrijfopdracht, of er zijn er twee. Het laatste geval heeft de naam *data race*. Indien er geen voorafgaande schrijfopdracht is of indien minstens twee onmiddellijk voorafgaande schrijfopdrachten elk een andere waarde schrijven, dan is het resultaat van de leesopdracht niet gedefinieerd. Dit is de eigenschap **geheugenwerking**. Voor elke leesopdracht $l \in LF$ geldt:

$$\begin{aligned} p &\triangleq proc(l) \\ pred(l) &\triangleq \{s \in SF_{mem(l)} | s \neq l \wedge s \xrightarrow{\text{mo}\,p} l\} \\ impred(l) &\triangleq \{s \in pred(l) | \forall\, s' \in pred(l) :\ s = s' \vee \neg(s \xrightarrow{\text{mo}\,p} s')\} \\ \forall\, s_1 \in impred(l) :\ &(\forall\, s_2 \in impred(l) :\ s_1 = s_2) \Longrightarrow val_l(l) = val_s(s_1) \end{aligned} \tag{3.3}$$

In deze definitie is $p$ de processor waarop de leesopdracht werd uitgevoerd. Voor de leesopdracht $l$ is $m = mem(l)$ de locatie waaruit gelezen werd, en is $pred(l)$ (*predecessors*) de verzameling van alle schrijfopdrach-

ten naar locatie $m$ die $l$ voorafgaan volgens geheugenordening $\xrightarrow{mo\,p}$. De verzameling *impred*$(l)$ (*immediate predecessors*) bevat de schrijfopdrachten die onmiddellijk aan $l$ voorafgaan. Er worden op basis van het aantal elementen in de verzameling *pred*$(l)$ drie gevallen onderscheiden: ofwel bevat de verzameling *pred*$(l)$ één element, ofwel is deze verzameling leeg, ofwel bevat deze verzameling meerdere elementen. Als de verzameling *pred*$(l)$ één element $s \in$ *pred*$(l)$ bevat, dan volgt uit bovenstaande definitie dat de leesopdracht $l$ de waarde leest die geschreven werd door de schrijfopdracht $s$. In het tweede geval, waarbij de verzameling *pred*$(l)$ leeg is en er dus geen schrijfopdrachten aan de leesopdracht $l$ voorafgaan, is volgens bovenstaande definitie de waarde gelezen door opdracht $l$ nog steeds een geldige waarde uit de verzameling *Val* maar is er verder niets over deze waarde gekend. In het geval dat er meerdere schrijfopdrachten, nl. deze in *pred*$(l)$, aan de leesopdracht $l$ voorafgaan zijn er twee deelgevallen. Als alle schrijfopdrachten in *pred*$(l)$ dezelfde waarde schrijven, dan wordt deze waarde ook door leesopdracht $l$ gelezen. Zoniet, dan is opnieuw de waarde gelezen door leesopdracht $l$ een geldige waarde uit *Val* maar verder niet gedefinieerd.

Er volgt onmiddellijk dat de waarde van alle leesopdrachten $l$ met betrekking tot de locatie $m \in$ *Mem*, uitgevoerd door een gegeven processor $p$, volledig bepaald wordt door de relatie $\xrightarrow{mo\,p} \cap (SF_m^2 \cup (SF_m \times LF_{m,p}))$. Of ook:

> De waarde van alle leesopdrachten $l \in LF_{m,p}$ wordt volledig bepaald door $\xrightarrow{mo\,p} \cap (SF_m^2 \cup (SF_m \times L_{m,p}))$.

### 3.6.2 Lock en Unlock

Volgende equivalentierelatie partitioneert *Lock* $\cup$ *Unlock* in corresponderende lock- en unlock-opdrachten:

$$N' \triangleq \{(op_1, op_2) \in (Lock \cup Unlock)^2 | id(op_1) = id(op_2)\}$$

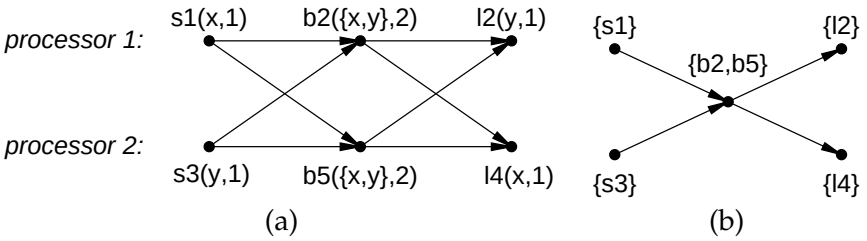
We definiëren het begrip kritische sectie als de equivalentieklassen van de partitionering van de opdrachtenverzameling *Lock* $\cup$ *Unlock* volgens de relatie $N'$. Met elke kritische sectie is er een uniek identificatienummer $j$ geassocieerd. Aan een kritische sectie met identificatienummer $j$ leggen we op dat deze moet bestaan uit één lock-opdracht $l$ en nul of

één unlock-opdrachten $u$, waarbij de lock-opdracht in programmaordening voor de unlock-opdracht moet komen. Indien er voor een lock-opdracht geen corresponderende unlock-opdracht bestaat, dan wordt de unlock-opdracht ondersteld zich voorbij het einde van proces $p = \mathit{proc}(l)$ te bevinden. Verder leggen we op dat de geassocieerde locatieverzamelingen dezelfde moeten zijn: $M = \mathit{mem}(l) = \mathit{mem}(u)$. Met elke kritische sectie is dus een verzameling locaties $M$ geassocieerd. Met elke kritische sectie kan er een verzameling opdrachten geassocieerd worden: $K(l,u) = \{op | l \xrightarrow{\text{po}} op \xrightarrow{\text{po}} u\}$ of $K(l) = \{op | l \xrightarrow{\text{po}} op\}$, naargelang al of niet een unlock-opdracht deel uitmaakt van de kritische sectie. Afhankelijk van de context zal met de benaming kritische sectie ofwel naar de lock- en unlock-opdracht verwijzen, ofwel naar de geassocieerde opdrachtenverzameling.

Naargelang de keuze die gemaakt wordt voor de locaties in verzameling $M$ krijgen de lock- en unlock-opdrachten een andere betekenis. Indien de verzameling $M$ steeds één enkele locatie $m$ bevat, en deze locatie in geen enkele lees- of schrijfopdracht voorkomt, dan gedragen de lock- en unlock-opdrachten zich als de klassieke P- en V-opdrachten voor binaire semaforen. Met elke locatie $m$ wordt dan een binaire semafoor geassocieerd. Een andere toepassingswijze voor kritische secties bestaat erin de verzameling $M$ geassocieerd met kritische sectie $j$ uitsluitend samen te stellen uit locaties waarnaar waarden geschreven worden of waaruit waarden gelezen worden door de opdrachten die in programmaordening tussen de lock-opdracht $l$ en de unlock-opdracht $u$ komen. In dit geval is er een rechtstreeks verband tussen een kritische sectie en de locaties die er door afgeschermd worden.

We voeren de quotientoperator / in voor verzamelingen en equivalentierelaties. Deze binaire operator werkt ofwel in op een verzameling en een equivalentierelatie, ofwel op een binaire relatie en een equivalentierelatie. Als $A$ een verzameling is en $N$ een equivalentierelatie, dan is het resultaat van $A/N$ de verzameling met de equivalentieklassen uit $A$ volgens equivalentierelatie $N$. Voor een relatie $R$ over verzameling $A$ en met $N$ opnieuw een equivalentierelatie in $A$ is het resultaat van $R/N$ de relatie corresponderend met $R$ die geldt tussen de equivalentieklassen van verzameling $A$ i.p.v. tussen elementen van $A$ zelf. Voor een voorbeeld van het gebruik van de relatie $N$ en de operator /, zie figuur 3.4. Een formele definitie van operator / staat in appendix A.1.

De formalisatie van de eerder geformuleerde eigenschappen voor




**Figuur 3.4:** Voorbeeld van een geheugenordeningsrelatie $\xrightarrow{\text{mo}}$ (a) en de corresponderende gereduceerde relatie $\xrightarrow{\text{mo}}/N$ (b). In figuur (a) zijn de grensopdrachten $b_2$ en $b_5$ afzonderlijke opdrachten, terwijl in figuur (b) deze beschouwd worden als één geheel.

lock- en unlock-opdrachten is als volgt:

$$\begin{aligned}
&\forall\, l \in \mathit{Lock} :\ \forall\, u_1, u_2 \in \mathit{Unlock} :\ lN'u_1 \wedge lN'u_2 \Longrightarrow u_1 = u_2 \\
&\wedge \forall\, l_1 \in \mathit{Lock} : \\
&\qquad (\neg \exists\, u \in \mathit{Unlock} :\ l_1 N' u) \Longrightarrow \forall\, l_2 \in \mathit{Lock}_{mem(l_1)} :\ \forall\, p \in P :\ l_2 \xrightarrow{\text{mo}\,p} l_1 \\
&\wedge\ \forall\, u \in \mathit{Unlock} :\ \exists!\, l \in \mathit{Lock} :\ lN'u \\
&\wedge\ \forall\, l \in \mathit{Lock} :\ \forall\, u \in \mathit{Unlock} :\ lN'u \Longrightarrow mem(l) = mem(u) \\
&\wedge\ \forall\, l \in \mathit{Lock} :\ \forall\, u \in \mathit{Unlock} :\ lN'u \Longrightarrow l \xrightarrow{\text{po}} u \\
&\wedge\ \forall\, l \in \mathit{Lock} :\ \forall\, op \in \mathit{Op}_{mem(l)} :\ (l \xrightarrow{\text{po}} op \Longrightarrow \forall\, p \in P :\ l \xrightarrow{\text{mo}\,p} op) \\
&\wedge\ \forall\, u \in \mathit{Unlock} :\ \forall\, op \in \mathit{Op}_{mem(u)} :\ (op \xrightarrow{\text{po}} u \Longrightarrow \forall\, p \in P :\ op \xrightarrow{\text{mo}\,p} u)
\end{aligned} \tag{3.4}$$

Kritische secties kunnen in elkaar genest zijn of overlappen. We leggen op dat alle kritische secties die minstens één locatie gemeenschappelijk hebben onderling en t.o.v. grensopdrachten totaal geordend moeten zijn. Dit sluit meteen overlappende kritische secties voor dezelfde locatie uit. Deze eigenschap is het eenvoudigst te formuleren door de verschillende relaties $\xrightarrow{\text{mo}\,p}/N'$ te beschouwen in de verzameling $(\mathit{Lock}_m \cup \mathit{Unlock}_m)/N'$:

$$\begin{aligned}
&\forall\, m \in \mathit{Mem} :\ \xrightarrow{\text{mo}\,1}/N' \ldots \xrightarrow{\text{mo}\,n}/N' \\
&\qquad \text{zijn sequentieel consistent in } (\mathit{Lock}_m \cup \mathit{Unlock}_m)/N'
\end{aligned} \tag{3.5}$$

De eis dat relaties sequentieel consistent (s.c.) zijn over een bepaalde verzameling, is equivalent met de eis dat al deze relaties dezelfde totale ordening voorstellen in die verzameling. Voor een formele defi-

nitie van de eigenschap sequentieel consistente relaties, zie ook appendix A.1.

Verder is het toegelaten dat een grensopdracht voorkomt in een kritische sectie.

### 3.6.3 Grensopdrachten

Elke grensopdracht $b$ heeft een identificatienummer $j = id(b)$. De barrière met identificatie $j$ bestaat uit de verzameling grensopdrachten met hetzelfde identificatienummer. Van de grensopdrachten binnen een zelfde barrière wordt opgelegd dat het moet gaan over grensopdrachten met hetzelfde type (ll, ls, sl of ss) en ook dat een zelfde verzameling locaties $mem(b)$ met elk van de grensopdrachten is geassocieerd. Het type van een barrière bepaalt welke van de opdrachten vóór en na de barrière erdoor geordend worden. De geassocieerde verzameling locaties bepaalt voor welke opdrachten de volgorde wordt vastgelegd.

Onderstaande equivalentierelatie $N$ groepeert grensopdrachten tot barrières:

$$N \triangleq \{(op_1, op_2) \in Bar^2 | id(op_1) = id(op_2)\} \cup \{(op, op) | op \in Op\}$$

De formele gedaante voor bovenstaande eigenschappen is als volgt:

$$\begin{aligned}
&\forall\, b_1, b_2 \in Bar :\ b_1 N b_2 \implies type(b_1) = type(b_2) \wedge mem(b_1) = mem(b_2) \\
&\qquad \wedge ((\exists\, p \in P :\ b_1 \xrightarrow{\text{mo}\,p} b_2) \implies b_1 = b_2) \\
&\wedge\, \forall\, b_1, b_2 \in Bar_{l\cdot} :\ \forall\, p \in P :\ \forall\, op \in (Op_{mem(b_1)} \cap LF) \setminus \{b_1\} : \\
&\qquad b_1 N b_2 \implies (op \xrightarrow{\text{po}} b_1 \implies op \xrightarrow{\text{mo}\,p} b_2) \\
&\wedge\, \forall\, b_1, b_2 \in Bar_{s\cdot} :\ \forall\, p \in P :\ \forall\, op \in (Op_{mem(b_1)} \cap SF) \setminus \{b_1\} : \\
&\qquad b_1 N b_2 \implies (op \xrightarrow{\text{po}} b_1 \implies op \xrightarrow{\text{mo}\,p} b_2) \\
&\wedge\, \forall\, b_1, b_2 \in Bar_{\cdot l} :\ \forall\, p \in P :\ \forall\, op \in (Op_{mem(b_1)} \cap LF) \setminus \{b_1\} : \\
&\qquad b_1 N b_2 \implies (b_1 \xrightarrow{\text{po}} op \implies b_2 \xrightarrow{\text{mo}\,p} op) \\
&\wedge\, \forall\, b_1, b_2 \in Bar_{\cdot s} :\ \forall\, p \in P :\ \forall\, op \in (Op_{mem(b_1)} \cap SF) \setminus \{b_1\} : \\
&\qquad b_1 N b_2 \implies (b_1 \xrightarrow{\text{po}} op \implies b_2 \xrightarrow{\text{mo}\,p} op)
\end{aligned} \tag{3.6}$$

Voor barrières waarmee minstens één gemeenschappelijke locatie geassocieerd is leggen we op dat deze barrières onderling en t.o.v. lock- en unlock-opdrachten geordend zijn:

$$\forall\, m \in Mem :\ \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N \text{ zijn sequentieel consistent in } Sync_m/N \tag{3.7}$$

Grensopdrachten worden hoofdzakelijk op twee wijzen toegepast. De eerste wijze is die waarbij elke barrière slechts één grensopdracht bevat. Hiervoor wordt ook wel de benaming *fence* gebruikt. De STBAR-opdracht (*store barrier*) in de SPARC-architectuur is hiervan een voorbeeld [SPA92]. Deze opdracht garandeert dat de volgorde van schrijfopdrachten die voor of na de STBAR-opdracht werden uitgevoerd behouden blijft. Deze opdracht stemt dus overeen met $(bar_{ss}, i, Mem, j, p)$, met $j$ een uniek geheel getal per STBAR-opdracht.

Een tweede wijze waarop grensopdrachten vaak toegepast worden, is als onderdeel van een grensopdracht die alle processors omvat. Het DSM-systeem TreadMarks bijvoorbeeld voorziet de functie Tmk_barrier(). Tijdens de uitvoering van de Tmk_barrier()-functie wordt de programma-uitvoering op elke processor uitgesteld tot wanneer alle andere processors een aanroep van Tmk_barrier() hebben bereikt. Zowel de uitvoering van lees- als schrijfopdrachten wordt dus op deze manier geordend. Een aanroep van deze functie correspondeert dus met de uitvoering op elke processor $p$ van de opdrachten $(bar_{ss}, i_p, p, Mem, j)$ gevolgd door $(bar_{ll}, i_p + 1, p, Mem, j + 1)$, ofwel de opdrachten $(bar_{sl}, i_p, p, Mem, j)$ gevolgd door $(bar_{ls}, i_p{+}1, p, Mem, j{+}1)$. Beide sequenties hebben dezelfde semantiek. Hierbij legt het getal $i_p$ vast waar de grensopdrachten in de programma-uitvoering voorkomen, en is $j$ een uniek even getal dat de barrière identificeert over de verschillende processors heen.

De eigenschappen (3.1) t.e.m. (3.7) zullen verder aangeduid worden met de benaming **basiseigenschappen van een geheugenmodel**.

## 3.7 Andere eigenschappen van geheugenmodellen

De eigenschappen in deze paragraaf gelden voor de meeste geheugenmodellen, maar niet voor allemaal. Er worden twee eigenschappen behandeld, nl. connectiviteit en consistentie.

Als in een programma een lus voorkomt die pas eindigt als een andere processor een locatie in het geheugen verandert, dan hangt de werking van het programma af van het gegeven of waarden geschreven door één processor worden doorgegeven naar het lokale geheugen

van een andere processor. Een voorbeeld van een dergelijk programma staat in figuur 3.3. De eigenschap dat elke waarde die naar het geheugen geschreven wordt zonder daarna overschreven te worden, ooit door alle andere processors zal worden waargenomen, heet **connectiviteit** (ook *liveness*). Indien we veronderstellen dat elke afzonderlijke opdracht in een eindige tijd wordt uitgevoerd, dan is een mogelijke formele vertaling van deze eigenschap als volgt:

$$\forall\, p, q \in P :\ \forall\, s \in SF_q :\ \{op | op \xrightarrow{\text{mo}\,p} s\} \text{ is een eindige verzameling.} \quad (3.8)$$

Als een geheugenmodel niet voldoet aan deze eigenschap, en de beschouwde programma's uitsluitend uit lees- en schrijfopdrachten bestaan, dan is een verzameling niet verbonden uniprocessors een volwaardige implementatie van dat geheugenmodel.

De volgorde waarin opdrachten met betrekking tot verschillende locaties worden uitgevoerd kan verschillen tussen de geheugenordeningen van de verschillende processors. Als het geheugenmodel voldoet aan de eigenschap **coherentie**, dan heeft elke processor hetzelfde beeld van de opdrachten uitgevoerd op een locatie:

$$\forall\, m \in Mem :\ \xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n} \text{ zijn consistent over } LSF_m \quad (3.9)$$

Relaties zijn consistent over een gegeven verzameling $A$ als en slechts als de restricties van al de beschouwde relaties tot de verzameling $A$ gelijk zijn. Voor een formele definitie, zie appendix A.1.

## 3.8 Geheugenmodel en voorbeelden

Een mogelijke uitvoering $E$ van een programma op een multiprocessorsysteem bestaat uit de uitgevoerde opdrachten *Op* en hun programmaordening, en wordt genoteerd als $E = (Op, \xrightarrow{\text{po}})$. Het **geheugenmodel** *Mod* van een multiprocessorsysteem legt vast welke uitvoeringen mogelijk zijn en welke niet op het gegeven systeem. Met elk geheugenmodel *Mod* is dus een verzameling van toegelaten uitvoeringen geassocieerd. Een geheugenmodel wordt meestal gedefinieerd in termen van uitgevoerde opdrachten *Op*, de programmaordening $\xrightarrow{\text{po}}$ en ook de geheugenordeningen $\xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n}$. Hierbij zijn *Op* en $\xrightarrow{\text{po}}$ gegeven, en zijn de $\xrightarrow{\text{mo}}$-relaties te bepalen onbekenden. Om aan te tonen dat een gegeven uitvoering $E$ mogelijk is in het geheugenmodel *Mod* dient er dus aangetoond te worden dat er $\xrightarrow{\text{mo}}$-relaties bestaan waarbij $E$ aan de

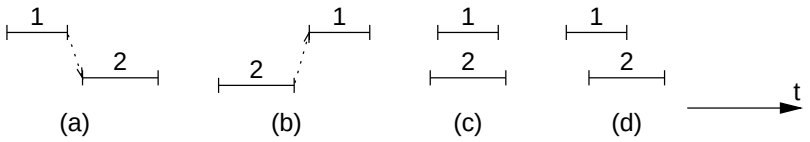
definitie van *Mod* voldoet. Aantonen dat een uitvoering $E$ niet aan een geheugenmodel *Mod* voldoet dient dus te gebeuren door te bewijzen dat er geen geheugenordeningsrelaties $\xrightarrow{\text{mo}}$ bestaan waarvoor $E$ aan de definitie van *Mod* voldoet. In wat volgt staan verscheidene voorbeelden van uitvoeringen, samen met de vermelding van een geheugenmodel waaraan ze voldoen en eventueel ook de vermelding van een geheugenmodel waaraan het voorbeeld niet voldoet. In de voorbeelden zal ofwel de reden waarom het voorbeeld niet aan het vermelde geheugenmodel voldoet voldoende duidelijk zijn, ofwel wordt kort gemotiveerd waarom het voorbeeld niet voldoet aan een geheugenmodel.

Deze paragraaf bevat de definitie van meerdere bestaande geheugenmodellen. Elke definitie bestaat uit twee delen: eerst de definitie zoals ze oorspronkelijk werd geformuleerd in de literatuur, en daarna een definitie die het oorspronkelijke geheugenmodel uitbreidt tot een model dat alle opdrachten bevat die in dit hoofdstuk reeds gedefinieerd werden. Er zal blijken dat de uitgebreide geheugenmodeldefinities het vergelijken van geheugenmodellen aanzienlijk vereenvoudigen.

### 3.8.1 Atomaire consistentie

Het begrip *atomaire consistentie* of *atomische consistentie* wordt door meerdere auteurs gedefinieerd, waarbij de definities onderling lichtjes van elkaar afwijken. Vandaar dat voor dit geheugenmodel geen definitie uit de literatuur werd overgenomen, maar op basis van de literatuur een definitie werd samengesteld. Onderstaande definitie is gebaseerd op de publicaties [Mis86, HA90, Mos93, AW94, ANK+95].

Om de definitie van atomaire consistentie te kunnen formuleren associëren we met elke opdracht $op$ een uitvoeringsinterval $[t_1, t_2]$. De uitvoering van opdracht $op$ vangt aan op of na het tijdstip $t_1$, en na het tijdstip $t_2$ is er niets dat nog het resultaat van de opdracht kan beïnvloeden. Dit onderstelt het bestaan van een totale ordening voor alle opdrachten. Op basis van de uitvoeringsintervallen wordt een opdracht $op_1$ dus ofwel voor, tezelfdertijd of na een opdracht $op_2$ uitgevoerd, naargelang het eerste tijdsinterval voor, overlappend, of na het tweede tijdsinterval komt – zie ook figuur 3.5. De partiële ordening die ontstaat op basis van de ordening van de tijdsintervallen geassocieerd met opdrachten krijgt de naam $<_t$. Met deze notatie is de definitie van **atomaire consistentie** of AC als volgt: een systeem voldoet aan AC als en slechts als alle opdrachten lees- en schrijfopdrachten zijn en in pro-




**Figuur 3.5:** Ordening gebeurtenissen volgens begin- en eindtijdstip: voor (a), na (b) of gelijktijdig (c en d).

grammaordening worden uitgevoerd. Bovendien moet deze volgorde overeenstemmen met de volgorde vastgelegd door $<_t$, en bovendien moet $<_t$ een totale ordening zijn. Of, equivalent hiermee:

$$
\begin{aligned}
& Op = L \cup F \cup S \\
& \wedge \xrightarrow{\text{mo}\,1} = \ldots = \xrightarrow{\text{mo}\,n} \\
& \wedge \xrightarrow{\text{mo}\,1} = <_t \\
& \wedge \xrightarrow{\text{mo}\,1} \text{ is een totale ordening in } Op \\
& \wedge \forall\, p \in P : \xrightarrow{\text{po}\,p} \subset \xrightarrow{\text{mo}\,p}
\end{aligned}
$$

Breiden we de definitie van AC uit met synchronisatieopdrachten, dan bekomen we uitgebreide atomaire consistentie of AC$^\star$:

$$
\begin{aligned}
& \xrightarrow{\text{mo}\,1} = \ldots = \xrightarrow{\text{mo}\,n} \\
& \wedge \xrightarrow{\text{mo}\,1} = <_t \\
& \wedge \xrightarrow{\text{mo}\,1} /N \text{ is een totale ordening in } Op/N \\
& \wedge \forall\, p \in P : \xrightarrow{\text{po}\,p} \subset \xrightarrow{\text{mo}\,p}
\end{aligned}
$$

### 3.8.2 Sequentiële consistentie of SC

Het geheugenmodel sequentiële consistentie werd door Lamport gedefinieerd in 1979. Het was de eerste nauwkeurige beschrijving van hoe een multiprocessor met gemeenschappelijk geheugen zich moest gedragen t.o.v. lees- en schrijfopdrachten. Lamport definieerde sequentiële consistentie als volgt:

> *...het resultaat van een uitvoering is hetzelfde als wanneer de opdrachten van alle processors zouden uitgevoerd worden in een sequentie, en de opdrachten van elke individuele processor daarbij in deze sequentie voorkomen in de volgorde vastgelegd door het programma uitgevoerd door die processor [Lam79].*

| |
|---|
| $Op = L_{ord} \cup S_{ord}$ |
| Enkel gewone lees- en schrijfopdrachten worden beschouwd. |
| $\forall\, p \in P :\ \xrightarrow{\text{po}\,p} \subset \xrightarrow{\text{mo}\,p}$ |
| Een processor voert zijn eigen opdrachten in volgorde uit. |
| $\xrightarrow{\text{mo}\,1} = \ldots = \xrightarrow{\text{mo}\,n}$ |
| Alle geheugenordeningen zijn identiek, |
| $\xrightarrow{\text{mo}\,1}$ is een totale ordening in $Op$ |
| en bovendien ook totaal geordend. |
| Eig. (3.1) |
| Uniprocessorcorrectheid is van toepassing. |
| Eig. (3.3), (3.8), (3.9) |
| Overige eigenschappen (impliciet in tekstuele definitie). |
| Eig. (3.2), (3.5), (3.7), (3.4), (3.6) |
| Geldig wegens zonder onderwerp. |

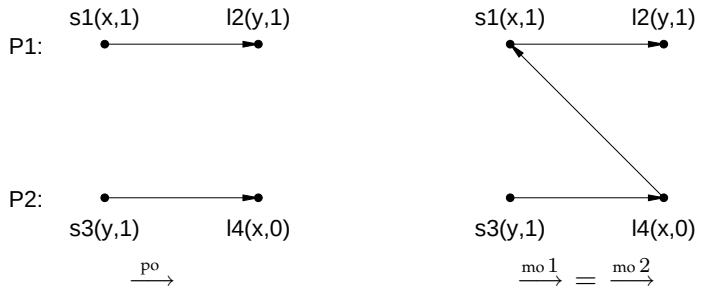
**Tabel 3.4:** Vertaling van Lamports definitie van sequentiële consistentie.

De formele vertaling van dit model staat in tabel 3.4.

Lamports definitie kan uitgebreid worden met andere types opdrachten zoals lees/wijzig/schrijf-opdrachten, bijzondere opdrachten (etiket $\neq$ $ord_L$), en met synchronisatieopdrachten. Bij het toevoegen van grensopdrachten is het noodzakelijk in ordeningsrelaties grensopdrachten tot barrières te groeperen m.b.v. de bewerking $/N$, die werd ingevoerd in paragraaf 3.6.1. In plaats van de relaties $\xrightarrow{\text{mo}\,p}$ en de verzameling $Op$ beschouwen we verder de relaties $\xrightarrow{\text{mo}\,p}/N$ en de verzameling $Op/N$. Het geheugenmodel uitgebreide sequentiële consistentie, of $\mathbf{SC}^\star$ , is het geheugenmodel waarbij de basiseigenschappen gelden, (3.8) en (3.9) gelden, en waarbij bovendien ook geldt dat

$$\begin{aligned}&\forall\, p \in P :\ \xrightarrow{\text{po}\,p} \subset \xrightarrow{\text{mo}\,p}\\ &\wedge\ \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N \text{ zijn sequentieel consistent in } Op/N\end{aligned}$$

Een voorbeeld van een sequentieel consistente uitvoering staat in figuur 3.6. Naargelang de volgorde waarin de opdrachten $s_1, l_2, s_3$ en $l_4$ worden uitgevoerd voldoet dit voorbeeld al of niet aan het model atomische consistentie (AC). Met de volgorde $s_3 <_t l_4 <_t s_1 <_t l_2$ bijvoorbeeld voldoet het voorbeeld aan AC, en met de volgorde $s_1 <_t s_3 <_t l_2 <_t l_4$ voldoet het voorbeeld niet aan atomische consistentie (AC).




**Figuur 3.6:** Voorbeeld van een sequentieel consistente (SC) uitvoering: programmaordening en geheugenordening.

---

$Op = L_{ord} \cup S_{ord}$
: Enkel gewone lees- en schrijfopdrachten worden beschouwd (impliciet).

$\forall\, p \in P :\ \left( \xrightarrow{\text{po}} \cup \bigcup_{q \in P} \bigcup_{m \in Mem} \left( \xrightarrow{\text{mo}\,q} \cap (SF_m \times LF_{m,q}) \right)^{-} \right)^{\star} \subset \xrightarrow{\text{mo}\,p}$
: Alle processors nemen causaal verwante gebeurtenissen in dezelfde volgorde waar.

$\forall\, p \in P :\ \xrightarrow{\text{mo}\,p}$ is een totale ordening in $Op$
: Elke geheugenordening is totaal geordend (impliciet).

Eig. (3.1), (3.3), (3.8)
: Geldige eigenschappen (impliciet in tekstuele definitie).

Eig. (3.2), (3.5), (3.7), (3.4), (3.6)
: Geldig wegens zonder onderwerp.

---

**Tabel 3.5:** Vertaling van Mosbergers definitie van causale consistentie.

### 3.8.3 Causale consistentie of CC

Mosberger definieert causale consistentie als volgt [Mos93]:

> Een geheugensysteem is causaal consistent als alle processors een overeenstemmend idee hebben van causaal verwante gebeurtenissen. Causaal niet verwante gebeurtenissen kunnen door verschillende processors in een andere volgorde worden waargenomen.

In deze definitie verwijst *causaal verwante gebeurtenissen* naar de programmaordening. De term *waarnemen* verwijst naar de geheugenordening. De formele vertaling van dit model staat in tabel 3.5.

Het geheugenmodel causale consistentie heeft als belangrijkste eigenschap dat het een implementatie toelaat waarbij voor leesopdrachten niet moet worden gewacht op andere geheugens dan het eigen lokale geheugen. Dit gaat echter ten koste van de eigenschap coherentie (3.9).

Breiden we Mosbergers definitie van causale consistentie uit met andere soorten opdrachten en etiketten, dan bekomen we het geheugenmodel CC$^\star$: CC$^\star$ is het geheugenmodel dat voldoet aan de basiseigenschappen (3.1) t.e.m. (3.7) en ook aan de connectiviteitseigenschap (3.8), en waarbij verder onderstaande uitdrukking geldt:

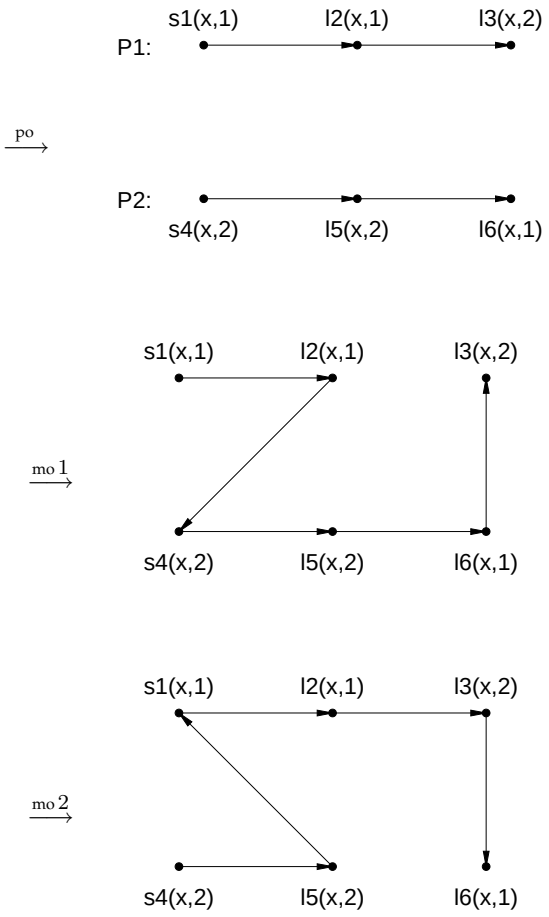
$$\forall\, p \in P :\ \xrightarrow{\text{mo}\,p} \text{ is een totale ordening in } Op$$
$$\wedge\, \forall\, p \in P :\ \left( \xrightarrow{\text{po}} \cup \bigcup_{q \in P} \bigcup_{m \in Mem} \left( \xrightarrow{\text{mo}\,q} \cap\, (SF_m \times LF_{m,q}) \right)^{-} \right)^{\star} \subset \xrightarrow{\text{mo}\,p}$$

Een voorbeeld van een causaal consistente uitvoering staat in figuur 3.7. Deze uitvoering voldoet niet aan de eigenschap coherentie (3.9) omdat de schrijfopdrachten $s_1$ en $s_4$ een waarde naar dezelfde locatie schrijven en de veranderingen van de waarde op die locatie in twee verschillende volgordes worden waargenomen.

### 3.8.4 Causale consistentie volgens Ahamad of CCA

Causale consistentie werd door Ahamad als volgt gedefinieerd [ANK+95]:

> Een lokale geschiedenis van een proces $p_i$, notatie $L_i$, is een sequentie van lees- en schrijfopdrachten. Als een opdracht $o_1$ voorafgaat aan opdracht $o_2$ in $L_i$, dan wordt dat genoteerd als $\emptyset_1 \rightarrow_i o_2$ en zeggen we dat $o_1$ aan $o_2$ voorafgaat in programmaordening.
>
> Een geschiedenis $H =< L_1 \ldots L_n >$ is een verzameling van lokale geschiedenissen, één per proces.
>
> Een serialisatie S respecteert een ordening $\rightarrow$ als voor elk paar opdrachten $o_1$ en $o_2$ in $S$, $o_1 \rightarrow o_2$ impliceert dat $o_1$ aan $o_2$ voorafgaat in $S$.
>
> Als $A$ een verzameling opdrachten is, dan is $S$ een serialisatie van $A$ als $S$ een lineaire sequentie opdrachten is die exact de opdrachten uit $A$ bevat en bovendien elke leesopdracht in die sequentie de waarde retourneert die werd geschreven door de meest recent voorafgaande schrijfopdracht.




**Figuur 3.7:** Voorbeeld van een causaal consistente (CC) uitvoering.

Initieel hebben alle locaties in het geheugen de waarde $\perp$. $r(x)v$ is een leesopdracht die van locatie $x$ waarde $v$ leest, en $w(x)v$ is een schrijfopdracht die waarde $v$ naar locatie $x$ schrijft. $L_i$ stelt de verzameling opdrachten uitgevoerd door processor $p_i$ voor. $H$ stelt de geschiedenis van een uitvoering voor, en bestaat uit alle uitgevoerde opdrachten. $A^H_{i+w}$ is daarbij de geschiedenis $H$ beperkt tot de unie van $L_i$ en alle schrijfopdrachten uit $H$. De programmaordening voor opdrachten $o_1$ en $o_2$ wordt genoteerd als $o_1 \rightarrow o_2$ en $o_1 \rightarrow_i o_2$ voor de programmaordening van de opdrachten van processor $i$.

Een schrijft-in volgorde $\mapsto$ op $H$ is een relatie die voldoet aan :

- als $o_1 \mapsto o_2$, dan bestaan er $x$ en $v$ zodanig dat $o_1 = w(x)v$ en $o_2 = r(x)v$;
- voor elke opdracht $o_2$ bestaat er hoogstens een $o_1$ zodanig dat $o_1 \mapsto o_2$;
- als $o_2 = r(x)v$ voor een $x$ en er bestaat geen $o_1$ zodanig dat $o_1 \mapsto o_2$, dan is $v = \perp$; m.a.w. een leesopdracht zonder schrijfopdracht moet de initiële waarde lezen.

Er kunnen meerdere van deze $\mapsto$ relaties bestaan.

Een causaliteitsordening $\rightsquigarrow$ geïnduceerd door $\mapsto$ voor $H$ is een partiële ordening die de transitieve sluiting is van de unie van de programmaordening van de geschiedenis $H$ en de ordening $\mapsto$.

Een geschiedenis $H$ is causaal als er voor elk proces $p_i$ een serialisatie $S_i$ bestaat van $A^H_{i+w}$ die $\rightsquigarrow$ respecteert. Bovendien moet de waarde van elke leesopdracht in deze serialisatie de waarde zijn die geschreven werd door de meest recente voorafgaande schrijfopdracht.

De relatie $\mapsto$ is een relatie tussen schrijf- en leesopdrachten, en wel zodanig dat $o_1 \mapsto o_2$ als opdracht $o_1$ de waarde schrijft die door opdracht $o_2$ gelezen werd. De daaropvolgende stap definieert in woorden de relatie $\rightsquigarrow$ als volgt: $\rightsquigarrow = (\xrightarrow{\text{po}} \cup \mapsto)^\star$. De relatie $\rightsquigarrow$ is nodig om in de volgende stap te kunnen eisen dat de geheugenordening $S_i$ voor proces $i$ elke programmaordening $\xrightarrow{\text{po}\,i}$ respecteert en bovendien voldoet aan de geheugenwerkingseigenschap.

| |
|---|
| $Op = L_{ord} \cup S_{ord}$<br>Enkel gewone lees- en schrijfopdrachten worden beschouwd. |
| $\forall\, p \in P :\ \xrightarrow{\text{mo}\,p}$ is een totale ordening in $Op$<br>Elke geheugenordening $\xrightarrow{\text{mo}\,p}$ is totaal geordend. |
| $\forall\, p \in P :\ \left(\xrightarrow{\text{po}} \cup \bigcup_{q \in P} \bigcup_{m \in Mem} (\xrightarrow{\text{mo}\,q} \cap (SF_m \times LF_{m,q}))^{-}\right)^{\star} \subset \xrightarrow{\text{mo}\,p}$<br>Causaliteitseigenschap, nl. dat relatie $S_p$ relatie $\rightsquigarrow$ respecteert. |
| Eig. (3.3)<br>De eigenschap geheugenwerking geldt (expliciet). |
| Eig. (3.1), (3.3), (3.8)<br>Geldige eigenschappen (impliciet in tekstuele definitie). |
| Eig. (3.2), (3.5), (3.7), (3.4), (3.6)<br>Geldig wegens zonder onderwerp. |

**Tabel 3.6:** Vertaling van Vertaald model gebaseerd op Ahamads definitie van causale consistentie.

Uit bovenstaande definitie van *respecteert* volgt dat de eigenschap dat serialisatie S ordening $\rightarrow$ respecteert equivalent is met de eigenschap $\rightarrow \cap dom(S)^2 \subset S$.

In wat volgt zal steeds ondersteld worden dat waar naar de schrijftin relatie $\mapsto$ verwezen wordt, dat dan de relatie $\rightsquigarrow = (\mapsto \cup \xrightarrow{\text{po}})^{\star}$ een partiële ordening is. Bovendien wordt ook ondersteld dat als $\mapsto$ vermeld wordt, dat dan een serialisatie $S_i$ bestaat, met $(\rightsquigarrow \cap (A^H_{i+w})^2) \subset S_i$, die voldoet aan eigenschap 3.3. Zonder deze voorwaarden heeft de relatie $\mapsto$ geen zinvolle interpretatie. Waar Ahamad stelt dat leesopdrachten zonder voorafgaande schrijfopdracht de initiële waarde $\perp$ opleveren, is in eigenschap 3.3 hun waarde niet gedefinieerd. Daar de waarde toegekend aan een leesopdracht zonder voorafgaande schrijfopdracht geen wezenlijke invloed heeft op het gedefinieerde geheugenmodel, wordt er abstractie gemaakt van de initiële waarde van geheugenlocaties.

Bovenstaande onderstellingen impliceren dat de relatie $\mapsto$ enkel een zinvolle interpretatie heeft in het geheugenmodel CCA en in geheugenmodellen die sterkere beperkingen opleggen aan uitvoeringen.

De formele vertaling van het geheugenmodel causale consistentie gebaseerd op Ahamads definitie staat in tabel 3.6. In die vertaling zijn de relaties $S_p$ terug te vinden als $S_p = \xrightarrow{\text{mo}\,p} \cap (Op_p \cup SF)^2$, en werd relatie $\mapsto$ vervangen door $\bigcup_{q \in P} \bigcup_{m \in Mem} (\xrightarrow{\text{mo}\,q} \cap (SF_m \times LF_{m,q}))^{-}$.

Breiden we deze definitie uit met andere soorten opdrachten en etiketten, dan bekomen we het geheugenmodel CCA$^\star$: CCA$^\star$ is het geheugenmodel dat voldoet aan de basiseigenschappen (3.1) t.e.m. (3.7), aan eigenschap 3.8 en waarbij bovendien geldt dat:

$$\begin{array}{l}\forall\, p \in P :\ \xrightarrow{\text{mo}\,p} \text{ is een totale ordening in } Op \\ \wedge\, \forall\, p \in P :\ \left( \xrightarrow{\text{po}} \cup \bigcup_{q \in P} \bigcup_{m \in Mem} \left( \xrightarrow{\text{mo}\,q} \cap\, (SF_m \times LF_{m,q}) \right)^{-} \right)^{\star} \subset \xrightarrow{\text{mo}\,p}\end{array}$$

De geheugenmodellen CC$^\star$ en CCA$^\star$ zijn equivalent omdat beide modellen dezelfde definitie hebben. Het bewijs van de equivalentie van Ahamads definitie en de definitie van CCA$^\star$ staat in lemma B.4.2.

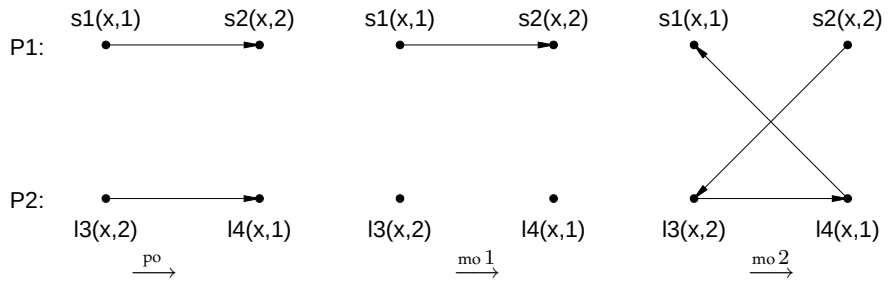
### 3.8.5 PRAM

De afkorting PRAM staat voor *Pipelined Random Access Memory*. Deze naam verwijst naar de eigenschap van het geheugenmodel PRAM dat alle schrijfopdrachten van een processor kunnen gepijplijnd worden, zodat een processor niet hoeft te wachten op hun voltooiing. Er wordt hierbij ondersteld dat bij leesopdrachten eerst de pijplijn gecontroleerd wordt op schrijfopdrachten naar hetzelfde adres. De afkorting PRAM wordt ook gebruikt als afkorting van het begrip *Parallel Random Access Machine*, dat echter niet verwant is aan het PRAM-geheugenmodel. We baseren ons op de definitie voor het PRAM-geheugenmodel volgens [ANK$^+$95], met dezelfde notatie als bij de definitie van causale consistentie in paragraaf 3.8.4:

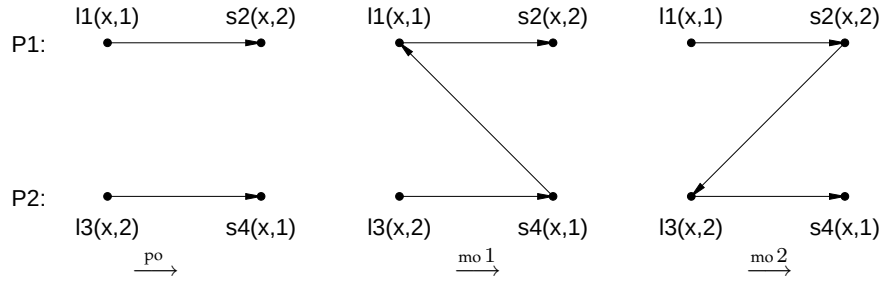
> Een geschiedenis is PRAM als er voor elk proces $p_i$ een serialisatie $S_i$ van $A^H_{i+w}$ bestaat die alle programmaordeningen $\rightarrow_j$ respecteert. Een geheugen is PRAM als het enkel PRAM-geschiedenissen toelaat.

Breiden we deze definitie uit met lees/wijzig/schrijf-opdrachten, etiketten en synchronisatieopdrachten, dan bekomen we de definitie van PRAM$^\star$. In het geheugenmodel **PRAM**$^\star$ gelden de basiseigenschappen (3.1) t.e.m. (3.7) en ook de connectiviteitseigenschap 3.8. Verder geldt dat opdrachten in programmaordening worden uitgevoerd en dat elke geheugenordening een totale ordening is:

$$\begin{array}{l}\forall\, p \in P :\ \xrightarrow{\text{po}\,p} \subset \xrightarrow{\text{mo}\,p} \\ \wedge\, \forall\, p \in P :\ \xrightarrow{\text{mo}\,p}/N \text{ is een totale ordening in } Op/N.\end{array}$$




**Figuur 3.8:** Eerste voorbeeld van een PRAM-uitvoering.




**Figuur 3.9:** Tweede voorbeeld van een PRAM-uitvoering.

Voorbeelden van PRAM-uitvoeringen staan in figuren 3.8 en 3.9. Het voorbeeld uit figuur 3.8 voldoet niet aan causale consistentie (CC), omdat uit de CC-eigenschappen 3.3 en $\forall\, p \in P :\ \xrightarrow{\text{po}} \subset \xrightarrow{\text{mo}\,p}$ volgt dat zou moeten gelden dat $s_1 \xrightarrow{\text{mo}\,2} s_2 \xrightarrow{\text{mo}\,2} l_3 \xrightarrow{\text{mo}\,2} l_4 \xrightarrow{\text{mo}\,2} s_1$, wat niet kan. Dit voorbeeld voldoet ook aan het geheugenmodel IA-64, zoals geïllustreerd in figuur 3.17.

Het tweede voorbeeld, nl. het voorbeeld uit figuur 3.9, voldoet niet alleen aan het geheugenmodel PRAM maar ook aan de geheugenmodellen WO, RC en LRC zoals verder wordt aangetoond. Het voorbeeld voldoet echter niet aan het geheugenmodel IA-64. Uit de definitie van IA-64 volgt dat de eigenschappen (3.1), (3.3) en (3.9) samen moeten voldaan zijn. Dit heeft als gevolg dat er zou moeten gelden dat $l_1 \xrightarrow{\text{mo}\,1} s_2 \xrightarrow{\text{mo}\,1} l_3 \xrightarrow{\text{mo}\,1} s_4 \xrightarrow{\text{mo}\,1} l_1$, wat niet kan.

### 3.8.6 Processorconsistentie volgens Ahamad of PC

Er bestaan minstens twee verschillende geheugenmodellen die beide met de naam processorconsistentie worden aangeduid – zie [ABJN93]

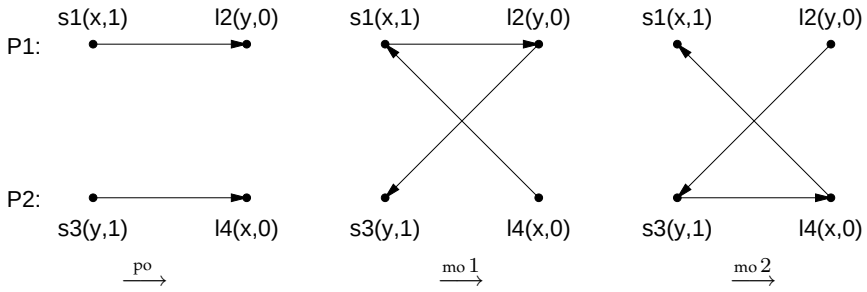
| | |
|---|---|
| $Op = L_{ord} \cup S_{ord}$ | Enkel gewone lees- en schrijfopdrachten worden beschouwd (impliciet). |
| $\forall\, p \in P :\ \xrightarrow{\text{po}\,p} \subset \xrightarrow{\text{mo}\,p}$ | Een processor voert zijn eigen opdrachten in volgorde uit. |
| $\forall\, p \in P :\ \left(\xrightarrow{\text{po}} \cap SF^2\right) \subset \xrightarrow{\text{mo}\,p}$ | Schrijfopdrachten worden in programmaordening uitgevoerd. |
| $\xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n}$ sequentieel consistent in $SF_m$ | Schrijfopdrachten per locatie zijn totaal geordend (impliciet). |
| Eig. (3.1) | Uniprocessorcorrectheid is van toepassing. |
| Eig. (3.3), (3.8) (3.9) | Overige eigenschappen gelden ook (impliciet in tekstuele definitie). |
| Eig. (3.2), (3.5), (3.7), (3.4), (3.6) | Geldig wegens zonder onderwerp. |

**Tabel 3.7:** Vertaling van processorconsistentie.

en [GLL+90]. In deze paragraaf wordt Ahamads definitie behandeld. Ahamad definieert **processorconsistentie** (*processor consistency*) of **PC** als volgt, gebaseerd op de eerdere definitie van Goodman:

> *Een multiprocessor wordt processorconsistent genaamd als het resultaat van om het even welke uitvoering hetzelfde is als wanneer de opdrachten van elke individuele processor in de sequentiële volgorde voorkomen zoals vastgelegd door het programma uitgevoerd door die processor. Dus de volgorde waarin schrijfopdrachten van twee processors voorkomen, zoals waargenomen door henzelf of door een derde processor, moet niet identiek zijn, maar schrijfopdrachten gelanceerd door om het even welke processor mogen niet in een andere volgorde worden waargenomen dan deze waarin ze werden gelanceerd [Goo89]. Ook moet er een unieke volgorde bestaan voor de schrijfopdrachten uitgevoerd op een gegeven locatie, en de schrijfopdrachten van om het even welke processor moeten waargenomen worden in de volgorde waarin die processor hen uitvoert. Bovendien moeten deze voorwaarden wederzijds consistent zijn [ABJN93].*

De formele vertaling van dit model staat in tabel 3.7.




**Figuur 3.10:** Voorbeeld van een processorconsistente (PC) uitvoering.

Ook het model processorconsistentie kunnen we uitbreiden met bijzondere lees- en schrijfopdrachten en met synchronisatieopdrachten. Uitgebreide processorconsistentie of **PC**$^\star$ is het geheugenmodel waarbij de basiseigenschappen gelden, (3.8) en (3.9) gelden, en waarbij bovendien ook geldt dat

$$\begin{aligned}
&\forall\, p \in P :\ \xrightarrow{\text{po}\,p} \subset \xrightarrow{\text{mo}\,p} \\
&\wedge\, \forall\, m \in \mathit{Mem} :\ \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N \text{ zijn s.c. in } SF_m/N \\
&\wedge\, \forall\, p \in P :\ \left(\xrightarrow{\text{po}} \cap\, SF^2\right) \subset \xrightarrow{\text{mo}\,p}.
\end{aligned}$$

De tweede eigenschap drukt uit dat alle processors hetzelfde beeld hebben van de volgorde van de schrijfopdrachten en lees/wijzig/schrijfopdrachten, en de derde eigenschap drukt uit dat schrijfopdrachten door alle processors in dezelfde volgorde worden waargenomen.

Een voorbeeld van een processorconsistente uitvoering staat in figuur 3.10. Dit voorbeeld voldoet ook aan het TSO-geheugenmodel. Uit de eigenschappen van het geheugenmodel sequentiële consistentie (SC), nl. (3.3), $\xrightarrow{\text{mo}\,1} = \xrightarrow{\text{mo}\,2}$ en $\forall\, p \in P :\ \xrightarrow{\text{po}\,p} \subset \xrightarrow{\text{mo}\,p}$, volgt dat er voor SC zou moeten gelden dat $s_1 \xrightarrow{\text{mo}\,1} l_2 \xrightarrow{\text{mo}\,1} s_3 \xrightarrow{\text{mo}\,1} l_4 \xrightarrow{\text{mo}\,1} s_1$. Dit is strijdig met de voorwaarde dat $\xrightarrow{\text{mo}\,1}$ een partiële ordening moet zijn, dus voldoet de uitvoering uit figuur 3.10 niet aan sequentiële consistentie (SC).

### 3.8.7 Totale ordening schrijfopdrachten of TSO

Het TSO-geheugenmodel is een geheugenmodel gedefinieerd door de organisatie SPARC International Inc. en dat het gedrag beschrijft van lees-, schrijf- en lees/wijzig/schrijf-opdrachten op hun processors. In tegenstelling tot vele andere producenten van processors voor multi-

| opdrachttype | SPARC-notatie | eigen notatie |
|---|---|---|
| leesopdracht | $L_a^p$ met $Val[L_a^p] = v_l$ | $(l, ord_L, i_p, p, \{a\}, v_l)$ |
| schrijfopdracht | $S_a^p \# v_s$ | $(s, ord_L, i_p, p, \{a\}, v_s)$ |
| atomische lees/wijzig/schrijf-opdracht | $[L_a^p; S_a^p \# v_s]$ met $Val[L_a^p] = v_l$ | $(f, ord_L, ord_L, i_p, p, \{a\}, v_l, v_s)$ |
| schrijf-schrijf-grensopdracht | $\mathcal{S}^p$ | $(bar_{ss}, i_p, p, Mem, j)$ |

**Tabel 3.8:** Verband tussen de notatie in de oorspronkelijke definitie van de TSO- en PSO-geheugenmodellen, en de eigen notatie. De SPARC-notatie veronderstelt het etiket $ord_L$. Grensopdrachten gelden voor schrijfopdrachten en elke barrière bestaat uit juist één grensopdracht. Elke barrière-identificatie $j$ is uniek per uitvoering. De instructienummers $i_p$ kunnen ingevuld worden op basis van de programmaordening aangeduid met het symbool `;`.

processorsystemen levert SPARC zowel een informele als een formele beschrijving voor de geheugenmodellen die ondersteund worden door hun processors. In deze paragraaf beschouwen we het geheugenmodel TSO, *total store order*, genaamd naar de eigenschap van het TSO-geheugenmodel dat de schrijfopdrachten in programmaordening worden uitgevoerd. Alhoewel door het SPARC-consortium het TSO-geheugenmodel gedefinieerd werd voor zichzelf wijzigende programma's, zullen we ons hier beperken tot zichzelf niet wijzigende programma's – zie tabel 3.9. Het verband tussen de SPARC-notatie en onze notatie staat in tabel 3.8.

Op basis van het geheugenmodel TSO definiëren we het geheugenmodel TSO$^\star$, namelijk de uitbreiding van TSO met bijzondere opdrachten, lock/unlock-synchronisatie en grensopdrachten over meerdere processors. **TSO$^\star$** is dus het geheugenmodel waarbij de basiseigenschappen (3.1) t.e.m. (3.7) gelden, en ook

$$\xrightarrow{\text{mo}\,1} = \ldots = \xrightarrow{\text{mo}\,n}$$
$$\wedge\ \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N \text{ is sequentieel consistent in } SF/N$$
$$\wedge\ \forall\, m \in Mem :\ \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N \text{ is sequentieel consistent in } Op_m/N$$
$$\wedge\ \forall\, m \in Mem :\ \forall\, p \in P :\ \xrightarrow{\text{po}\,p} \cap\, Op_m^2 \subset \xrightarrow{\text{mo}\,p}$$
$$\wedge\ \forall\, p \in P :\ \left(\xrightarrow{\text{po}} \cap\, ((LF \times Op) \cup SF^2)\right) \subset \xrightarrow{\text{mo}\,p}$$

Zie appendix B.2 voor het bewijs van de equivalentie van TSO en TSO$^\star$. In figuur 3.11 staat een voorbeeld van een TSO-uitvoering. Dit voor-

*Instructietypes:* $Op = L_{ord} \cup S_{ord} \cup F_{ord}$.
*Programmaordening:* $;$ is een partiële ordening in $Op$, zodanig dat $;$ een geldige programmaordeningsrelatie is.
*Geheugenordening:* $\leq$ is een partiële ordening in $Op$, en $\leq$ is een totale ordening in $SF$.
*Ondeelbaarheid:* een lees/wijzig/schrijf-opdracht mag niet onderbroken worden door een opdracht op een andere processor op dezelfde locatie.
*Beëindiging:* als een processor een eindeloze reeks leesopdrachten uitvoert op een locatie, en een andere processor voert een schrijfopdracht uit naar die locatie, dan bestaat er een leesopdracht in die sequentie die de schrijfopdracht waarneemt.
*Waarde* , met $l \in LF$:

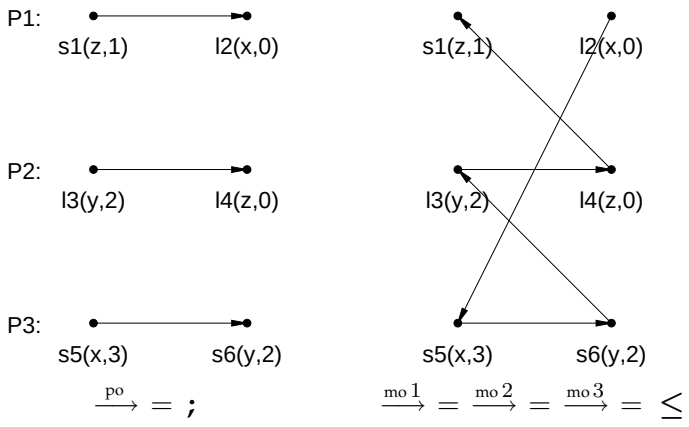
$$pred_{SPARC,\leq}(l) \triangleq \{s \in SF_{mem(l)} | s \neq l \wedge (s \leq l \vee s\,;\,l)\}$$

$$impred_{SPARC,\leq}(l) \triangleq \{s \in pred_{SPARC,\leq}(l) \mid \forall s' \in pred_{SPARC,\leq}(l) : s = s' \vee \neg(s \leq s')\}$$

$$\forall\, s_1 \in \max_{\leq}(l) :\ (\forall\, s_2 \in \max_{\leq}(l) :\ s_1 = s_2) \Longrightarrow val_l(l) = val_s(s_1).$$

*Lees-Opdracht:* $\forall\, l \in LF :\ \forall\, op \in Op :\ l\,;\,op \Longrightarrow l \leq op$.
*Schrijf-Schrijf:* $\forall\, op_1, op_2 \in SF :\ op_1\,;\,op_2 \Longrightarrow op_1 \leq op_2$.

**Tabel 3.9:** TSO-geheugenmodel voor dataopdrachten [SPA92].




**Figuur 3.11:** Voorbeeld van een TSO-uitvoering waarbij de relaties ; en $\leq$ conflicteren: $s_1$ ; $l_2$ en $l_2 \leq s_1$ gelden tezelfdertijd. Dit betekent dat opdrachten $s_1$ en $l_2$ werden uitgevoerd in een volgorde tegengesteld aan de programmaordening.

beeld illustreert tevens dat zowel de relaties $\leq$ als $\xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n}$ in het geheugenmodel TSO tegengesteld kunnen zijn aan de programmaordening.

Zowel uit de definitie van TSO als uit die van TSO$^\star$ is eenvoudig af te lezen dat geen herordening van een leesopdracht met een daaropvolgende opdracht is toegelaten, en dat ook geen herordening van twee opeenvolgende schrijfopdrachten is toegelaten. Terwijl uit de definitie van TSO rechtstreeks volgt dat een schrijfbuffer toegepast mag worden, blijkt uit de formulering van TSO$^\star$ onmiddellijk dat data-afhankelijke opdrachten niet mogen worden herordend. Deze laatste eigenschap is moeilijker af te leiden uit de oorspronkelijke TSO-definitie dan uit de definitie van TSO$^\star$.

### 3.8.8 Gedeeltelijke ordening schrijfopdrachten of PSO

Net zoals TSO is PSO (*partial store order*) een geheugenmodel dat voorkomt bij bestaande multiprocessors. In tegenstelling tot TSO neemt een processor de schrijfopdrachten van een andere processor niet noodzakelijk in programmaordening waar. Vandaar de naam gedeeltelijke ordening schrijfopdrachten. Om toch nog een volgorde te kunnen opleggen tussen opdrachten van verschillende processen vanuit het beschouwde programma heeft het PSO-geheugenmodel een opdracht

*Programmaordening, Geheugenordening, Ondeelbaarheid, Beëindiging, Waarde, Lees-Opdracht:* zie TSO.
*Instructietypes:* $Op = L \cup S \cup F \cup Bar_{ss}$.
*Schrijf-Schrijf:* $\forall\, op_1, op_2 \in SF :\ \forall\, s \in Bar_{ss} :\ op_1 \,;\, s \,;\, op_2 \Longrightarrow op_1 \leq op_2$.
*Schrijf-Schrijf-Zelfde:*
$\forall\, m \in Mem :\ \forall\, op_1, op_2 \in SF_m :\ op_1 \,;\, op_2 \Longrightarrow op_1 \leq op_2$.

**Tabel 3.10:** PSO-geheugenmodel voor dataopdrachten [SPA92].

die de herordening van schrijfopdrachten tegenhoudt, nl. een schrijf-schrijf-grensopdracht met de naam STBAR. Voor de definitie van PSO, zie tabel 3.10.

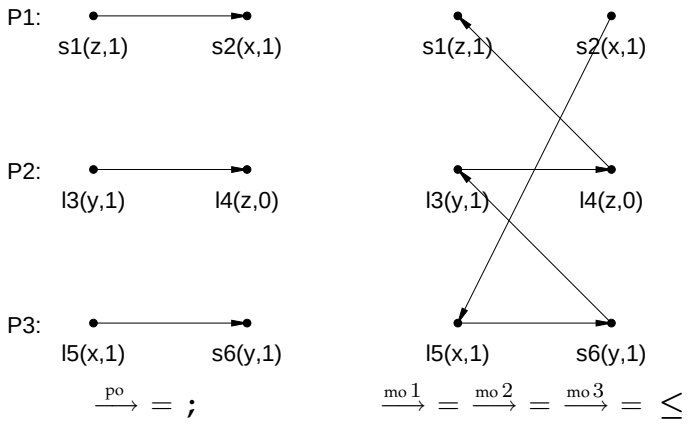
Het PSO-geheugenmodel bevat gewone lees- en schrijfopdrachten, en ook $bar_{ss}$-opdrachten op een processor. Op basis van het geheugenmodel PSO definiëren we het geheugenmodel PSO$^\star$, namelijk de uitbreiding van PSO met bijzondere opdrachten, andere grensopdrachten dan $bar_{ss}$ en grensopdrachten over meerdere processors. PSO$^\star$ is het geheugenmodel waarbij de basiseigenschappen (3.1) t.e.m. (3.7) gelden, en ook

$$
\begin{aligned}
&\xrightarrow{\text{mo}\,1} = \ldots = \xrightarrow{\text{mo}\,n} \\
&\wedge\ \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N \text{ is sequentieel consistent in } SF/N \\
&\wedge\ \forall\, m \in Mem :\ \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N \text{ is sequentieel consistent in } Op_m/N \\
&\wedge\ \forall\, m \in Mem :\ \forall\, p \in P :\ \xrightarrow{\text{po}\,p} \cap\, Op_m^2 \subset \xrightarrow{\text{mo}\,p} \\
&\wedge\ \forall\, p \in P :\ \left(\xrightarrow{\text{po}} \cap\, (LF \times Op)\right) \subset \xrightarrow{\text{mo}\,p}
\end{aligned}
$$

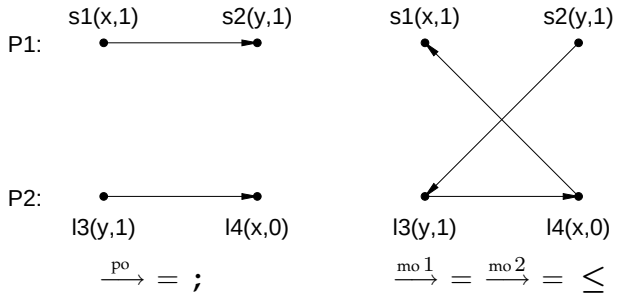
Zie appendix B.1 voor het bewijs van de equivalentie van PSO en PSO$^\star$.

Voorbeelden van PSO-uitvoeringen staan in figuren 3.12 en 3.13. De uitvoering in figuur 3.12 is niet mogelijk onder het TSO-geheugenmodel. Uit de verschillende eigenschappen van het TSO-geheugenmodel volgt namelijk dat $s_1 \xrightarrow{\text{mo}} s_2 \xrightarrow{\text{mo}} l_5 \xrightarrow{\text{mo}} s_6 \xrightarrow{\text{mo}} l_3 \xrightarrow{\text{mo}} l_4 \xrightarrow{\text{mo}} s_1$, wat niet kan omdat $\xrightarrow{\text{mo}}$ een partiële ordening moet zijn.

De uitvoering in figuur 3.13 is niet mogelijk onder het PCD-geheugenmodel. Uit de PCD-eigenschappen volgt namelijk dat $s_1 \xrightarrow{\text{mo}\,2} s_2$, $l_3 \xrightarrow{\text{mo}\,2} l_4$, $l_4 \xrightarrow{\text{mo}\,2} s_1$ en $s_2 \xrightarrow{\text{mo}\,2} l_3$ samen moeten voldaan zijn, wat niet kan.




**Figuur 3.12:** Eerste voorbeeld van een PSO-uitvoering.




**Figuur 3.13:** Tweede voorbeeld van een PSO-uitvoering.

### 3.8.9 Processorconsistentie van de DASH-multiprocessor of PCD

De DASH-multiprocessor, waarbij DASH staat voor *Directory Architecture for Shared Memory*, bestaat uit meerdere identieke deelsystemen met in elk deelsysteem meerdere processors en lokaal geheugen voor deze processors. Het is een directory-gebaseerd systeem met volledig gedistribueerd geheugen. Directory-gebaseerd betekent dat ergens wordt bijgehouden in welk lokaal geheugen elke gedeelte van het gemeenschappelijk geheugen werd geplaatst. De processors in een deelsysteem van de DASH-multiprocessor zijn MIPS-processors. Dit betekent dat de instructieset bestaat uit lees-, schrijf-, lees/wijzig/schrijf-opdrachten en grensopdrachten [LLG+90, LLJ+92, LLG+92, LW95]. Etiketten worden geïmplementeerd door met het adres van een opdracht een etiket te associëren. Deze aanpak kan eenvoudig vertaald worden naar de in dit hoofdstuk gevolgde aanpak. Het DASH-systeem biedt twee consistentievormen aan: een eigen vorm van processorconsistentie en release-consistentie. Het tweede geheugenmodel wordt verder besproken, en het eerste geheugenmodel wordt in deze paragraaf besproken. De definitie van PCD of processorconsistentie van de DASH-multiprocessor is als volgt [GLL+90]:

1. Vorig verwijst naar programmaordening.
2. Een opdracht is ofwel een leesopdracht ofwel een schrijfopdracht.
3. Een leesopdracht door processor $P_i$ wordt als uitgevoerd beschouwd t.o.v. processor $P_k$ op een zeker tijdstip als het uitvoeren van een schrijfopdracht door $P_k$ naar dezelfde locatie de waarde geretourneerd door de leesopdracht niet meer kan beïnvloeden.
4. Een schrijfopdracht door processor $P_i$ is wordt als uitgevoerd beschouwd t.o.v. processor $P_k$ op een zeker tijdstip als het uitvoeren van een leesopdracht door $P_k$ naar dezelfde locatie de waarde weggeschreven door de schrijfopdracht (of een volgende schrijfopdracht naar dezelfde locatie) retourneert.
5. Een opdracht is uitgevoerd als die uitgevoerd is t.o.v. alle processors.
6. Een leesopdracht is globaal uitgevoerd als die is uitgevoerd en ook de schrijfopdracht die de waarde geretourneerd door de leesopdracht genereerde, is uitgevoerd.

7. Uniprocessor controle- en data-afhankelijkheden moeten gerespecteerd worden.

8. Elke implementatie vermijdt deadlock doordat opdrachten voorafgaand aan een uitgevoerde opdracht ooit (globaal) worden uitgevoerd.

9. Alle schrijfopdrachten naar dezelfde locatie worden geserialiseerd, en worden in die volgorde uitgevoerd t.o.v. elke processor.

10. Vooraleer een leesopdracht mag worden uitgevoerd t.o.v. een andere processor, moeten alle voorgaande leesopdrachten uitgevoerd zijn.

11. Vóór een schrijfopdracht mag uitgevoerd worden t.o.v. een andere processor, moeten alle voorgaande lees- en schrijfopdrachten uitgevoerd zijn.

Alhoewel deze definitie onafhankelijk is van de organisatie van een systeem, heeft zij een eenvoudige interpretatie voor een gedistribueerd systeem waar elke processor enkel over een eigen lokaal geheugen beschikt. Het begrip *uitgevoerd* verwijst dan naar uitgevoerd zijn in het lokale geheugen van de betreffende processor, en het begrip *globaal uitgevoerd* verwijst dan naar uitgevoerd zijn in *alle* lokale geheugens.

Bovenstaande definitie van PCD is een operationele definitie: het gedrag wordt beschreven aan de hand van een mogelijke implementatie. Bovendien wordt er ondersteld dat er een totale ordening voor alle opdrachten bestaat: er wordt verwezen naar de volgorde in de tijd van gebeurtenissen op verschillende processors via termen als *uitgevoerd* en *tijdstip*. Uit deel PCD.5 van de PCD-definitie volgt dat alle processors de schrijfopdrachten in dezelfde totale volgorde waarnemen. Er geldt dus dat de ordeningen $\xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N$ sequentieel consistent zijn in *SF*.

Verder volgt uit PCD.3, PCD.4 en PCD.5 de betekenis van het begrip *uitgevoerd t.o.v. processor* $k$. "Een schrijfopdracht $s \in Op_i$ wordt als uitgevoerd beschouwd t.o.v. een leesopdracht $l \in Op_k$" is nl. equivalent met $s \xrightarrow{\text{mo}\,k} l$. Een opdracht $op_1$ is uitgevoerd t.o.v. een opdracht $op_2$ als en slechts als $\forall\, p \in P :\ op_1 \xrightarrow{\text{mo}\,p} op_2$. Eigenschappen PCD.6 en PCD.8 impliceren dat de connectiviteitseigenschap (3.8) geldt. Uit PCD.7 volgt dat uniprocessorcorrectheid, eigenschap (3.1) geldt. En uit PCD.10 en

| |
|---|
| $Op = L_{ord} \cup S_{ord}$<br>Enkel gewone lees- en schrijfopdrachten worden beschouwd. |
| $\forall p \in P: \xrightarrow{\text{po}} \cap SF^2 \subset \xrightarrow{\text{mo}\,p}$<br>Schrijfopdrachten worden in programmaordening uitgevoerd. |
| $\xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n}$ zijn sequentieel consistent in $SF$<br>Alle processors nemen schrijfopdrachten in dezelfde volgorde waar. |
| $\forall p \in P: \xrightarrow{\text{po}} \cap (LF \times LSF) \subset \xrightarrow{\text{mo}\,p}$<br>De volgorde van opdrachten na een leesopdracht blijft behouden. |
| Eig. (3.1), (3.3), (3.8)<br>Uniprocessorcorrectheid is van toepassing. |
| Eig. Eig. (3.2), (3.5), (3.7), (3.4), (3.6)<br>Geldig wegens zonder onderwerp. |

**Tabel 3.11:** Vertaling van PCD.

PCD.11 volgt dat $\forall p \in P: \xrightarrow{\text{po}} \cap (LF \times Op \cup SF^2) \subset \xrightarrow{\text{mo}\,p}$. De vertaalde definitie van PCD is samengevat in tabel 3.11.

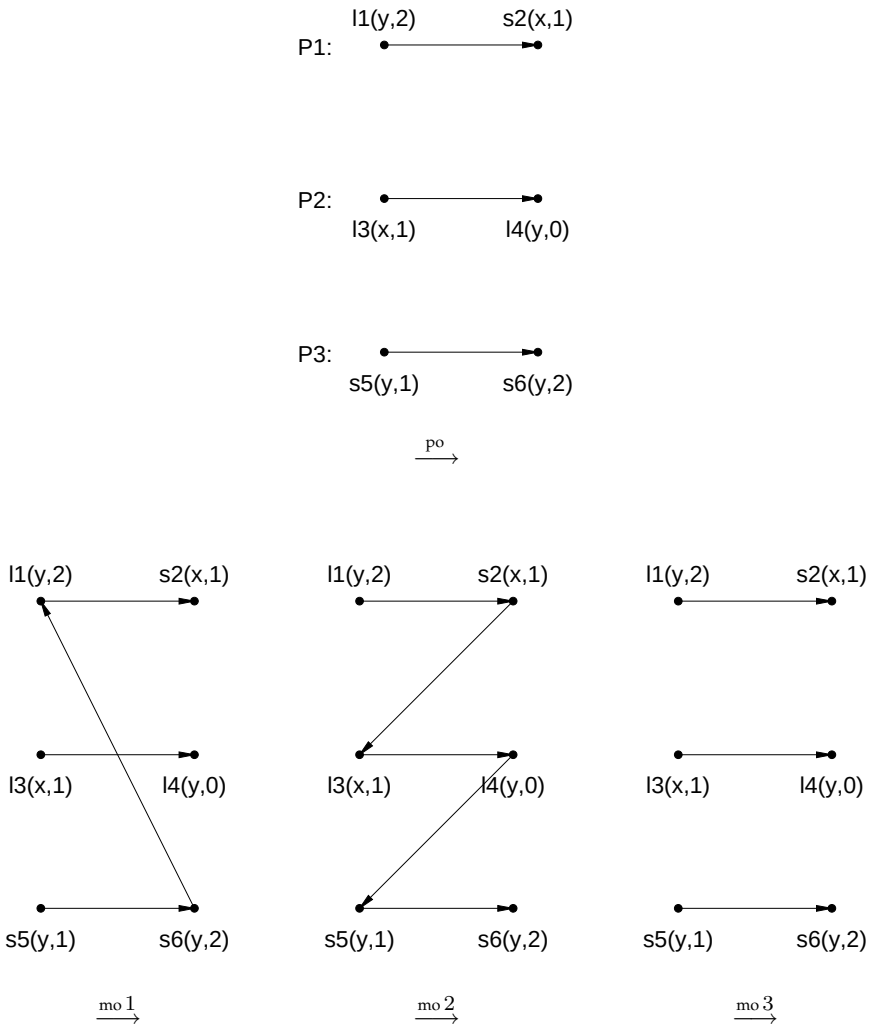
Als we de oorspronkelijke definitie van PCD uitbreiden met etiketten, synchronisatieopdrachten en lees/wijzig/schrijf-opdrachten, dan bekomen we het geheugenmodel PCD$^\star$: PCD$^\star$ is het geheugenmodel waarbij de basiseigenschappen (3.1) t.e.m. (3.7) gelden, (3.8) en (3.9) gelden, en waarbij bovendien ook geldt dat

$$\begin{array}{l} \xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n} \text{ zijn sequentieel consistent in } SF \\ \wedge\, \forall p \in P: \xrightarrow{\text{po}} \cap (SF^2 \cup LF \times LSF) \subset \xrightarrow{\text{mo}\,p} \end{array}$$

In figuur 3.14 staat een voorbeeld van een PCD-uitvoering. Dit voorbeeld voldoet niet aan het PSO-geheugenmodel omdat uit de PSO-eigenschappen namelijk volgt dat $\xrightarrow{\text{mo}\,1} = \xrightarrow{\text{mo}\,2} = \xrightarrow{\text{mo}\,3}$, $l_1 \xrightarrow{\text{mo}\,1} s_2$, $l_3 \xrightarrow{\text{mo}\,1} l_4$, $s_5 \xrightarrow{\text{mo}\,1} s_6$, $s_2 \xrightarrow{\text{mo}\,1} l_3$, $l_4 \xrightarrow{\text{mo}\,1} s_5$ en $s_6 \xrightarrow{\text{mo}\,1} l_1$ samen zouden moeten gelden, wat strijdig is met de eigenschap dat $\xrightarrow{\text{mo}\,1}$ een partiële ordening is.

### 3.8.10 Zwakke ordening of WO

Dubois definieert in zijn geheugenmodel *zwakke ordening* (*weak ordering*) twee soorten locaties, namelijk gewone locaties en synchroniserende locaties. De definitie van zwakke ordening volgens Dubois is als volgt:




**Figuur 3.14:** Voorbeeld van een PCD-uitvoering.

| |
|---|
| $Op = \{op \in (L \cup S) \mid lbl_1(op) \in \{ord_L, sync_L\} \wedge lbl_2(op) \in \{ord_L, sync_L\}\}$<br>Lees- en schrijfopdrachten met etiket $ord_L$ of $sync_L$ worden beschouwd. |
| Eig. (3.2)<br>Volgt rechtstreeks uit de definitie van WO. |
| Eig. (3.1), (3.3), (3.8), (3.9)<br>Overige eigenschappen gelden (impliciet in tekstuele definitie). |
| Eig. (3.5), (3.7), (3.4), (3.6)<br>Geldig wegens zonder onderwerp. |

**Tabel 3.12:** Vertaling van Dubois' definitie van zwakke ordening.

> Variabelen in een geheugen worden onderverdeeld in synchroniserende globale variabelen, gewone globale variabelen en lokale variabelen. In een multiprocessor zijn opdrachten zwak geordend als
>
> 1. opdrachten op globale synchroniserende variabelen sterk geordend zijn, en als
> 2. geen enkele opdracht op een synchroniserende variabele door een processor wordt gelanceerd vóór alle voorgaande opdrachten op globale data uitgevoerd zijn, en als
> 3. geen enkele opdracht op globale data wordt gelanceerd door een processor voor een voorgaande opdracht op een synchroniserende variabele uitgevoerd is [DSB86].

Mits het begrip lees- of schrijfopdracht naar een synchroniserende variabele vervangen wordt door het equivalente begrip lees- of schrijfopdracht met etiket $sync_L$ naar een gemeenschappelijke variabele, dan is de formele vertaling van het PCD-model zoals in tabel 3.12.

Ook het geheugenmodel WO kunnen we uitbreiden met de andere opdrachttypes en etiketten. Naargelang we de keuze maken of de bijzondere opdrachten in de uitbreiding sequentieel consistent of processorconsistent zijn bekomen we de geheugenmodellen WOsc$^\star$ resp. WOpc$^\star$. In beide modellen is aan de basiseigenschappen voldaan. Voor WOsc$^\star$ geldt bovendien dat

$$(Op_s, \xrightarrow{\text{po}}, \xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n}) \text{ is SC in } Op_s.$$

De definitie voor WOpc$^\star$ is analoog aan de definitie van WOsc$^\star$. Merk op dat de definitie hierboven van WOsc$^\star$ verschillend is van Gharachorloo's definitie van WOsc$^\star$. Gharachorloo geeft in zijn WOsc$^\star$ de $acq_L$- en $rel_L$-etiketten dezelfde semantiek als de $sync_L$-etiketten. Hier werd ervoor gekozen de semantiek van de verschillende etiketten in alle geheugenmodellen gelijk te houden. Dit impliceert verder dat WOsc$^\star$ en RCsc$^\star$ dezelfde geheugenmodellen zijn, evenals WOpc$^\star$ en RCpc$^\star$.

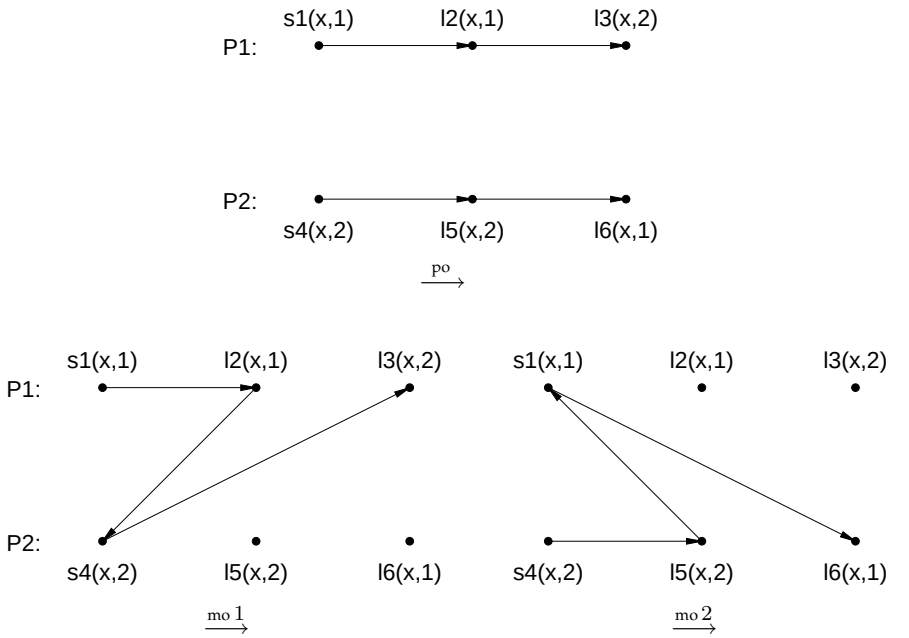
In figuren 3.15 en 3.16 staan voorbeelden van uitvoeringen mogelijk onder het geheugenmodel WO. Het eerste voorbeeld is ook mogelijk onder de geheugenmodellen PRAM, RC en LRC maar is niet mogelijk onder het geheugenmodel IA-64. Uit de IA-64 eigenschappen (3.1), (3.3) en (3.9) samen volgt namelijk dat $s_1 \xrightarrow{\text{mo}\,1} l_2 \xrightarrow{\text{mo}\,1} s_4 \xrightarrow{\text{mo}\,1} l_3$ en ook dat $s_4 \xrightarrow{\text{mo}\,1} l_5 \xrightarrow{\text{mo}\,1} s_1 \xrightarrow{\text{mo}\,1} l_6$. Dit is strijdig met de vereiste dat $\xrightarrow{\text{mo}\,1}$ een partiële ordening moet zijn.

Het tweede voorbeeld in figuur 3.16 voldoet aan WO, RC en LRC. Er bestaan voor dit voorbeeld echter geen totaal geordende $\xrightarrow{\text{mo}}/N$-relaties die voldoen aan de eigenschap dat elke $\xrightarrow{\text{mo}\,p}$ een totale ordening is en eigenschap (3.3) samen. Het door $\xrightarrow{\text{po}}$ voorgestelde gedrag komt bijvoorbeeld voor in release-consistente DSM-systemen.

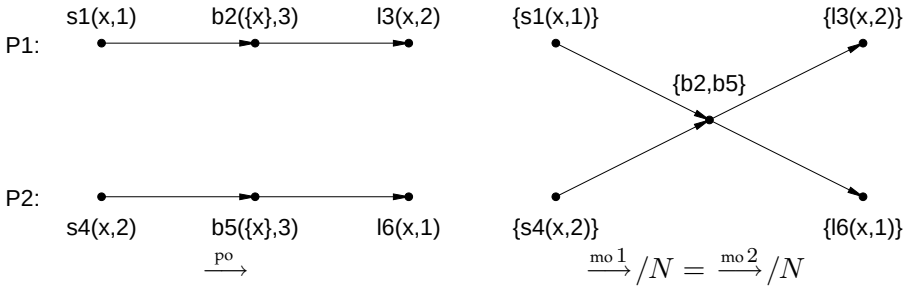
### 3.8.11 Release-consistentie of RC

Het geheugenmodel release-consistentie (*release consistency*) gaat nog een stap verder dan zwakke ordening in het verminderen van de ordening gegarandeerd door het geheugensysteem. Terwijl bij zwakke ordening voor de bijzondere opdrachten (etiket $sync_L$) de volgorde met voorgaande en volgende opdrachten gegarandeerd is, geldt dit bij release-consistentie enkel voor opdrachten met etiket $sync_L$. De ordening voor opdrachten met etiket $acq_L$ of $rel_L$ is enkel gegarandeerd tegenover opdrachten die in programmaordening na resp. voor de opdracht voorzien van het genoemde etiket komen. Er bestaan twee varianten van release-consistentie: ofwel met sequentieel consistente bijzondere opdrachten, ofwel met processorconsistente bijzondere opdrachten. Gharachorloo definieert release-consistentie met sequentieel consistente bijzondere opdrachten of **RCsc** als volgt:

> *Vooraleer een gewone lees- of schrijfopdracht uitgevoerd mag worden t.o.v. om het even welke processor, moeten alle voorgaan-*




**Figuur 3.15:** Eerste voorbeeld van een WO-uitvoering.




**Figuur 3.16:** Tweede voorbeeld van een WO-uitvoering.

| |
|---|
| $Op = L \cup S$<br>Lees- en schrijfopdrachten met elk etiket worden beschouwd. |
| $(Op_s, \xrightarrow{\text{po}}, \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N)$ is SC in $Op_s/N$<br>Bijzondere opdrachten zijn onderling sequentieel consistent. |
| Eig. (3.1), (3.2), (3.3), en (3.8)<br>Overige eigenschappen uit [GLL$^+$90]. |

**Tabel 3.13:** Vertaling van Gharachorloo's definitie van release-consistentie met sequentieel consistente bijzondere opdrachten.

> *de acquire-opdrachten uitgevoerd zijn. Voor een release-opdracht mag uitgevoerd worden t.o.v. om het even welke processor, moeten alle voorgaande gewone lees- en schrijfopdrachten worden uitgevoerd. Bijzondere opdrachten worden onderling sequentieel consistent uitgevoerd [GLL$^+$90].*

De betekenis van de term *uitvoeren* is dezelfde als in de definitie van PCD uit paragraaf 3.8.9. Ook de eigenschappen 3 t.e.m. 8 uit het PCD-geheugenmodel blijven van toepassing.

De definitie van **RCpc** is analoog aan die van RCsc, behalve dat in het model RCpc de bijzondere opdrachten onderling processorconsistent, d.w.z. PCD worden uitgevoerd. De definities van PCD, RCsc en RCpc staan in hetzelfde artikel [GLL$^+$90]. Gharachorloo gebruikt in zijn tekst de naam processorconsistentie om het geheugenmodel PCD aan te duiden, terwijl in deze tekst de afkorting PC verwijst naar de historisch eerste definitie van een geheugenmodel met de naam processorconsistentie.

De benaming release-consistentie verwijst naar een mogelijke implementatie van dit geheugenmodel: het is mogelijk onder de geheugenmodellen RCsc/RCpc de consistentie te verzekeren door enkel bij het uitvoeren van een opdracht met release-etiket de waarden in gewijzigde locaties naar de andere processors door te sturen.

Een formele vertaling van Gharachorloo's definitie staat in tabel 3.13. Breiden we de definitie van RCsc uit met lees/wijzig/schrijfopdrachten en met synchronisatieopdrachten, dan bekomen we het geheugenmodel **RCsc**$^\star$ : RCsc$^\star$ is het geheugenmodel waar eigenschappen (3.1) t.e.m. (3.7) en eigenschap (3.8) gelden, en ook geldt dat

$$(Op_s, \xrightarrow{\text{po}}, \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N) \text{ is SC in } Op_s/N.$$

De definitie voor RCpc is analoog, maar dan met sequentiële consistentie (SC) vervangen door processorconsistentie (PCD).

### 3.8.12 Luie release-consistentie of LRC

Terwijl het reeds bij een multiprocessor zeer belangrijk is de communicatie tussen processors te reduceren om goede prestaties te behalen, is dit nog belangrijker bij een netwerk van werkstations (NOW of *network of workstations*). Vandaar dat ook voor NOW's nog steeds onderzoek verricht wordt naar het reduceren van coherentietrafiek. Enkele technieken die voor dit doel toegepast worden zijn:

- Breid het DSM-systeem uit met synchronisatieprimitieven (lock, unlock, barrier). Gewoonlijk zijn deze primitieven geïmplementeerd als functies in een meegeleverde functiebibliotheek.
- Stel het doorsturen van veranderingen zo lang mogelijk uit. Dit betekent bijvoorbeeld pas bij de uitvoering van een unlock-opdracht de veranderingen globaal bekend maken (*Eager release consistency*), ofwel dat bij de uitvoering van een lock-opdracht andere processors ondervraagd worden over voorbije veranderingen (*Lazy release consistency* of luie release-consistentie).

Meerdere auteurs, vooral van publicaties over DSM-systemen, gebruiken de benamingen acquire en release voor de hierboven beschreven lock- en unlock-opdrachten [CBZ91, KCZ92, KDCZ94, ACD+96, Car95, DCZ96, HP96, ISL98, IS99]. Deze laatste opdrachten hebben een fundamenteel andere semantiek dan degene die Gharachorloo oorspronkelijk aan de benamingen acquire en release gaf [GLL+90]. De semantiek van acquire en release versus lock en unlock is verschillend om de volgende redenen:

- Gharachorloo's acquire en release zijn etiketten bij lees- en schrijfopdrachten, en geen afzonderlijke opdrachten.
- De acquire- en release-etiketten zorgen voor het tegenhouden van herordening op een zelfde processor, terwijl lock en unlock dienen om een ordening tussen processors op te leggen.
- Een acquire-opdracht en een release-opdracht kunnen pas een ordening tussen processors opleggen als de acquire-opdracht in een lus wordt uitgevoerd. Een lock-opdracht in een lus uitvoeren leidt tot deadlock.

| |
|---|
| $Op = L_{ord} \cup S_{ord} \cup Lock \cup Unlock$<br>Gewone lees- en schrijfopdrachten en ook lock- en unlock-opdrachten worden beschouwd. |
| $\forall\, op \in Sync :\ mem(op) = Mem$<br>Een lock- of unlock-opdracht verwijst naar alle locaties. |
| Eig. (3.1), (3.2), (3.3), (3.5), (3.7), (3.4), (3.6)<br>Overige eigenschappen uit [KCZ92]. |
| Eig. (3.8) en (3.9) gelden *niet*<br>Connectiviteit en coherentie zijn niet van toepassing. |

**Tabel 3.14:** Vertaling van Kelehers definitie van LRC.

Uit [CBZ91, KCZ92] blijkt dat met de benamingen acquire en release in de context van luie release-consistentie wel degelijk de semantiek van lock- en unlock-opdrachten wordt bedoeld. Vandaar dat verder deze laatste benamingen zullen worden gebruikt.

Luie release-consistentie volgens Keleher is het geheugenmodel waarbij lees- en schrijfopdrachten worden beschouwd, en ook lock- en unlock-opdrachten. De connectiviteitseigenschap is niet van toepassing. Behalve de basiseigenschappen zijn er geen beperkingen op de geheugenordeningsrelaties. Voor een formele vertaling, zie tabel 3.14.

De uitvoeringstijd voor een parallelle applicatie op een multiprocessor met een release-consistent geheugen vergeleken met een sequentieel consistent geheugen kan tot een factor 2.33 korter zijn [GGH91b]. Voor DSM-systemen is het prestatieverschil tussen release-consistente en sequentieel consistente DSM-systemen omwille van de tragere communicatie nog groter. Release-consistentie presteert in deze gevallen beter omdat bij release-consistentie tussen synchronisatieopdrachten geen coherentie wordt vereist.

De veralgemening van LRC resulteert in het geheugenmodel LRC$^\star$, waarbij de basiseigenschappen (3.1), (3.2), (3.3), (3.5), (3.7), (3.4) en (3.6) gelden.

Daar in WO, RC en LRC niet vereist wordt dat elke geheugenordening een totale ordening is, betekent dit dat in een implementatie van één van deze geheugenmodellen geen communicatie tussen processors noodzakelijk is zolang er geen synchronisatieopdrachten worden uitgevoerd. Dit is een voordeel voor multiprocessors en een groot voordeel voor DSM-systemen. Het geheugenmodel luie release-consistentie wordt o.m. toegepast in de DSM-systemen Munin [Car95]

| opdrachttype | IA-64 notatie | eigen notatie |
|---|---|---|
| leesopdracht | *ld* $r_1 = [r_2]$ | $(l, ord_L, i, p, \{r_2\}, r_1)$ |
| $r_1 := [r_2]$ | *ld.acq* $r_1 = [r_2]$ | $(l, acq_L, i, p, \{r_2\}, r_1)$ |
| schrijfopdracht | *st* $[r_2] = r_1$ | $(s, ord_L, i, p, \{r_2\}, r_1)$ |
| $[r_2] := r_1$ | *st.rel* $[r_2] = r_1$ | $(s, rel_L, i, p, \{r_2\}, r_1)$ |
| at. omw. | *xchg* $r_1 = [r_3], r_2$ | $(f, ord_L, ord_L, i, p, \{r_3\}, r_1, r_2)$ |
| at. voorw. omw. | *cmpxchg* $r_1 = [r_3], r_2$ | $(f, ord_L, ord_L, i, p, \{r_3\}, r_1, r_2)$ |
| at. verhogen | *fetchadd* $r_1 = [r_3], \Delta$ | $(f, ord_L, ord_L, i, p, \{r_3\}, r_1, r_1{+}\Delta)$ |
| | *fetchadd.acq* $r_1 = [r_3], \Delta$ | $(f, acq_L, acq_L, i, p, \{r_3\}, r_1, r_1{+}\Delta)$ |
| | *fetchadd.rel* $r_1 = [r_3], \Delta$ | $(f, rel_L, rel_L, i, p, \{r_3\}, r_1, r_1{+}\Delta)$ |
| grensopdracht | *mf* | $(bar_{ss}, i, p, Mem, j)$; $(bar_{ll}, i+1, p, Mem, j+1)$ |

**Tabel 3.15:** Verband tussen de notatie gehanteerd in de beschrijving van de IA-64 architectuur en de eigen notatie. $r_x$ staat voor de inhoud van een IA-64 register, en $[r_x]$ voor de inhoud van locatie met adres $r_x$. De grensopdracht *mf* stemt overeen met twee opdrachten in het hier voorgestelde formalisme, de andere IA-64 opdrachten met één opdracht. De symbolen $i$ en $j$ staan voor een toepasselijk instructienummer resp. grensopdracht-identificatienummer.

en TreadMarks[KCZ92, KDCZ94, ACD+96, KCDZ95].

### 3.8.13 Geheugenmodel van de IA-64 architectuur

In de IA-64 architectuur zijn naast de lees- en schrijfopdrachten (ld resp. st) ook atomische lees/wijzig/schrijf-opdrachten beschikbaar, nl. xchg, cmpxchg en fetchadd. Deze opdrachten staan resp. voor het uitwisselen van de waarde in een gegeven locatie met de waarde in een register, voor de voorwaardelijke uitwisseling van de waarde in een gegeven locatie met de waarde in een register en vóór het optellen van een constante bij de waarde in een gegeven locatie. Bij alle opdrachten wordt het etiket $ord_L$ ondersteund. Bij de atomische opdrachten cmpxchg en fetchadd kunnen ook de etiketten $ord_L$, $acq_L$ en $rel_L$ gebruikt worden. De opdrachten ld en st ondersteunen elk een etiket naast $ord_L$: nl. $acq_L$ resp. $rel_L$– zie ook tabel 3.15. De *memory fence* opdracht *mf* ordent zowel de lees- en schrijfopdrachten vóór en na deze grensopdracht onderling [Int99].

In tabel 3.15 wordt de relatie tussen IA-64 opdrachten en de in dit hoofdstuk gedefinieerde opdrachten weergegeven; een samenvatting van het geheugenmodel van de IA-64 architectuur staat in tabel 3.16.

Als we het IA-64 geheugenmodel veralgemenen met opdrachten met het $sync_L$-etiket en met lock- en unlock-opdrachten, dan bekomen

| |
|---|
| $Op = \{op \in LSF \mid lbl_1(op) \in \{ord_L, acq_L, rel_L\} \vee lbl_2(op) \in \{ord_L, acq_L, rel_L\}\} \cup Bar$<br>Alle opdrachten behalve die met etiket $sync_L$ worden beschouwd. |
| $\forall\, m \in Mem :\ \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N$ sequentieel consistent in $Op_m/N$<br>Coherentie per locatie geldt. |
| Eig. (3.1)<br>Uniprocessorcorrectheid geldt<br>(zie paragraaf 4.4.7 blz. p4-23 in [Int99]). |
| Eig. (3.2) en (3.6)<br>Zie tabel 4-20 blz. 4-24 in [Int99]. |
| Eig. (3.9)<br>Coherentie geldt (zie paragraaf 4.4.6.2 op blz. 4-23 in [Int99]). |
| Eig. (3.5), (3.7), (3.3), (3.8).<br>Impliciet in [Int99]. |
| Eig. (3.4).<br>Geldig wegens zonder onderwerp. |

**Tabel 3.16:** Vertaling van Intels definitie van het IA-64 geheugenmodel.

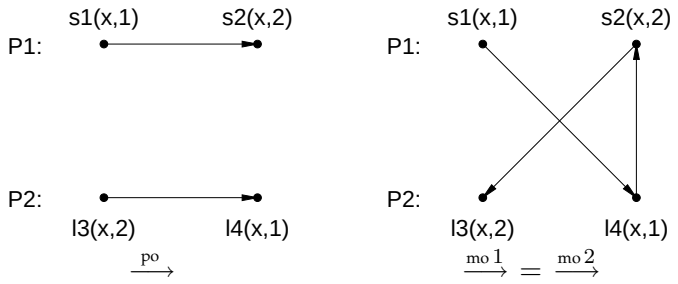
we het geheugenmodel **IA-64**$^\star$ : dit is het geheugenmodel waarbij de basiseigenschappen (3.1) t.e.m. (3.7) gelden, en ook de opdrachten per locatie totaal geordend zijn:

$$\forall\, m \in Mem :\ \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N \text{ is sequentieel consistent in } Op_m/N.$$

In figuur 3.17 is een uitvoering voorgesteld die mogelijk is onder het IA-64 geheugenmodel en ook onder het geheugenmodel PRAM – zie ook figuur 3.8. Deze uitvoering is niet mogelijk onder de geheugenmodellen PSO of PC omdat bij deze geheugenmodellen vereist wordt dat twee opeenvolgende lees- of schrijfopdrachten in volgorde worden uitgevoerd.

## 3.9 Uitbreiding naar berichtendoorgave

Door een gepaste interpretatie te geven aan de symbolen uit het voor geheugenmodellen ingevoerd formalisme kan niet alleen de werking van een gemeenschappelijk-geheugensysteem beschreven worden, maar ook de werking van een systeem gebaseerd op berichtencommunicatie. Er bestaan zelfs overeenkomsten tussen geheugenmodellen en berichtendoorgaveparadigma's.




**Figuur 3.17:** Voorbeeld van een IA-64-uitvoering.

### 3.9.1 Vertaling tussen geheugenopdrachten en berichtendoorgave

Een bericht heeft inhoud $v_s$ en wordt ontvangen via of verstuurd naar een kanaal $m$. Zowel de inhoud van een bericht als het kanaal waarlangs een bericht verstuurd wordt, worden vastgelegd bij verzending. Deze verzending wordt voorgesteld als een schrijfopdracht $s(m, v_s)$. De ontvangst van een bericht uit het kanaal $m$ heeft als resultaat de verzonden waarde $v_l$. Formeel wordt dit voorgesteld door de uitgevoerde opdracht $l(m, v_l)$. Berichten worden ontvangen uit het kanaal waarlangs ze verstuurd werden. Deze herinterpretatie van de symbolen $s$,$l$,$m$,$v_s$ en $v_l$ is samengevat in tabel 3.17. Uit wat volgt zal blijken dat etiketten bij opdrachten vervallen, en dat synchronisatieopdrachten hun betekenis behouden.

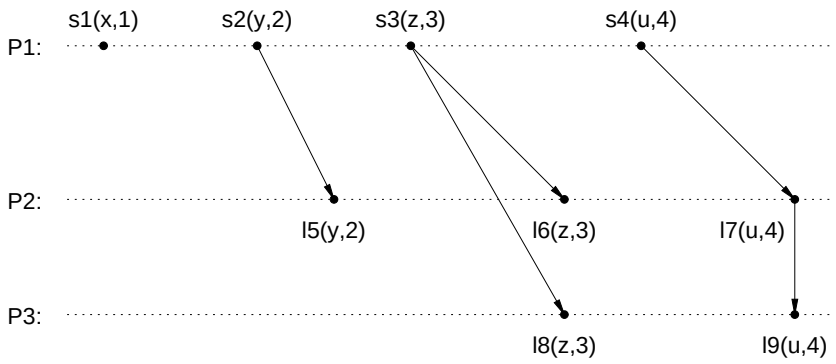
De verschillende relaties $\xrightarrow{\text{mo}\,p}$ stellen nu de volgorde van zend- en ontvangstopdrachten voor. Naargelang de vorm van deze relaties worden verschillende communicatiepatronen voorgesteld: ofwel een bericht dat verstuurd werd zonder ontvangen te worden, ofwel een bericht dat in unicast verstuurd werd, ofwel een bericht dat in multicast verstuurd werd. Dit is voorgesteld in figuur 3.18.

### 3.9.2 Basiseigenschappen berichtendoorgavemodellen

Er dient onderscheid gemaakt te worden tussen twee soorten berichtendoorgave, nl. tussen actieve berichtendoorgave en klassieke berichtendoorgave. Onder **actieve berichtendoorgave** wordt verstaan dat een bericht met inhoud $v$ via kanaal $m$ verstuurd wordt en na verzending ook afgeleverd wordt bij de ontvanger, zonder dat de ontvanger

| | |
|---|---|
| $v_s \in$ *Val* | Inhoud van een verstuurd bericht. |
| $v_l \in$ *Val* | Inhoud van een ontvangen bericht. |
| $m \in$ *Mem* | Identificatie kanaal. |
| $s(m, v_s) \in S$ | Opdracht tot versturen van een bericht. |
| $l(m, v_l) \in L$ | Ophalen inhoud ontvangen bericht. |
| $f(m, v_s, v_l) \in F$ | Ontvangen, wijzigen en versturen van een bericht. |

**Tabel 3.17:** Correspondentie tussen gemeenschappelijk-geheugenbegrippen en berichtencommunicatie.




**Figuur 3.18:** De drie mogelijke verzendingswijzen voor een bericht: verloren gegaan bericht ($s_1$), unicast ($s_2$) en multicast ($s_3$ en $s_4$). In de figuur zijn de berichtenordeningen $\xrightarrow{\text{mo}\,1} = \xrightarrow{\text{mo}\,2} = \xrightarrow{\text{mo}\,3}$ voorgesteld. $x$, $y$ en $z$ zijn de namen van de kanalen. De volgorde van $l_6$ en $l_8$ resp. $l_7$ en $l_9$ is van geen belang zolang $s_4 \xrightarrow{\text{mo}} l_9$ blijft gelden.

daarbij hoeft in te grijpen. In wat volgt gaan we ervan uit dat dit gebeurt door bij de ontvanger de inhoud $v$ op locatie $m$ te schrijven. De ontvanger kan dan de inhoud van de boodschap waarnemen door de waarde aanwezig op locatie $m$ te lezen. Synchronisatie tussen zenden en ontvangen dient te gebeuren door andere opdrachten dan de zend- en ontvangstopdrachten. De semantiek van deze zend- en ontvangstopdrachten wordt weerspiegeld in eigenschap (3.3), de waarde van een leesopdracht. Actieve berichtendoorgave en modellen voor actieve berichtendoorgave zijn direct verwant aan geheugenmodellen. Daar in wat voorafging geheugenmodellen reeds uitgebreid werden besproken, wordt hier niet verder ingegaan op actieve berichtendoorgave.

Bij toepassing van **klassieke berichtendoorgave** wordt een verzonden bericht na aankomst bij de ontvanger bijgehouden in een wachtrij. Een opdracht tot ontvangen van een bericht betekent dat het eerste bericht dat in de wachtrij werd geplaatst uit de wachtrij verwijderd wordt, ofwel dat gewacht wordt tot het volgende bericht ontvangen wordt bij een lege wachtrij. De semantiek van klassieke berichtendoorgave verschilt sterk van die van een gemeenschappelijk-geheugensysteem, vandaar dat voor berichtendoorgave andere basiseigenschappen gelden. Deze basiseigenschappen volgen hieronder.

Bij berichtencommunicatie veronderstellen we dat alle processen zend- en ontvangstopdrachten in dezelfde volgorde waarnemen:

$$\xrightarrow{\text{mo}\,1} = \ldots = \xrightarrow{\text{mo}\,n} \tag{3.10}$$

Omdat er slechts één geheugenordeningsrelatie bestaat, wordt in wat volgt $\xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n}$ vervangen door $\xrightarrow{\text{mo}}$.

Zendopdrachten voor een gegeven kanaal $m$, dit zijn de opdrachten in verzameling $SF_m$, gebeuren onderling of t.o.v. ontvangstopdrachten, dit zijn de opdrachten uit $LF_m$, nooit gelijktijdig maar altijd in een bepaalde volgorde:

$$\xrightarrow{\text{mo}} \text{ is een totale ordening in } LSF_m^2 \setminus L_m^2 \tag{3.11}$$

In bovenstaande eigenschap werd volgende identiteit toegepast: $SF_m^2 \cup LF_m \times SF_m \cup SF_m \times LF_m = LSF_m^2 \setminus L_m^2$.

Voor het vastleggen van het resultaat van een ontvangstopdracht definiëren we de bijkomende functie $C_m()$ en de relatie SR. Met de functie $C_m(op)$ stellen we het aantal elementen voor uit de door relatie $R_m = \xrightarrow{\text{mo}} \cap SF_m^2$ totaal geordende verzameling $SF_m$ dat gelijk is aan of

voorafgaat aan opdracht $op$. Hierbij is $op \in LSF$ en $m = mem(op)$. De definitie van $C_m()$ is als volgt:

$$C_m(op_2) \triangleq \#\{op_1 \in SF_m | op_1 \xrightarrow{\text{mo}} op_2\}.$$

In combinatie met eigenschappen 3.10 en 3.11 volgt uit de definitie van de functie $C_m()$ verder dat:

$$\forall\, s_1, s_2 \in SF_m :\ C_m(s_1) = C_m(s_2) \iff s_1 = s_2,$$

en ook dat

$$\forall\, op_1, op_2 \in LSF_m :\ C_m(op_1) < C_m(op_2) \implies op_1 \xrightarrow{\text{mo}} op_2.$$

De relatie SR wordt gedefinieerd als een relatie tussen zend- en ontvangstopdrachten, en geldt voor een zendopdracht $s$ en een ontvangstopdracht $l$ enkel als $l$ en $s$ hetzelfde kanaal gebruiken, als $l$ en $s$ opdrachten uit verschillende processen zijn en als opdracht $l$ een opdracht is die het bericht verstuurd door opdracht $s$ ontvangt. Met $m \in mem(s)$ en $m \in mem(l)$ is de definitie van relatie SR als volgt:

$$s \,\text{SR}\, l \iff C_m(s) = C_m(l) \wedge proc(s) \neq proc(l) \wedge s \xrightarrow{\text{mo}} l$$

Uit de definitie van relatie SR volgt dat als zendopdracht $s$ een bericht in multicast verstuurt, en als het bericht verstuurd door opdracht $s$ door zowel opdrachten $l_1$ en $l_2$ ontvangen wordt, dat dan opdrachten $l_1$ en $l_2$ noodzakelijk tot verschillende processen behoren:

$$\forall\, s \in SF_m :\ \forall\, l_1, l_2 \in LF_m :\ s \,\text{SR}\, l_1 \wedge s \,\text{SR}\, l_2 \implies proc(l_1) \neq proc(l_2)$$

Het **resultaat van een ontvangstopdracht** $l \in L_{m,p}$ wordt vastgelegd aan de hand van onderstaande eigenschap:

$$s \,\text{SR}\, l \implies val_l(l) = val_s(s) \tag{3.12}$$

Bovenstaande definities en eigenschap impliceren samen dat de opdracht die eerst verstuurd werd ook eerst wordt ontvangen. Uit de definitie van de functie $C_m()$ en uit de eigenschap dat zendopdrachten per kanaal totaal geordend zijn (3.11) volgt onmiddellijk de eigenschap dat de functie $C_m()$ en de relatie $\xrightarrow{\text{mo}}$ dezelfde ordening opleggen in verzameling $SF_m$:

$$\begin{aligned}\forall\, s_1, s_2 \in SF_m :\ s_1 \xrightarrow{\text{mo}} s_2 &\iff C_m(s_1) \leq C_m(s_2) \\ \wedge\ s_1 = s_2 &\iff C_m(s_1) = C_m(s_2).\end{aligned}$$

Er is niet alleen een verband tussen relatie $\xrightarrow{\text{mo}}$ en de functie $C_m()$ voor zendopdrachten, maar ook voor ontvangstopdrachten. Dit verband is echter minder strikt:

$$\forall\, op_1, op_2 \in LSF_m :\ op_1 \xrightarrow{\text{mo}} op_2 \Longrightarrow C_m(op_1) \leq C_m(op_2) \\ \wedge\, C_m(op_1) < C_m(op_2) \Longrightarrow op_1 \xrightarrow{\text{mo}} op_2.$$

Verder wordt vereist dat voor elke ontvangstopdracht een corresponderende zendopdracht moet bestaan:

$$\forall\, l \in LF_m :\ \exists\, s \in SF_m :\ s\,\text{SR}\,l \tag{3.13}$$

De basiseigenschappen (3.1), (3.2), (3.4), (3.5), (3.6), en (3.7) blijven gelden.

Uit bovenstaande definities volgt de eigenschap dat indien zendopdrachten in een gegeven volgorde werden uitgevoerd, dat dan de corresponderende ontvangstopdrachten in dezelfde volgorde worden uitgevoerd:

$$\forall\, m \in Mem :\ \forall\, s_1, s_2 \in SF_m :\ \forall\, l_1, l_2 \in LF_m : \\ s_1 \neq s_2 \wedge s_1\,\text{SR}\,l_1 \wedge s_2\,\text{SR}\,l_2 \\ \Longrightarrow (s_1 \xrightarrow{\text{mo}} s_2 \wedge l_1 \xrightarrow{\text{mo}} l_2) \vee (s_2 \xrightarrow{\text{mo}} s_1 \wedge l_2 \xrightarrow{\text{mo}} l_1)$$

Zie lemma B.3.1 voor het bewijs van bovenstaande eigenschap.

Uit de eigenschap 3.12, waarde van een ontvangstopdracht genaamd, blijkt dat indien voor een ontvangstopdracht $l$ er geen corresponderende zendopdracht is, dat dan de door opdracht $l$ ontvangen waarde niet gedefinieerd is. Bij het uitvoeren van een programma waarin deze situatie zich voordoet zal de ontvangstopdracht niet termineren.

### 3.9.3 Ordeningen bij berichtencommunicatie

Garg beschouwt vier ordeningen bij berichtencommunicatie [Gar96]. Het belang van deze ordeningen is dat een ordening aangepast aan de beschouwde toepassing toelaat deze toepassing eenvoudiger te implementeren. Garg beschouwt berichten die alle via hetzelfde berichtenkanaal verstuurd worden, en definieert onderstaande ordeningen voor berichtencommunicatie op basis van de tijdstippen waarop berichten verstuurd en ook ontvangen worden als volgt:

- Bij **synchrone berichtenordening** (SM) kan het tijdsdiagram zo getekend worden dat alle pijlen die de verzending van een bericht voorstellen verticaal kunnen worden getekend. Dit betekent ook dat aan alle opdrachten een tijdstip kan worden toegekend, strikt stijgend volgens programmaordening. Ontvangstopdrachten krijgen hetzelfde tijdstip toegekend als de corresponderende zendopdracht.
- **Causale berichtenordening** (CM) betekent dat als een verzending van een eerste bericht causaal voorafgaat aan de verzending van een tweede bericht, en beide berichten worden door hetzelfde proces ontvangen, dan wordt het eerste bericht voor het tweede ontvangen.
- Berichtencommunicatie voldoet aan **FIFO-ordening** als elke twee berichten die van proces $i$ naar proces $j$ verzonden worden ook in die volgorde door proces $j$ worden ontvangen.
- Bij **asynchrone ordening** (AM) zijn er naast de voor alle berichtencommunicatie geldende eigenschappen geen verdere beperkingen op de volgorde van berichten.

In wat volgt breiden we deze definities uit naar berichtencommunicatie met meerdere berichtenkanalen en ook met synchronisatieopdrachten.

Garg definieert synchrone communicatie door aan elke opdracht een tijdstip toe te kennen. Dit betekent dat er voor alle tijdstippen samen een totale ordening bestaat, consistent met de verstuurde en ontvangen berichten. Deze ordening moet zodanig zijn dat elke zendopdracht onmiddellijk voorafgaat aan de volgens relatie SR ermee corresponderende ontvangstopdracht. Een mogelijke definitie van het communicatiemodel uitgebreide synchrone communicatie, $\mathbf{SM}^\star$ , is dus het model waarbij de basiseigenschappen voor berichtencommunicatie gelden, samen met volgende bijkomende eigenschappen:

$$\begin{array}{l} \xrightarrow{\text{mo}}/N \text{ is een totale ordening in } Op/N \\ \wedge\ \xrightarrow{\text{po}} \subset \xrightarrow{\text{mo}} \\ \wedge\ \forall\, p \in P :\ \forall\, s, l \in LSF :\ s\,\text{SR}\,l \Longrightarrow s\ (\xrightarrow{\text{mo}} \setminus L^2)^-\ l. \end{array}$$

Dit model wordt ook totaal geordend [WMK95] of logisch ogenblikkelijk [SI95] genaamd.

Stel dat $s_1$ en $l_1$ de versturing resp. ontvangst van een eerste bericht voorstellen, en dat $s_2$ en $l_2$ de versturing resp. ontvangst van een tweede bericht voorstellen. Dan gelden $s_1\,\text{SR}\,l_1$ en $s_2\,\text{SR}\,l_2$. De verzending

van het eerste bericht gaat causaal vooraf aan de verzending van het tweede bericht als en slechts als $s_1 \xrightarrow{\text{mo}} s_2$ geldt. De twee vermelde berichten worden in volgorde ontvangen als en slechts als $l_1 \xrightarrow{\text{po}} l_2$ geldt. Het communicatiemodel **CM**$^\star$ voldoet dus aan de basiseigenschappen voor berichtencommunicatie samen met de volgende eigenschappen:

$$\begin{aligned}&\xrightarrow{\text{po}} \subset \xrightarrow{\text{mo}}\\ &\wedge \forall\, s_1, s_2 \in SF_m : \forall\, l_1, l_2 \in LF_m :\\ &\quad (s_1 \,\text{SR}\, l_1 \wedge s_2 \,\text{SR}\, l_2) \Longrightarrow \neg(s_1 \xrightarrow{\text{mo}} s_2 \wedge l_2 \xrightarrow{\text{po}} l_1).\end{aligned}$$

In het FIFO-communicatiemodel moeten berichten die door een proces verstuurd werden en beide door een ander proces ontvangen worden in volgorde aankomen. Het **FM**$^\star$ -communicatiemodel is dus het communicatiemodel dat voldoet aan de basiseigenschappen voor berichtencommunicatie samen met de volgende eigenschappen:

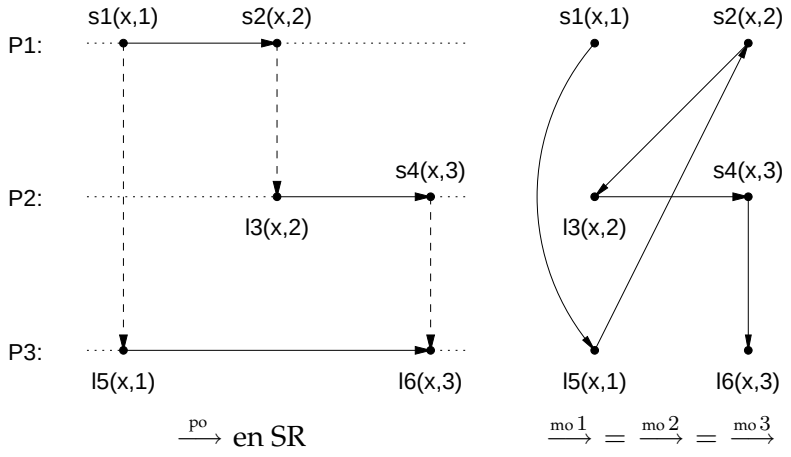
$$\begin{aligned}&(\xrightarrow{\text{po}} \cap\, SF^2) \subset \xrightarrow{\text{mo}}\\ &\wedge \forall\, s_1, s_2 \in SF_m : \forall\, l_1, l_2 \in LF_m :\\ &(s_1 \,\text{SR}\, l_1 \wedge s_2 \,\text{SR}\, l_2) \Longrightarrow \neg(s_1 \xrightarrow{\text{po}} s_2 \wedge l_2 \xrightarrow{\text{po}} l_1).\end{aligned}$$

Asynchrone berichtencommunicatie legt naast de hogervermelde algemene eigenschappen geen bijkomende eigenschappen op, dus definiëren we het model **AM**$^\star$ als het model waarbij de basiseigenschappen voor berichtencommunicatie gelden.

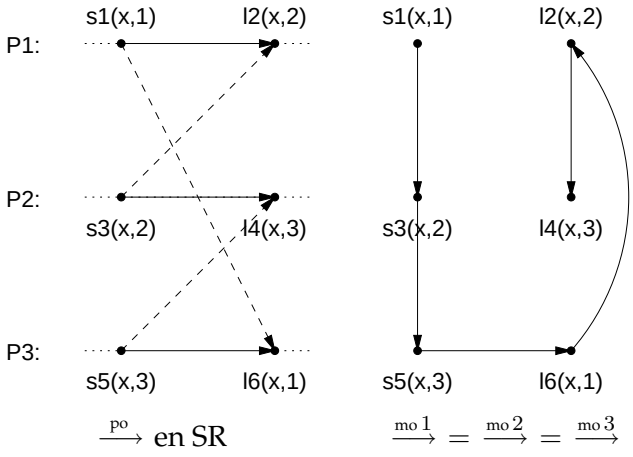
Voor elk van de berichtencommunicatiemodellen SM$^\star$, CM$^\star$, FM$^\star$ en AM$^\star$ staat een voorbeeld van een uitvoering in de figuren 3.19, 3.20, 3.21 en 3.22. De uitvoering uit figuur 3.20 voldoet aan CM maar niet aan synchrone ordening. Het FM-voorbeeld uit figuur 3.21 voldoet niet aan causale ordening omdat causale ordening vereist dat tezelfdertijd $s_1 \,\text{SR}\, l_6$, $s_4 \,\text{SR}\, l_5$, $s_1 \xrightarrow{\text{mo}} s_4$ en $l_5 \xrightarrow{\text{po}} l_6$ gelden. Dit is strijdig met de definitie van causale ordening. De AM-uitvoering voorgesteld in figuur 3.22 voldoet niet aan FM-ordening omdat voor dat voorbeeld geldt dat $s_1 \,\text{SR}\, l_4$, $s_2 \,\text{SR}\, l_3$, $s_1 \xrightarrow{\text{po}} s_2$ samen met $l_3 \xrightarrow{\text{po}} l_4$, wat strijdig is met de definitie van FIFO-ordening.

### 3.9.4 Multicast

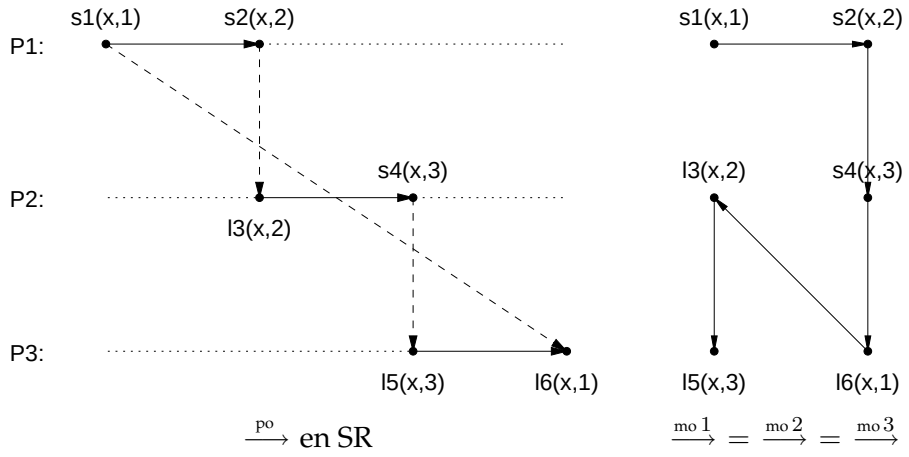
Bovenstaande definities en eigenschappen gelden zowel voor verzending van berichten in unicast als in multicast. Verzending in unicast wordt gemodelleerd door één enkele ontvangstopdracht te laten corresponderen met één enkele zendopdracht via relatie SR. Verzending



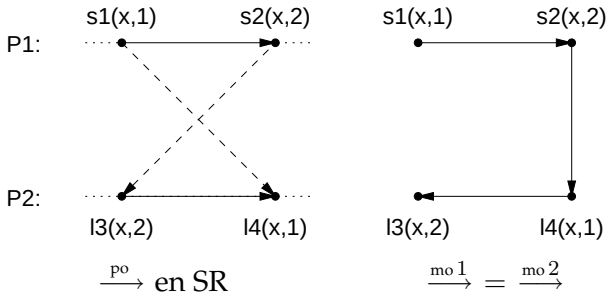

**Figuur 3.19:** Voorbeeld van communicatie die voldoet aan synchrone berichtenordening (SM). In de figuur is links de relatie $\xrightarrow{\text{po}}$ in volle lijn voorgesteld, en in streeplijn de verstuurt-ontvangt relatie SR. Rechts staan mogelijke relaties $\xrightarrow{\text{mo}\,1} = \xrightarrow{\text{mo}\,2} = \xrightarrow{\text{mo}\,3}$.




**Figuur 3.20:** Voorbeeld van communicatie die voldoet aan causale berichtenordening (CM).




**Figuur 3.21:** Voorbeeld van communicatie die voldoet aan FIFO-berichtenordening (FM).




**Figuur 3.22:** Voorbeeld van communicatie die voldoet aan asynchrone berichtenordening (AM).

in multicast wordt gemodelleerd door twee of meer ontvangstopdrachten te laten corresponderen via relatie SR met een zendopdracht.

## 3.10 Bespreking modellen en hun eigenschappen

### 3.10.1 Predikaten uit definities modellen

Alle predikaten die voorkomen in een van de definities van geheugenmodellen of berichtencommunicatiemodellen uit dit hoofdstuk zijn samengevat in tabel 3.18. De predikaten in deze tabel zijn onderverdeeld in zes groepen. De eerste vier groepen hebben betrekking op geheugenmodellen, de vijfde groep op beide soorten modellen en de laatste groep op berichtencommunicatiemodellen. De groepen eigenschappen voor geheugenmodellen zijn: de eigenschap (a) die een verband legt tussen opdrachten en de uitvoeringstijdstippen van elke opdracht, de eigenschappen (b), (c), (d), (e), (f), (g), (i) en (j) die de geheugenordening van opdrachten bepalen, de eigenschappen (k), (l), (m), (n) en (o) die vastleggen welke herordening van opdrachten t.o.v. de programmaordening is toegelaten, en de eigenschappen (p) en (q). Deze laatste eigenschappen zijn combinaties van de voorgaande eigenschappen, maar dan toegepast op de bijzondere opdrachten i.p.v. de gewone. Dit is nodig om de modellen RCsc en RCpc te kunnen voorstellen. De vijfde groep eigenschappen, de eigenschappen die zowel voor geheugenmodellen als voor berichtencommunicatiemodellen gelden, is samengevat in (r). Als laatste volgen de eigenschappen specifiek voor berichtencommunicatiemodellen, nl. (s), (t), (u) en (v).

### 3.10.2 Vergelijking van geheugenmodellen en berichtencommunicatiemodellen

Met elk geheugenmodel of berichtencommunicatiemodel *Mod* wordt een verzameling geldige uitvoeringen $E = (Op, \xrightarrow{\text{po}})$ geassocieerd. Die verzameling zullen we hier ook noteren als *Mod*. Indien voor twee modellen $Mod_1$ en $Mod_2$ deze verzamelingen gelijk zijn, dan zijn beide modellen **equivalent**. Als daarentegen $Mod_1 \subsetneq Mod_2$ geldt, dan is model $Mod_1$ **sterker** dan model $Mod_2$. De benaming sterker verwijst naar het gegeven dat het model $Mod_1$ meer restrictieve eigenschappen oplegt aan de toegelaten uitvoeringen dan model $Mod_2$. Analoog is $Mod_2 \subsetneq Mod_1$ equivalent met de eigenschap dat model $Mod_1$ **zwak-**

**ker** is dan model $Mod_2$. Geldt daarentegen geen van de voorgaande uitdrukkingen, dan zijn modellen $Mod_1$ en $Mod_2$ **onvergelijkbaar**. De relatie *is sterker dan* is een partiële ordening over modellen, gezien de eigenschappen van $\subset$ over verzamelingen.

Gezien de manier waarop in voorgaande paragrafen de verschillende modellen gedefinieerd werden, volstaat een goed gekozen particuliere uitvoering om aan te tonen dat een model niet sterker is dan een ander model. Stel dat de uitvoering $E$ voldoet aan model $Mod_1$, maar niet aan model $Mod_2$. Er volgt dat $Mod_1 \not\subset Mod_2$, en dus dat model $Mod_1$ niet sterker kan zijn dan model $Mod_2$. Twee goed gekozen voorbeelden volstaan dus om aan te tonen dat twee modellen onvergelijkbaar zijn. Om een eigenschap als $Mod_2 \subset Mod_1$ aan te tonen dient echter bewezen te worden dat deze geldt voor elke uitvoering $E \in Mod_2$. Gelijkheid van modellen kan worden aangetoond via wederzijdse inclusie.

De vergelijking van modellen kan als volgt systematisch aangepakt worden. In tabel 3.18 staan een aantal predikaten opgesomd met betrekking tot de verzameling $Op$ en de relaties $\xrightarrow{\text{po}}$, $\xrightarrow{\text{mo}\,1}$ $\ldots$ $\xrightarrow{\text{mo}\,n}$. Alle in deze tekst gedefinieerde modellen zijn opgebouwd als een conjunctie van deze predikaten, en wel als volgt: de uitvoering $E = (Op, \xrightarrow{\text{po}})$ voldoet aan model $Mod$ als en slechts als er relaties $\xrightarrow{\text{mo}\,1}$ $\ldots$ $\xrightarrow{\text{mo}\,n}$ bestaan zodanig dat $(Op, \xrightarrow{\text{po}}, \xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n})$ voldoet aan alle predikaten in de verzameling $Q = \{P_1 \ldots P_k\}$. De vraag rijst nu of er meerdere verzamelingen van predikaten $Q$ bestaan voor een zelfde model. Uit tabel 3.19 volgt dat dit zeker het geval is. Indien de predikaten $Q$ het model $Mod$ voortbrengen, en predikaat (c) behoort tot de predikatenverzameling $Q$, dan mag eigenschap $(f)$ zowel toegevoegd als weggelaten worden uit $Q$ zonder dat dit iets aan het model verandert. Als de predikaten $Q$ gelden voor de uitvoering $E = (Op, \xrightarrow{\text{po}}, \xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n})$, en als men deze predikaten $Q$ met de eigenschappen in tabel 3.19 transformeert tot de predikaten $Q'$, dan gelden predikaten $Q'$ ook voor uitvoering $E$.

In tabel 3.20 correspondeert elke rij met een model, en correspondeert elke kolom met een predikaat. Alhoewel in het algemeen met elk model meerdere predikatenverzamelingen $Q, Q', \ldots$ overeenstemmen, stemt hier met elk beschouwd model één enkele predikatenverzameling overeen. Het bewijs daarvoor bestaat uit twee facetten: enerzijds moet aangetoond worden dat de verzameling uitvoeringen toegelaten door de predikaten $Q$ identiek is aan $Mod$, en anderzijds moet aangetoond worden dat de andere predikaten onmogelijk kunnen gelden

voor het beschouwde model. De respectieve bewijzen van het eerste facet zijn opgenomen in een appendix – zie ook de meest rechtse kolom in tabel 3.20, en het tweede facet wordt gestaafd met de voorbeelden van uitvoeringen in eerdere paragrafen.

De vergelijking van modellen wordt door tabel 3.20 sterk vereenvoudigd. Als een model $Mod_2$ aan alle predikaten voldoet waar $Mod_1$ aan voldoet, dan geldt dat $Mod_1 \subseteq Mod_2$. Het tegengestelde, $Mod_1 \not\subset Mod_2$, kan ook afgeleid worden uit de tabel. De aldus verkregen ordening van geheugenmodellen is voorgesteld in figuur 3.23.

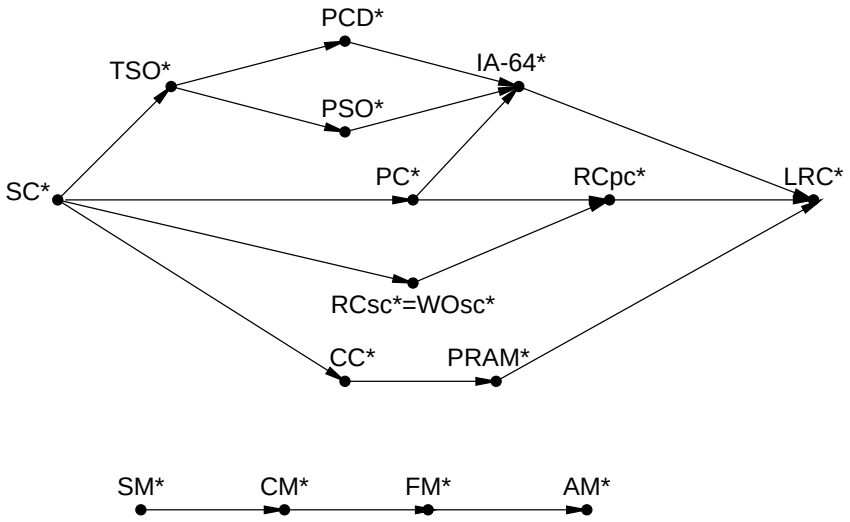
### 3.10.3 Interpretatie predikaten

Van de eigenschappen in tabel 3.18 zijn er enkele die een eenvoudige interpretatie hebben. De eigenschap **atomiciteit van de schrijfopdrachten** bijvoorbeeld is voldaan als voor alle processen op basis van om het even welke uitvoering onder het beschouwde geheugenmodel geen onderscheid kan worden gemaakt met de situatie waarin alle schrijfopdrachten voor alle processen ogenblikkelijk plaatsgrijpen. Dit betekent dat schrijfopdrachten atomair worden uitgevoerd als voor elk proces de schrijfopdrachten totaal geordend zijn, en bovendien deze totale ordening voor elk proces dezelfde is. Dit wordt uitgedrukt door eigenschap (i). In de literatuur komt deze eigenschap voor onder de namen *write synchronization* bij Collier [Col92] en *write atomicity* bij Adve [AG96].

Een andere belangrijke eigenschap is **coherentie per locatie**, of de eigenschap waarbij in elke geheugenordening de opdrachten per locatie in dezelfde volgorde worden waargenomen. Dit is de eigenschap (g). Deze eigenschap blijkt te gelden voor alle geheugenmodellen behalve voor de zgn. release-consistente modellen WO, RC en LRC.

### 3.10.4 Connectiviteit en LRC

Een belangrijk aspect van een geheugenmodel is de hoeveelheid communicatie die nodig is om de consistentie opgelegd door een geheugenmodel in stand te houden. In het bijzonder bij DSM-systemen is de eigenschap dat het geheugenmodel geen communicatie vereist tussen opeenvolgende synchronisatiepunten in een programma belangrijk. Dit betekent dat eigenschap (g) niet mag gelden. Eigenschap (g) is niet van toepassing voor de verschillende geheugenmodellen gebaseerd op release-consistentie, vandaar dat recente DSM-systemen meestal geba-




**Figuur 3.23:** Rangschikking van geheugen- en berichtencommunicatiemodellen volgens de relatie *is sterker dan*.

seerd zijn op release-consistentie. Alhoewel release-consistente DSM-systemen zoals LRC de connectiviteitseigenschap (3.8) niet garanderen, is het ontbreken van deze eigenschap geen belemmering voor de correcte werking van parallelle programma's. Het is namelijk zo dat voor programma's zonder data-races het resultaat van het programma onafhankelijk is van het al of niet gelden van de connectiviteitseigenschap – zie ook paragraaf 2.4.

### 3.10.5 Vergelijking geheugenmodellen en berichtencommunicatiemodellen

De gegevens in tabel 3.20 laten niet alleen toe geheugenmodellen onderling te vergelijken of berichtencommunicatiemodellen onderling te vergelijken, maar maken het ook mogelijk geheugenmodellen en berichtencommunicatiemodellen met elkaar te vergelijken. Uit tabel 3.20 volgt onmiddellijk dat zoals verwacht geheugenmodellen en berichtencommunicatiemodellen fundamenteel verschillen. Alle geheugenmodellen en geen enkel berichtencommunicatiemodel voldoen namelijk aan eigenschap (3.3), terwijl geen enkel geheugenmodel en alle berichtencommunicatiemodellen voldoen aan eigenschap ((s)). Stel nu dat we uit de besproken modellen nieuwe modellen afleiden door aan de defi-

| Naam | Eigenschap |
|---|---|
| (a) | $\xrightarrow{\text{mo}\,1} = \ldots = \xrightarrow{\text{mo}\,n} = <_t$ |
| (b) | $\xrightarrow{\text{mo}\,1} = \ldots = \xrightarrow{\text{mo}\,n}$ |
| (c) | $\forall\, p \in P :\ \xrightarrow{\text{mo}\,p}/N$ is een totale ordening in $Op/N$ |
| (d) | $\xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N$ zijn sequentieel consistent in $Op/N$ |
| (e) | $\forall\, p \in P :\ (\xrightarrow{\text{po}} \cup \mapsto)^\star \subset \xrightarrow{\text{mo}\,p}$ |
| (f) | $\forall\, p \in P :\ \forall\, m \in Mem :\ \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N$ zijn t.o. in $Op_m/N$ |
| (g) | $\forall\, m \in Mem :\ \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N$ zijn s.c. in $Op_m/N$ |
| (h) | $\forall\, m \in Mem :\ \xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n}$ zijn s.c. in $LSF_m$ |
| (i) | $\xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N$ zijn sequentieel consistent in $SF/N$ |
| (j) | $\forall\, m \in Mem :\ \xrightarrow{\text{mo}\,1}/N \ldots \xrightarrow{\text{mo}\,n}/N$ zijn s.c. in $SF_m/N$ |
| (k) | $\forall\, p \in P :\ \xrightarrow{\text{po}} \subset \xrightarrow{\text{mo}\,p}$ |
| (l) | $\forall\, p \in P :\ \xrightarrow{\text{po}} \cap\, SF^2 \subset \xrightarrow{\text{mo}\,p}$ |
| (m) | $\forall\, p \in P :\ \xrightarrow{\text{po}} \cap\, (LF \times LSF) \subset \xrightarrow{\text{mo}\,p}$ |
| (n) | $\forall\, p \in P :\ \xrightarrow{\text{po}\,p} \subset \xrightarrow{\text{mo}\,p}$ |
| (o) | $\forall\, p \in P :\ \forall\, m \in Mem :\ \xrightarrow{\text{po}\,p} \cap\, L_m^2 \subset \xrightarrow{\text{mo}\,p}$ |
| (p) | (b), (c), (f), (n) en (o) toegepast op $Op_s$ i.p.v. $Op$ |
| (q) | (i), (j), (l) en (m) toegepast op $Op_s$ i.p.v. $Op$ |
| (r) | (3.1), (3.2), (3.4), (3.5), (3.6) en (3.7) |
| (s) | (3.10), (3.11), (3.12) en (3.13) |
| (t) | $\forall\, s, l \in LSF :\ s\,\mathrm{SR}\,l \Longrightarrow s\,(\xrightarrow{\text{mo}} \setminus\, L^2)^-\, l$ |
| (u) | $\forall\, s_1, s_2 \in SF_m :\ \forall\, l_1, l_2 \in LF_m :$ $(s_1\,\mathrm{SR}\,l_1 \wedge s_2\,\mathrm{SR}\,l_2) \Longrightarrow \neg(s_1 \xrightarrow{\text{mo}} s_2 \wedge l_2 \xrightarrow{\text{po}} l_1)$ |
| (v) | $\forall\, s_1, s_2 \in SF_m :\ \forall\, l_1, l_2 \in LF_m :$ $(s_1\,\mathrm{SR}\,l_1 \wedge s_2\,\mathrm{SR}\,l_2) \Longrightarrow \neg(s_1 \xrightarrow{\text{po}} s_2 \wedge l_2 \xrightarrow{\text{po}} l_1)$ |

**Tabel 3.18:** Predikaten uit de definities van geheugen- en berichtencommunicatiemodellen, onderverdeeld in eigenschappen i.v.m. de uitvoeringstijdstippen van opdrachten, coherentie-eigenschappen, herordeningseigenschappen, algemeen geldende eigenschappen, algemene en tenslotte modelspecifieke berichtencommunicatie-eigenschappen. Zie ook lemma B.3.2 op blz. 159.

| Eigenschap |
|---|
| (b) $\Longrightarrow$ (3.9) |
| (b) $\wedge$ (c) $\Longleftrightarrow$ (d) |
| (b) $\wedge$ (n) $\Longrightarrow$ (k) |
| (b) $\Longleftrightarrow$ (3.10) |
| (d) $\wedge$ (n) $\Longrightarrow$ (p) |
| (c) $\Longrightarrow$ (f) |
| (c) $\wedge$ (3.9) $\Longrightarrow$ (g) |
| (d) $\Longrightarrow$ (g) $\wedge$ (i) |
| (e) $\Longrightarrow$ (k) |
| (f) $\wedge$ (3.9) $\Longrightarrow$ (g) |
| (g) $\Longrightarrow$ (h) $\wedge$ (j) $\wedge$ (3.9) |
| (h) $\wedge$ (3.2) $\Longrightarrow$ (g) |
| (h) $\wedge$ (b) $\Longrightarrow$ (3.11) |
| (i) $\Longrightarrow$ (j) |
| (i) $\wedge$ (l) $\wedge$ (m) $\Longrightarrow$ (q) |
| (k) $\Longrightarrow$ (l) $\wedge$ (m) $\wedge$ (n) $\wedge$ (o) |
| (m) $\Longrightarrow$ (o) |
| (n) $\Longrightarrow$ (o) |
| (s) $\Longrightarrow$ (b) $\wedge$ (f) $\wedge$ (g) |
| (t) $\wedge$ (m) $\Longrightarrow$ (u) |
| (u) $\Longrightarrow$ (v) $\wedge$ (k) |
| (3.3) $\wedge$ (j) $\Longrightarrow$ (t) |

**Tabel 3.19:** Eigenschappen van geheugenmodelpredikaten geldig voor een gegeven uitvoering $E = (Op, \xrightarrow{\text{po}})$ en bij behoud van de $\xrightarrow{\text{mo}}$-relaties. Zie ook lemma B.3.2

| | (a) | (b) | (c) | (e) | (f) | (g) | (i) | (j) | (k) | (l) | (m) | (n) | (o) | (p) | (q) | (3.3) | (3.8) | (3.9) | (r) | (s) | (t) | (u) | (v) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AC⋆ | **+** | **+** | **+** | + | + | + | + | + | + | + | + | **+** | + | + | + | **+** | **+** | **+** | **+** | - | - | - | - | |
| SC⋆ | - | **+** | **+** | + | + | + | + | + | + | + | + | **+** | + | + | + | **+** | **+** | **+** | **+** | - | - | - | - | |
| CC⋆ | - | - | **+** | **+** | + | - | - | - | + | + | + | + | + | - | - | **+** | **+** | - | **+** | - | - | - | - | B.4.1 |
| CCA⋆ | - | - | **+** | **+** | + | - | - | - | + | + | + | + | + | - | - | **+** | **+** | - | **+** | - | - | - | - | B.4.1 |
| PRAM⋆ | - | - | **+** | - | + | - | - | - | - | - | - | **+** | + | - | - | **+** | **+** | - | **+** | - | - | - | - | B.4.5 |
| PC⋆ | - | - | + | - | + | + | - | **+** | - | **+** | - | **+** | + | - | - | **+** | **+** | **+** | **+** | - | - | - | - | B.4.6 |
| TSO⋆ | - | **+** | + | - | + | + | **+** | + | - | **+** | **+** | - | + | - | + | **+** | **+** | **+** | **+** | - | - | - | - | B.2.1 |
| PSO⋆ | - | **+** | + | - | + | + | **+** | + | - | - | **+** | - | + | - | - | **+** | **+** | **+** | **+** | - | - | - | - | B.1.1 |
| PCD⋆ | - | - | + | - | + | + | **+** | + | - | **+** | **+** | - | + | - | + | **+** | **+** | **+** | **+** | - | - | - | - | B.4.7 |
| IA-64⋆ | - | - | + | - | + | **+** | - | + | - | - | - | - | - | - | - | **+** | **+** | **+** | **+** | - | - | - | - | B.4.8 |
| WOsc⋆ | - | - | - | - | - | - | - | - | - | - | - | - | - | **+** | **+** | **+** | **+** | - | **+** | - | - | - | - | |
| RCsc⋆ | - | - | - | - | - | - | - | - | - | - | - | - | - | **+** | **+** | **+** | **+** | - | **+** | - | - | - | - | |
| RCpc⋆ | - | - | - | - | - | - | - | - | - | - | - | - | - | - | **+** | **+** | **+** | - | **+** | - | - | - | - | |
| LRC⋆ | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | **+** | - | - | **+** | - | - | - | - | |
| SM⋆ | - | **+** | + | - | **+** | + | + | + | **+** | + | + | + | + | + | + | - | **+** | **+** | **+** | **+** | **+** | + | + | B.4.9 |
| CM⋆ | - | **+** | + | - | **+** | + | + | + | **+** | + | + | + | + | + | + | - | **+** | **+** | **+** | **+** | - | **+** | + | B.4.9 |
| FM⋆ | - | **+** | + | - | **+** | + | + | + | - | + | - | - | - | - | - | - | **+** | **+** | **+** | **+** | - | - | **+** | B.4.10 |
| AM⋆ | - | **+** | + | - | **+** | + | + | + | - | + | - | - | - | - | - | - | **+** | **+** | **+** | **+** | - | - | - | B.4.10 |

**Tabel 3.20:** Tabelsgewijze weergave van de definities van de verschillende geheugenmodellen en berichtenordeningen voor eindige uitvoeringen. Een plusteken duidt aan dat de betreffende eigenschap van toepassing is, en een minteken betekent dat de betreffende eigenschap niet van toepassing is in het geheugenmodel. Eigenschappen die rechtstreeks uit de definitie volgen zijn aangeduid met **+** en afgeleide eigenschappen met +. In de laatste kolom staat een verwijzing naar het bewijs van de afgeleide eigenschappen.

nities van de verschillende modellen de eigenschappen 3.3 en ((s)) toe te voegen. De modellen worden dan vergelijkbaar. Daar eigenschappen 3.3 en ((s)) gelden, volgt uit tabel 3.19 dat ook volgende eigenschappen gelden: (b), (f), (g), (j), (3.9), en (t). Door eigenschappen (3.3) en ((s)) aan de bestaande geheugenmodellen toe te voegen bekomen we nieuwe geheugenmodellen. Deze nieuwe geheugenmodellen duiden we aan met AC$^{\star\star}$, SC$^{\star\star}$, enz. De modellen SC$^{\star}$, CC$^{\star}$, CCA$^{\star}$, PCD$^{\star}$, TSO$^{\star}$, PSO$^{\star}$, SM$^{\star}$ en CM$^{\star}$ worden via de eigenschappen in tabel 3.19 en via bovenstaande transformatie alle omgezet in hetzelfde model: SC$^{\star\star}$ = CC$^{\star\star}$ = CCA$^{\star\star}$ = PCD$^{\star\star}$ = TSO$^{\star\star}$ = PSO$^{\star\star}$ = SM$^{\star\star}$ = CM$^{\star\star}$. Dit model voldoet aan eigenschappen (b) t.e.m. (v) in tabel 3.18. Bij het voorbeeld in figuur 3.19 werd de geheugenordeningsrelatie $\xrightarrow{\text{mo}}$ zodanig gekozen dat het voorbeeld ook voldoet aan de modellen SC$^{\star\star}$ t.e.m. CM$^{\star\star}$.

Uit de transformatie van SC$^{\star}$ naar SC$^{\star\star}$ is duidelijk dat door de ingevoerde modeltransformatie de semantiek van de modellen grondig verandert. Vandaar dat niet verder op deze transformatie wordt ingegaan.

Raynal definieert o.m. sequentiële en causale consistentie en ook logisch ogenblikkelijke en causale berichtencommunicatie [RS97]. Daarbij toont Raynal het verband aan tussen sequentiële consistentie en logisch ogenblikkelijke communicatie enerzijds en ook tussen causale consistentie en causale communicatie anderzijds. Verdere implicaties van het vergelijken van geheugenmodellen en berichtencommunicatiemodellen, zoals SC$^{\star\star}$ = CC$^{\star\star}$ = SM$^{\star\star}$ = CM$^{\star\star}$, worden niet beschouwd in [RS97].

## 3.11 Voorbeelden van implementaties van geheugensystemen

Ter illustratie volgen hier enkele voorbeelden van implementaties van geheugensystemen. Deze voorbeelden werden gekozen om het verband tussen programmaordening, geheugenordening en een implementatie aan te tonen. Het zijn zeker geen optimale implementaties.

Een implementatie bestaat erin de functionaliteit van een geheugensysteem onder te verdelen in modules, deze modules te implementeren en een communicatieprotocol te definiëren tussen de modules. De keuzes gemaakt bij elk van deze taken hebben een belangrijke invloed op de prestaties en kostprijs van een geheugensysteem. De bovenstaande

| Opdracht | Verstuurd bericht | Ontvangen antwoord |
|---|---|---|
| $l(m, v_l)$ | lees $m$ | waarde $v_l$ |
| $s(m, v_s)$ | schrijf $v_s$ in $m$ | stuur volgende opdracht |

**Tabel 3.21:** Communicatieprotocol voor lees- en schrijfopdrachten

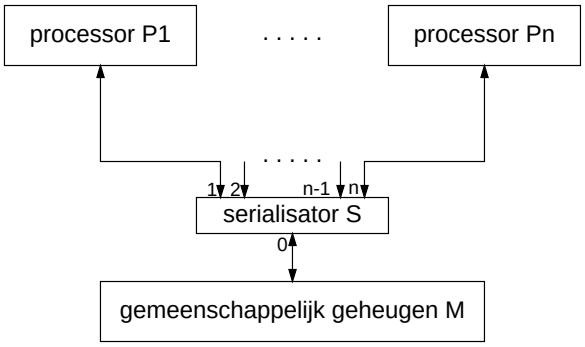
definities van geheugenmodellen leggen weinig eisen op aan de structuur van een implementatie – enkel de architectuur wordt vastgelegd.

### 3.11.1 Voorbeeld van een sequentieel consistent systeem

De implementatie van sequentiële consistentie zoals voorgesteld in figuur 3.24 bestaat uit $n$ processors $P_1 \ldots P_n$, een serialisator, een éénpoortsgeheugen $M$ en communicatiekanalen tussen deze eenheden. Via elk communicatiekanaal kunnen berichten verstuurd worden in twee richtingen, maar slechts in één richting tegelijk. Dit betekent dat voor elk communicatiekanaal de verstuurde en ontvangen berichten onderling totaal geordend zijn. In de beschouwde implementatie doet de processor met nummer $p$ niets anders dan de geordende lijst opdrachten $E_p = (Op_p, \xrightarrow{\text{po}\,p})$ één voor één uitvoeren, en past dus geen herordening toe. Er worden enkel gewone lees- en schrijfopdrachten beschouwd. De uitvoering door processor $p$ van de opdrachtenlijst $E_p$ bestaat erin telkens het bericht te versturen dat volgens tabel 3.21 overeenstemt met de opdracht, en dan te wachten op het volgens dezelfde tabel met de opdracht overeenstemmende antwoord.

Het protocol tussen de serialisatie-eenheid en het geheugen is hetzelfde als dat tussen processor en geheugen. De serialisatie-eenheid geeft de ontvangen opdrachten in volgorde door aan het geheugen, wacht op het antwoord van het geheugen en zendt dat antwoord dan naar de betreffende processor. Dit heeft als gevolg dat voor elk communicatiekanaal niet alleen een totale ordening bestaat voor de verstuurde berichten, maar ook voor de opdrachten waartoe deze berichten behoren. In het algemeen mag een implementatie van het geheugenmodel sequentiële consistentie aan de eigenschap voldoen dat de volgorde van opdrachten correspondeert met de volgorde van de berichten, maar dat hoeft niet.

In figuur 3.25 is een voorbeeld opgenomen van een mogelijk verloop van de gebeurtenissen corresponderend met de uitvoering uit fi-

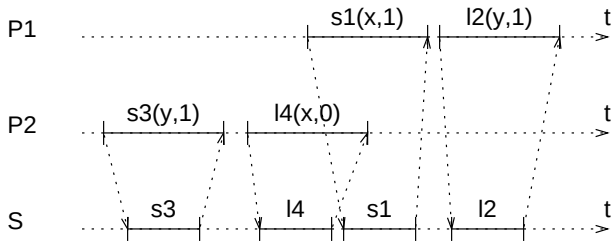



**Figuur 3.24:** Een mogelijke implementatie van sequentiële consistentie.

guur 3.6 en dat in de implementatie van een sequentieel consistent systeem zoals in figuur 3.24. Processors $P_1$ en $P_2$ voeren daarbij in volgorde de opdrachten $s_1$ en $l_2$ resp. $s_3$ en $l_4$ uit. De serialisator $S$ creëert op basis van de programmaordening de totale ordening $s_3$, $l_4$, $s_1$, $l_2$. Dit is de geheugenordening $\xrightarrow{\text{mo}}$.

Als we de ordeningen van opdrachten aan de ingangen van de serialisator aanduiden met $S_1$ t.e.m. $S_n$, en als we de ordening van de opdrachten aan de uitgang van de serialisator aanduiden met $S_0$, dan volgt uit het gedrag van de serialisator dat in het algemeen $S_0 = \text{linext}(S_1 \cup \ldots \cup S_n)$. De ordening waargenomen door het geheugen $M$, is per definitie de geheugenordening $\xrightarrow{\text{mo}}$. Daar de processors geen herordening toepassen geldt dat $S_p = \xrightarrow{\text{po}\,p}$. Omdat $S_0 = \xrightarrow{\text{mo}}$ en $S_p = \xrightarrow{\text{po}\,p}$, volgt er dat voor het systeem in figuur 3.24 geldt dat $\xrightarrow{\text{mo}} = \text{linext}(\xrightarrow{\text{po}\,1} \cup \ldots \cup \xrightarrow{\text{po}\,n})$. Dit bewijst dat het systeem voorgesteld in figuur 3.24 voldoet aan het geheugenmodel sequentiële consistentie. Merk op dat de notatie $R_2 = \text{linext}\, R_1$, met $R_1$ en $R_2$ partiële ordeningen, betekent dat de relatie $R_2$ één van de mogelijke lineaire extensies is van de relatie $R_1$. De geheugenordening $\xrightarrow{\text{mo}}$ is dus zoals verwacht niet eenduidig bepaald door de programmaordening $\xrightarrow{\text{po}}$.

Het is mogelijk voor de implementatie van een sequentieel consistent systeem geavanceerdere technieken zoals herordening of caches toe te passen – voor een bespreking hiervan verwijzen we naar de literatuur [AH90a, GMG91, GGH91a].

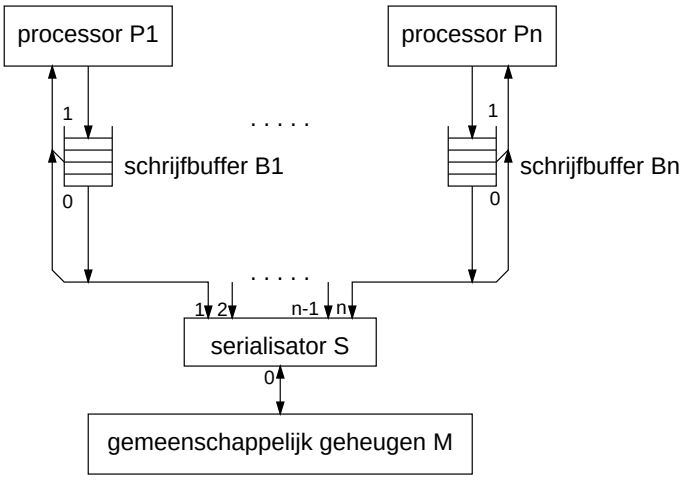



**Figuur 3.25:** Een mogelijke berichtensequentie voor de uitvoering uit figuur 3.6.

### 3.11.2 Voorbeeld van een TSO-systeem

In figuur 3.26 staat een voorbeeld van een implementatie van het TSO-geheugenmodel. Deze implementatie verschilt van de eerder beschreven implementatie van sequentiële consistentie door de aanwezigheid van een schrijfbuffer en doordat de processors herordening toepassen. De toegepaste herordening bestaat erin dat in een uitvoering een schrijfopdracht met een in de programmaordening onmiddellijk erop volgende leesopdracht voor een andere locatie mogen worden omgewisseld. De schrijfbuffers zijn FIFO-buffers voor schrijfopdrachten, waarbij voor leesopdrachten eerst wordt gecontroleerd of in de schrijfbuffer een schrijfopdracht zit voor dezelfde locatie. Indien dat het geval is dan wordt de schrijfopdracht niet naar het gemeenschappelijk geheugen $M$ doorgestuurd. De communicatie tussen processor en schrijfbuffer en tussen schrijfbuffer en serialisator verloopt volgens hetzelfde protocol als de communicatie tussen serialisator en gemeenschappelijk geheugen – zie ook tabel 3.21. Als de schrijfbuffer een schrijfopdracht van de processor kan bufferen wordt onmiddellijk aan de processor meegedeeld dat de volgende opdracht mag worden doorgestuurd. Als de schrijfbuffer volledig gevuld is wordt gewacht tot er weer ruimte in de buffer is vrijgekomen vooraleer gemeld wordt aan de processor dat de volgende opdracht mag worden gestuurd.

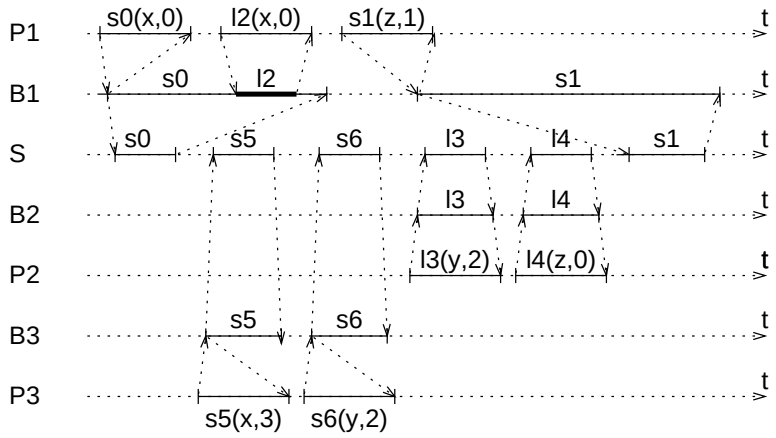
In figuur 3.27 staat een voorbeeld van de verwerking van de uitvoering uit figuur 3.11 op het TSO-systeem uit figuur 3.26. De in figuur 3.11 impliciet aanwezige initialisatie van $x$ met de waarde nul werd toegevoegd om de bufferwerking te kunnen illustreren. Uit figuur 3.27 leiden we af dat processor $P_1$ de opdrachten $s_1$ en $l_2$ heeft omgewisseld. Uit de figuur is verder af te leiden dat opdracht $l_2$ werd afgehandeld




**Figuur 3.26:** Een mogelijke implementatie van TSO.

door de buffer zelf en niet naar het geheugen werd doorgestuurd. Op basis van de beschrijving van het gedrag van de serialisator volgt dat de geheugenordening $S_0 = \xrightarrow{\text{mo}}$ gegeven wordt door volgende uitdrukking: $S_0 = \text{linext}(S_1 \cup \ldots \cup S_n)$. Hierin is $S_0$ de opdrachtenvolgorde waargenomen tussen serialisator en geheugen, en staat $S_p$ voor de opdrachtenvolgorde zoals waargenomen tussen processor $P_p$ en schrijfbuffer $B_p$. Als we met $B_{p,1}$ de volgorde van de opdrachten zoals waargenomen tussen processor $p$ en schrijfbuffer $p$ aanduiden, en met $B_{p,0}$ de volgorde van opdrachten zoals waargenomen tussen schrijfbuffer $p$ en poort $S_p$ van de serialisator, dan geldt dat $B_{p,0} \subset B_{p,1}$. Door de eigenschap dat een processor alleen onmiddellijk opeenvolgende lees- en schrijfopdrachten naar een verschillende locatie omwisselt, de eigenschap dat een schrijfbuffer alleen leesopdrachten lokaal afhandelt en dat $B_{p,0} \subset B_{p,1}$ en de eigenschap $S_0 = \text{linext}(S_1 \cup \ldots \cup S_n)$ te combineren kan worden bewezen dat dit systeem voldoet aan het geheugenmodel TSO.

In dit voorbeeld is de volledige geheugenordening $\xrightarrow{\text{mo}}$ nergens in het systeem waarneembaar, maar moeten meerdere ordeningen gecombineerd worden om de geheugenordening te reconstrueren. Dit geldt voor de meeste implementaties van gemeenschappelijk-geheugensystemen.




**Figuur 3.27:** Voorbeeld van de verwerking van de uitvoering uit figuur 3.11 op een TSO-geheugensysteem. De vermelde ordeningen zijn de ordeningen waargenomen aan de uitgang van de processors $P_p$, schrijfbuffers $B_p$ en serialisator $S$.

### 3.11.3 Andere organisatieaspecten

Via de voorgaande voorbeelden werd geïllustreerd hoe voor lees- en schrijfopdrachten een communicatieprotocol kan worden opgesteld en werd het gedrag van een serialisator en een schrijfbuffer besproken. Andere opdrachten, zoals lees/wijzig/schrijf-opdrachten, lock-, unlock- en grensopdrachten kunnen gemodelleerd worden via aanpassingen in het communicatieprotocol. De implementatie van etiketten hoort thuis in de processor. Caches kunnen worden toegevoegd als modules die communiceren met één processor, met het geheugen en ook onderling communiceren. Speculatieve uitvoering en herordening van opdrachten zijn toegelaten zolang door deze optimalisaties het geheugenmodel niet veranderen.

## 3.12 Verwant werk

Er werden reeds eerder formalismen ontwikkeld om geheugenmodellen te beschrijven. In de literatuur wordt met volgende benamingen naar het begrip geheugenmodel verwezen: *memory model*, *consistency model*, *memory consistency model*, *consistency condition* en *correctness condition*. De reeds gepubliceerde formalismen verschillen onderling op

vlakken als het soort geheugenmodellen dat voorgesteld kan worden, of ze geschikt zijn voor een programmeur of voor een hardwareontwerper, of er rekening gehouden wordt met etiketten, of er synchronisatieopdrachten beschouwd worden en zo ja welke. Hieronder wordt een overzicht gegeven van de bestaande benaderingen, onderverdeeld in programmagerichte benaderingen, programmeurgerichte benaderingen, hardwaregerichte benaderingen, en benaderingen waarbij geheugenmodellen gecombineerd worden met berichtencommunicatie. Onder deze benamingen wordt resp. verstaan dat geheugenmodellen geformuleerd worden als gedrag dat gegarandeerd wordt voor een bepaalde klasse van programma's, als gedrag dat gegarandeerd wordt voor om het even welk programma en in functie van het beschouwde programma, als richtlijnen waarmee een processor- en/of cacheontwerper direct aan de slag kan of als een combinatie van gemeenschappelijkgeheugen opdrachten en opdrachten voor berichtendoorgave.

Een beknopt overzicht van de vele bestaande geheugenmodellen kan ook teruggevonden worden in de overzichtsteksten door Mosberger, Adve en Iftode [Mos93, AG96, APR99, IS99]. Terwijl Mosberger en Adve de nadruk leggen op geheugenmodellen voor multiprocessors, beschrijft Iftode uitvoerig de geheugenmodellen toegepast in bestaande DSM-systemen. Iftode's beschrijving per geheugenmodel bevat zowel het programmeermodel als de implementatietechnieken die in het DSM-systeem werden toegepast om goede prestaties te bereiken.

Uit onderstaand overzicht zal blijken dat één van de bijdragen van dit werk erin bestaat naast de lees- en schrijfopdrachten ook barrièresynchronisatie over meerdere processen en semafooroperaties in hetzelfde raamwerk te behandelen.

In dit hoofdstuk worden enkel programma's beschouwd waarvan de programmacode niet verandert tijdens de uitvoering van het programma. De lees- en schrijfopdrachten verwijzen dus naar data en niet naar de code van het programma zelf.

### 3.12.1 Programmagerichte benaderingen

De programmagerichte benadering onderscheidt zich van andere formuleringen van geheugenmodellen door te vertrekken van een bepaald type programma's, en alleen voor die programma's de consistentie van het geheugensysteem te garanderen. De verzameling programma's waarvoor zo'n geheugenmodel geldt wordt meestal ge-

formuleerd als een aantal eigenschappen van geheugenordeningen die vervuld moeten zijn. Binnen deze klasse geheugenmodellen worden programma's beschouwd die bestaan uit lees- en schrijfopdrachten. Deze opdrachten zijn al of niet voorzien van een etiket dat mee bepaalt welke herordening de eigen processor op de opdracht mag toepassen. Onderstaande programmagerichte geheugenmodellen garanderen sequentiële consistentie (zie ook paragraaf 3.8.2) voor op gepaste wijze van etiketten voorziene programma's. Deze geheugenmodellen en de ondersteunde etiketten in die modellen zijn **DRF0** (*data-race-free-0*) met het etiket *synchronizing* [AH90b, AH98], **DRF1** (*data-race-free-1*) met de etiketten *paired synchronizing* en *unpaired synchronizing* [AH93], **PL** (*properly labeled*) met de etiketten *acquire*, *release*, *non-synchronization* en *non-competing* [GLL+90, GMG91] en **PLpc** (*programs with Properly Labeled competing accesses for systems that guarantee Processor Consistency among competing accesses*) met de etiketten *loop*, *competing* en *non-competing* [GAG+92, Adv93]. Voor de betekenis van deze etiketten en in welke omstandigheden deze moeten worden toegepast, zie de respectieve publicaties.

Singh [Sin95] omschrijft voor meerdere geheugenmodellen de verzameling programma's waarvan uitvoering op het beschouwde geheugenmodel hetzelfde resultaat oplevert als bij uitvoering op één sequentieel consistent systeem. Singh maakt in zijn formulering gebruik van een globale toestand over alle processors heen en ook van één enkele geheugenordening. Dit heeft als gevolg dat uitbreidingen van het formalisme nodig zijn om het volledige gedrag van geheugenmodellen zoals processorconsistentie (PC) of release-consistentie (RC) te kunnen beschrijven. Verder beschouwt Singh geen locks of barriers.

### 3.12.2 Programmeurgerichte benaderingen

De programmeurgerichte benadering bestaat erin het geheugenmodel zo te formuleren dat het gedrag van een programma in min of meerdere mate eenvoudig door een programmeur uit de definitie van het geheugenmodel kan worden afgeleid. Er zijn hierin twee benaderingswijzen mogelijk: ofwel wordt het geheugenmodel gedefinieerd in termen van het programma dat moet worden uitgevoerd (inclusief lussen en herhalingsopdrachten), ofwel in termen van de programma-uitvoering (een lineaire sequentie opdrachten per proces).

Het hieronder geciteerde werk van Linder en Raynal en het werk in

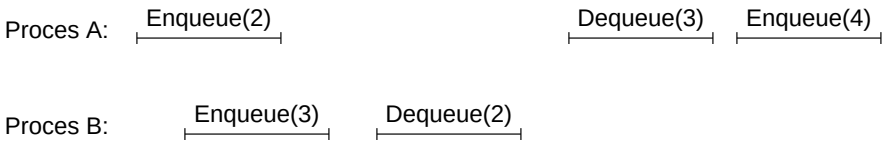
dit hoofdstuk vertrekt van de programma-uitvoering. Linder gebruikt *access graphs*, waarbij lees- en schrijfopdrachten knopen in een graaf zijn, om een geschiedenis van een programma-uitvoering voor te stellen [LH94]. Door te controleren of de graaf aan bepaalde voorwaarden voldoet kan worden nagegaan of de graaf een programma-uitvoering voorstelt die mogelijk is onder een bepaald geheugenmodel. Raynal daarentegen gaat uit van lees/schrijf-geschiedenissen (*histories*) over verschillende processors heen als de basis om geheugenmodellen te definiëren [RS97]. Een geheugenmodel wordt door Raynal gedefinieerd als een beperking op de toegelaten geheugenordeningen tussen lees- en schrijfopdrachten. Alhoewel beide benaderingen resulteren in een beknopte en overzichtelijke definitie per geheugenmodel, is zowel de aanpak van Linder als die van Raynal niet algemeen genoeg om geheugenmodellen als processorconsistentie te kunnen definiëren, en wordt er geen rekening gehouden met etiketten of synchronisatieopdrachten. Bovendien wordt geen onderscheid gemaakt tussen modelspecifieke en modelonafhankelijke eigenschappen.

Attiya vertrekt van het programma om de geheugenmodellen sequentiële consistentie (SC), zwakke ordening (WO) en hun eigen model hybride consistentie te definiëren. Deze aanpak heeft als voordeel dat naast data-afhankelijkheden ook beslissingsafhankelijkheden (*control dependencies*) binnen het model kunnen worden behandeld [ACFW98, AF98]. Er worden echter noch acquire- of release-etiketten noch synchronisatieopdrachten behandeld.

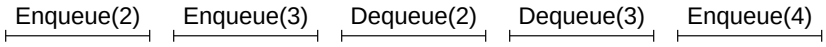
### 3.12.3 Hardwaregerichte benaderingen

Terwijl in de programmeurgerichte aanpak lees- en schrijfopdrachten als één geheel worden behandeld, wordt in een hardwaregerichte aanpak elke schrijfopdracht gesplitst in een opdracht per lokaal geheugen.

Herlihy splitst in een programma-uitvoering elke opdracht in twee gebeurtenissen, nl. de zgn. *invocation* en *response events*. Daarbij is een *invocation event* de doorgave van een opdracht van een proces naar het geheugen, en is het *response event* het antwoord dat van het geheugen naar het proces teruggestuurd wordt. Herlihy ordent deze gebeurtenissen partieel op basis van de tijdstippen waarop ze uitgewisseld worden tussen een proces en het geheugen. Op basis van de tijdstippen waarop de gebeurtenissen worden uitgevoerd wordt dan de volgorde van opdrachten vastgelegd, eveneens een partiële ordening. Op basis van




(a) Geschiedenis $H$.




(b) Linearisatie van geschiedenis $H$.

**Figuur 3.28:** Voorbeeld van een lineariseerbare geschiedenis van opdrachten uitgevoerd op een FIFO-queue: de geschiedenis zelf (a) en de linearisatie van die geschiedenis (b).

deze opdrachten wordt lineariseerbaarheid (*linearizability*) van een geschiedenis (*history*) op een dataobject gedefinieerd. Een geschiedenis is lineariseerbaar als en slechts als er voor de opdrachten uit de geschiedenis een totale ordening bestaat zodanig dat de uitvoering van de opdrachten in die totale ordening hetzelfde resultaat heeft als de oorspronkelijke geschiedenis, en bovendien de volgorde van de gebeurtenissen per proces behouden blijft [HW90]. De term object heeft hierbij dezelfde betekenis als in de context van objectgeoriënteerde programmeertalen, en er wordt ondersteld dat elk object opgeslagen is in het gemeenschappelijk geheugen. De bewerkingen op een object zijn dus niet beperkt tot lees- en schrijfopdrachten. In tegenstelling tot bij andere auteurs gaat het hier dus om een consistentievoorwaarde per object, en niet over een consistentievoorwaarde over meerdere objecten. Zie ook het voorbeeld in figuur 3.28.

Collier volgt ook de hardwaregerichte aanpak en splitst elke schrijfopdracht in evenveel deelopdrachten als er processors zijn. Een geheugenmodel wordt door Collier gedefinieerd als een aantal architectuurregels, waarbij elke architectuurregel een aspect vastlegt van de manier waarop een programma mag worden uitgevoerd door een architectuur [Col92]. Daar de individuele regels op een vrij laag niveau gedefinieerd zijn, kan het aantal regels nodig voor een architectuurbeschrijving vrij omvangrijk zijn. Collier beschouwt alleen lees- en schrijfopdrachten, en dus geen etiketten of synchronisatieopdrachten.

Sindhu ontwierp een formalisme om de geheugenmodellen TSO, PSO en RMO uit de SPARC-architectuur te kunnen beschrijven [SFC92]. Daarbij wordt uitgegaan van een architectuur die bestaat uit één enkel gemeenschappelijk geheugen, eventueel een schrijfbuffer per processor en een lokaal geheugen per processor. Het gemeenschappelijk geheugen wordt ondersteld een één-poortsgeheugen te zijn, zodat alle opdrachten die langs die poort passeren totaal geordend zijn. Er worden twee partiële ordeningen beschouwd, die aangeduid worden met het symbool ; voor de programmaordening en $\leq$ voor de ordening waargenomen aan de poort van het gemeenschappelijk geheugen. Terwijl in het formalisme van Sindhu slechts twee relaties nodig zijn om de werking van een geheugensysteem te beschrijven, heeft dit formalisme als nadeel dat het niet geschikt is om een gedistribueerd systeem te beschrijven.

Gibbons formalisme voor het beschrijven van gemeenschappelijk-geheugensystemen is vooral gericht op de beschrijving van de interactie tussen processor en geheugensysteem [GM92]. Elke lees- en schrijfopdracht wordt onderverdeeld in drie deelopdrachten: de versturing van de opdracht van processor naar geheugen, de verwerking van de opdracht door het geheugensysteem, en de melding van het geheugen naar de processor dat de opdracht voltooid is. Gibbons geeft deze deelopdrachten de namen *ReadRequest, MemoryRead, ReadReturn* en *WriteRequest, MemoryWrite* en *WriteReturn* voor resp. lees- en schrijfopdrachten. Dit formalisme laat toe het onderscheid te modelleren tussen geheugensystemen die opdrachten al of niet in volgorde afwerken, maar is te gedetailleerd om als algemene beschrijvingsmethode voor geheugenmodellen te dienen. Dezelfde opsplitsing van opdrachten werd door Afek [ABM93] toegepast om een implementatie van zwakke ordening te beschrijven.

Gharachorloo beschouwt lees-, schrijf- en lees/wijzig/schrijf-opdrachten met etiketten en ook grenssynchronisatie op een processor. In zijn benadering wordt elke schrijfopdracht in $n + 1$ deelopdrachten gesplitst: een lokale opdracht $s_{init}$ en $n$ schrijfopdrachten $(s_1, \ldots, s_n)$. De opdracht $s_{init}$ is alleen zichtbaar voor de processor die de schrijfopdracht lanceerde, terwijl de schrijfopdrachten $(s_1, \ldots, s_n)$ de schrijfopdracht ook voor de andere processors zichtbaar maken. Gharachorloo stelde naast zijn formalisme ook naar de architectuurontwerper geoptimaliseerde definities op voor een breed gamma aan geheugenmodellen. Zoals bij Collier is ook bij Gharachorloo de beschrijving van een geheugenmodel behoorlijk complex [GAG$^+$93, Gha95].

Lamport stelt in [Lam97] een algemeen formalisme voor om niet-atomische opdrachten te behandelen, zonder dat daarbij vastgelegd is uit welke gebeurtenissen de behandelde opdrachten moeten bestaan. De operatoren $\rightarrow$ en $\dashrightarrow$ stellen daarbij de relaties *gaat vooraf aan* resp. *kan beïnvloeden* voor. Dit betekent dat elke gebeurtenis resp. een gebeurtenis uit de eerste opdracht aan de tweede opdracht voorafgaat. Samen met de operatoren worden axioma's ingevoerd voor deze operatoren. Lamport past dit formalisme toe om de correctheid van de implementatie van een synchronisatieopdracht te bewijzen. Er wordt in de publicatie [Lam97] niet ingegaan op het onderwerp van de specificatie van geheugenmodellen.

### 3.12.4 Verificatie van implementaties van geheugenmodellen

Een belangrijke moeilijkheid bij de verificatie van cacheconsistentieprotocollen geïmplementeerd als een eindige automaat is dat bij verificatie de afmetingen van de toestandsruimte zeer sterk toenemen in functie van het aantal processors en de complexiteit van het consistentieprotocol. Daarom werd o.m. door Pong onderzoek verricht naar methodes om het aantal toestanden te verkleinen [PD95, PD98].

Condon introduceerde twee belangrijke ideeën voor hardwaregerichte geheugenmodellen: i.p.v. een schrijfopdracht in evenveel deelopdrachten te splitsen als er processors zijn, splitst zij een schrijfopdracht in een lokale en globale schrijfopdracht. Verder worden Lamportklokken gebruikt om op een efficiënte wijze de partiële ordeningen uit een geheugenmodel via de hardware te kunnen opleggen tijdens programma-uitvoering. Dit zijn twee belangrijke bijdragen om de verificatie van cachecoherentieprotocollen te vereenvoudigen [CHPS99]. Hierbij worden geen etiketten of interprocessorsynchronisatieopdrachten beschouwd.

### 3.12.5 Combinatie met berichtencommunicatie

Met als doel het consistentiebeheer van caches drastisch te vereenvoudigen, verandert Shen de semantiek van lees- en schrijfopdrachten zodat deze alleen nog effect hebben op het lokale geheugen van een processor. Naast lees- en schrijfopdrachten worden ook grensopdrachten voor één processor (*fences*) beschouwd, en twee opdrachten om de consistentie van lokale geheugens te beheren: *commit* en *reconcile*. Deze laatste opdrachten maken een waarde op een lokatie globaal zichtbaar

resp. halen de meest recente globale waarde op [SAR99]. Terwijl deze aanpak conceptueel eenvoudig is en ook relatief eenvoudig kan worden geïmplementeerd, wordt de verantwoordelijkheid voor het garanderen van de consistentie van het geheugensysteem naar de programmeur resp. compilerschrijver doorgeschoven. Het is zelfs zo dat een programma met een minimaal aantal *commit*- en *reconcile*-opdrachten de structuur heeft van een programma op basis van actieve berichten i.p.v. de structuur van een gemeenschappelijk-geheugenprogramma.

Byrd beschouwt in een overzichtspublicatie analoge technieken als die toegepast door Shen, maar dan toegepast op DSM-systemen. Byrd beschouwt meerdere vormen van producentgedreven communicatie en ook communicatie op initiatief van de consument (*prefetch*). Er werden goede resultaten behaald voor simulaties van uitvoeringen op een hardware DSM-systeem [BF99].

## 3.13 Besluit

Het voorgestelde formalisme laat toe om geheugenmodellen en hun eigenschappen nauwkeurig te beschrijven. Geheugenmodellen zijn fundamenteel voor parallelwerking op basis van gemeenschappelijk geheugen. Het formalisme omvat naast lees- en schrijfopdrachten ook atomische lees/wijzig/schrijf-opdrachten, vier soorten etiketten die controle over herordening van opdrachten binnen een proces mogelijk maken, en synchronisatieopdrachten. Het beschrijven van geheugenmodellen in dit formalisme laat toe geheugenmodellen onderling eenvoudiger te vergelijken, en de voorgestelde aanpak is bovendien geschikt voor automatische bewijsvoering. Ook werd er onderscheid gemaakt tussen modelspecifieke en modelonafhankelijke eigenschappen, wat de eenvoud van een specificatie ten goede komt. Ook werd de geschiktheid aangetoond van het formalisme om berichtencommunicatiemodellen te definiëren.

# Hoofdstuk 4

# Prestatie van berichtencommunicatie met multicast

## 4.1 Inleiding

Voor het uitvoeren van parallel rekenwerk bestaat er een ruime keuze uit hardwarefamilies om deze berekeningen op uit te voeren. De twee meest gebruikte families zijn de multiprocessor en het netwerk van werkstations (*network of workstations* of NOW). Terwijl multiprocessors gespecialiseerde systemen zijn die in relatief kleine aantallen worden gefabriceerd, wordt een NOW opgebouwd uit in veel hogere oplagen geproduceerde werkstations en netwerkhardware. Daardoor heeft een NOW een gunstiger kostprijs dan een vergelijkbare multiprocessor. Bovendien is een NOW eenvoudiger uit te breiden. Ook kan er binnen een NOW gebruik worden gemaakt van de meest recente ontwikkelingen in uniprocessortechnologie en netwerkhardware. Daartegenover staat dat de communicatie binnen een NOW trager verloopt dan binnen een multiprocessor. Daarom is het belangrijk binnen een NOW voor zo snel mogelijke communicatie te zorgen. Daartoe kunnen drie technieken toegepast worden: optimalisatie van de communicatiesoftware, toepassen van snellere netwerkhardware en het toepassen van multicast. Onder multicast wordt verstaan dat bij het versturen van identieke data naar meerdere hosts, deze data slechts eenmaal over het netwerk verstuurd wordt. In dit hoofdstuk wordt dieper ingegaan op

de toepassing van multicast voor gedistribueerd rekenen binnen een NOW, en dit voor een gegeven berichtencommunicatiemodel.

Na de situering van het probleem en een korte bespreking van de mogelijke andere benaderingen in paragraaf 4.2, worden in de volgende twee paragrafen, nl. 4.3 en 4.4, de begrippen multicast zelf en het inzetten van een netwerk van werkstations als parallelle computer behandeld. Daarna volgt in paragraaf 4.5 de gekozen werkwijze volgens welke PVM werd uitgebreid met multicast. Met deze uitgebreide PVM-software werden testprogramma's uitgevoerd, voorgesteld in 4.6, waarvan de uitvoeringstijden werden gemeten en geanalyseerd. Deze resultaten worden besproken in 4.7. Als slot van dit hoofdstuk volgen een overzicht van het verwante werk in paragraaf 4.8, en het besluit van het in dit hoofdstuk voorgestelde onderzoek in paragraaf 4.9.

## 4.2 Probleemstelling

Om parallel rekenwerk uit te voeren op een netwerk van werkstations moet er eerst en vooral een netwerk aanwezig zijn tussen de samenwerkende werkstations. Onder de benaming *netwerk* wordt het geheel van bekabeling en software verstaan dat de werkstations onderling verbindt en dat het mogelijk maakt te communiceren. Meer algemeen wordt elk toestel dat berichten kan versturen en ontvangen via het beschouwde netwerk een **host** genaamd. De afspraken over de volgorde waarin berichten mogen worden verstuurd heet het berichtencommunicatieprotocol.

Twee veelvoorkomende communicatiepatronen bij parallel rekenwerk zijn de versturing van een bericht naar één andere host, en de versturing van een bericht naar alle andere hosts waarmee wordt samengewerkt. Stel dat elke host één netwerkinterface heeft, en dat er alleen gebruik wordt gemaakt van berichten met één bestemmeling. Dan duurt de verzending van een bericht naar meerdere hosts niet alleen langer dan de verzending naar een andere host, maar neemt bovendien de hoeveelheid communicatie toe naarmate er meer hosts meerekenen. Dit is nadelig voor de uitvoeringstijd.

Dit probleem kan verlicht worden door berichten met meerdere bestemmelingen te versturen in multicast. Dat betekent dat het bericht één maal over het netwerk wordt verstuurd, en dat alle samenwerkende hosts dat bericht zullen ontvangen. Ook al vermindert het communicatievolume door het toepassen van multicast, toch blijft er nog

steeds de communicatie in unicast. Deze unicastcommunicatie neemt toe samen met het aantal hosts. Met de term unicast wordt hier de communicatie aangeduid die door een host naar één enkele andere host wordt verstuurd.

Het probleem van de hoeveelheid communicatie bij gedistribueerd rekenwerk wordt reeds geruime tijd onderzocht. Andere oplossingen die reeds succesvol werden toegepast zijn het optimaliseren van de laag met de communicatiesoftware, het aanpassen van het besturingssysteem voor snellere communicatie en het toepassen van snellere netwerkhardware. Deze benaderingen worden meer in detail behandeld in paragraaf 4.8.

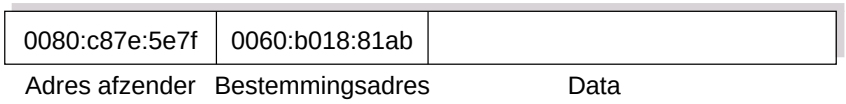
## 4.3 Multicast

Multicast is een techniek die het mogelijk maakt dezelfde data vanuit één host tezelfdertijd naar meerdere andere hosts te versturen. Deze techniek heeft zowel toepassingen op het domein van multimedia als van *distributed computing*, waarbij in dit hoofdstuk het tweede aspect zal worden besproken.

Bij gebruik van multicast dient gespecificeerd te worden welke de bestemmelingen zijn voor de verstuurde multicastdata. Een groep bestemmelingen krijgt de naam multicastgroep. Elke multicastgroep heeft een adres. Multicastdata wordt steeds verzonden naar één enkele multicastgroep. Alle hosts in een multicastgroep ontvangen de data bestemd voor die groep. Een host beslist zelfstandig van welke multicastgroepen hij deel of geen deel wil uitmaken. Op deze wijze zijn verzending van multicastdata en het beheer van een multicastgroep van elkaar losgekoppeld.

Multicast kan eenvoudig geïmplementeerd worden op een lokaal netwerk dat gebaseerd is op Ethernet, omdat data verzonden over een lokaal Ethernet-netwerk zichtbaar zijn voor alle andere hosts in dat lokale netwerk. Voor andere netwerktypes, b.v. ATM, komt er meer kijken bij de implementatie van multicast.

Data verzenden in multicast kan niet alleen op lokale netwerken, maar ook over het Internet. Daarbij dient dan voorzien te worden in een equivalent van het routingmechanisme voor de multicastpakketten.




**Figuur 4.1:** Structuur van een Ethernet-datagram.

### 4.3.1 Multicast in de fysische laag

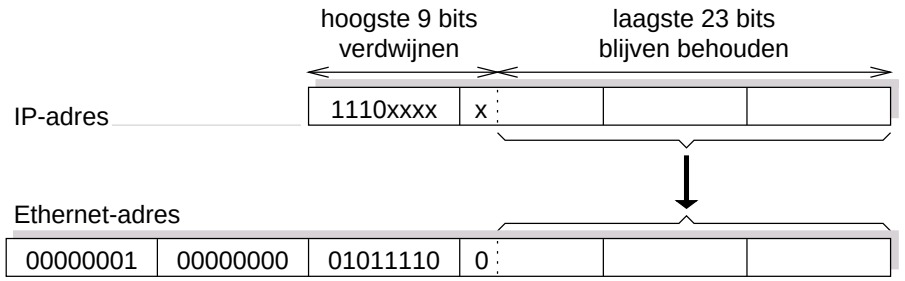
Voor de eenvoud beperken we ons in deze paragraaf tot één type lokaal netwerk, nl. Ethernet. De principes voor implementatie van multicast in de fysische laag van andere lokale netwerken zijn niet fundamenteel verschillend.

Op een Ethernet-netwerk worden data verstuurd per frame. Elk frame bestaat uit het adres van de zender, het adres van de bestemmeling en data – zie ook figuur 4.1. Een Ethernet-adres bestaat uit 48 bits, waarbij elke Ethernetkaart ter wereld een uniek adres heeft. De verstuurde data kunnen b.v. een IP-datagram zijn, maar kunnen ook tot een ander protocol behoren.

Elk Ethernet-pakket wordt door de verzendende netwerkkaart op dezelfde wijze verstuurd. Het bestemmingsadres bepaalt mee de verzendingswijze: als unicast-, multicast- of als broadcastdatagram. Bij unicastverzending is het bestemmingsadres het adres van de ontvangende netwerkkaart. Bij versturing in multicast ligt het bestemmingsadres in het bereik 0100:5e00:0000 t.e.m. 0100:5e07:ffff en stelt het bestemmingsadres een multicastgroep voor. De hosts in hetzelfde lokale netwerk die deel uitmaken van de betreffende multicastgroep zullen het datagram ontvangen.

### 4.3.2 Multicast in de netwerklaag

Op het niveau van de netwerklaag wordt voor de adressering binnen een IP-netwerk gewerkt met IP-nummers. Een IP-nummer in versie vier van het IP-protocol, IPv4, bestaat uit 32 bits. IP-nummers worden onderverdeeld in een aantal klassen. Deze klassen worden aangeduid met de letters A, B, C, D en E. Een adres wordt in een van deze klassen onderverdeeld naargelang het begint met de bitreeks 0, 10, 110, 1110 resp. 1111. Er wordt hierbij ondersteld dat IP-adressen genoteerd worden met de meest significante bit links. Elk IP-adres uit de klassen A, B of C duidt een unieke IP-host aan. Klasse-D adressen daarentegen stel-




**Figuur 4.2:** Afbeelding van een IP-multicastadres naar een Ethernet-multicastadres.

len een multicastgroep voor, en klasse-E adressen zijn voorbehouden voor toekomstig gebruik. Klasse-D adressen liggen dus in het bereik e000:0000 t.e.m. efff:ffff. Binnen het IP-protocol is een multicastgroep een verzameling van nul of meer hosts. Om binnen een lokaal netwerk lid te worden van een groep volstaat het voor een host aan zijn netwerk-interface opdracht te geven de pakketten voor die groep binnen te laten. Er is daarvoor geen communicatie met andere hosts nodig. Dit betekent ook dat de samenstelling van een multicastgroep en het adres van die multicastgroep van elkaar losgekoppeld zijn. Op een Ethernet-netwerk gebeurt de vertaling van IP-multicastadressen naar Ethernet-adressen door de laagste 23 bits van 0100:5e00:0000 te vervangen door de laagste 23 bits van het IP-adres – zie ook figuur 4.2. Er zijn dus meerdere IP-multicastadressen die corresponderen met hetzelfde Ethernet-multicastadres.

De opvolger van het IPv4-protocol is het IPv6-protocol. De belangrijkste verschillen voor multicast tussen IPv4 en IPv6 zijn de uitbreiding van 32 naar 128 bits voor IP-adressen, de uitbreiding van het veld voor de groepsaanduiding binnen een adres van 28 naar 112 bits, de aanduiding in het adres zelf of het gaat over een al of niet permanent toegekend adres en de toevoeging van een *scope*-veld voor routing in multicastadressen. Het *scope*-veld bepaalt het deel van het Internet waarin een multicastbericht verspreid wordt, en kan een van de volgende waarden aannemen: *node-local*, *link-local*, *site-local*, *organization-local* of *global*. Multicastgroepen behouden hun semantiek in IPv6 [Pos81, DH98, HD98].

### 4.3.3 Multicastrouting

Het Internet bestaat uit een complex geheel van lokale netwerken die met elkaar verbonden zijn. Een dergelijke verbinding bestaat tussen exact twee lokale netwerken. Deze verbindingen worden gerealiseerd door apparaten opgenomen in het lokale netwerk, de zgn. routers. Een datagram dat vanuit een lokaal netwerk naar een ander lokaal netwerk moet worden verstuurd, wordt opgepikt door de lokale router. Deze beslist dan langs welke inter-netwerkverbinding het datagram verder gestuurd dient te worden naar het bestemmingsnetwerk.

De implementatie van multicast op een lokaal Ethernet-netwerk is gebaseerd op het principe de multicastdata voor alle aangesloten hosts gelijktijdig zichtbaar te maken. Het uitbreiden van dit principe naar het volledige Internet is uitgesloten wegens de overbelasting die dit zou meebrengen. Er is dus een mechanisme nodig om multicastpakketten enkel uit te wisselen tussen die lokale netwerken die hosts bevatten die deel uitmaken van dezelfde multicastgroep. Er zijn dus twee bijkomende voorzieningen nodig: er dient over lokale netwerken heen bijgehouden te worden welke hosts in een gegeven multicastgroep zitten, en er is ook een mechanisme nodig om multicastpakketten tussen lokale netwerken uit te wisselen. Voor eerste aspect, het bijhouden van de groepsinformatie, werd het IGMP (*Internet Group Management Protocol*) ontwikkeld [Dee89]. Het IGMP is een protocol tussen host en router in hetzelfde lokaal netwerk, en maakt het mogelijk dat de router kan bijhouden tot welke groep een host behoort. Het tweede aspect, uitwisselen van multicastpakketten tussen lokale netwerken, gebeurt op basis van het gekozen multicastroutingprotocol in de routers op het deel van het netwerk dat men gebruikt. Multicastroutingsprotocollen zijn meestal kortste-pad of minimum-kost algoritmen – voor meer informatie verwijzen we naar [Pau98, LC99]. Deze protocollen kunnen ofwel in de routers ofwel als daemon op een host geïmplementeerd worden. Het meest bekende voorbeeld van een multicastpakketten routende software is de MBone (*Multicast Backbone of the Internet*) [Eri94].

### 4.3.4 Multicast in de transportlaag

Multicast op IP-niveau is onbetrouwbaar en heeft geen kwaliteitsgaranties. Dit wil zeggen dat niet gegarandeerd wordt dat een verzonden pakket werkelijk bij alle leden van de multicastgroep zal aankomen, en ook dat er geen garanties zijn over de tijd die nodig zal zijn voor-

aleer een multicastpakket bij een bepaalde host aankomt. Indien een bepaald kwaliteitsniveau of betrouwbare transmissie vereist is, dan is er nog een protocol nodig bovenop IP-multicast. Een dergelijk protocol heet een multicasttransportprotocol. Naast betrouwbaarheid en kwaliteitsgaranties zijn er nog andere factoren die een rol spelen bij multicasttransportprotocollen. Deze zijn de schaalbaarheid van het protocol, het meer geschikt zijn voor LAN resp. WAN, de wijze waarop het uitvallen van een host wordt opgevangen en de optimalisatie naar een gegeven waarschijnlijkheid op pakketfouten op het onderliggend netwerk. Door de vele criteria waarmee rekening dient te worden gehouden bij het ontwerp bestaat er een grote diversiteit aan multicastprotocollen. Voorbeelden hiervan zijn de door de IETF (*Internet Engineering Task Force*) gestandaardiseerde protocollen RMTP (*Reliable Multicast Transport Protocol*) [Dee89, PSLB97] en recenter PGM (heette vroeger *Pretty Good Multicasting*, en nu *Pragmatic General Multicast*) [SBE+99]. Voor een overzicht van de bestaande protocollen verwijzen we naar [LG98, Pau98].

Voor de implementatie en studie van multicast beperken we ons tot lokale netwerken met een lage kans op pakketfouten. Bovendien moet voor onze doeleinden een multicasttransportprotocol betrouwbaar zijn. Vandaar dat gekozen werd voor RMP (*Reliable Multicast Protocol*) [WMK95]. Doordat de werking van RMP gebaseerd is op een rondgaand token genereert het erg weinig protocoloverlast binnen een LAN, en presteert het daardoor beter dan vele andere multicasttransportprotocollen. Tot 1996 was de broncode van RMP vrij beschikbaar. Ondertussen behoren de intellectuele rechten tot het bedrijf GlobalCast Communications Inc.

### 4.3.5 Het RMP multicasttransportprotocol

Het RMP-protocol is een protocol dat functioneert op het niveau van de gebruikersapplicatie: elke applicatie die het RMP-protocol wenst te gebruiken kan daarvoor de RMP-bibliotheek met de applicatie meelinken.

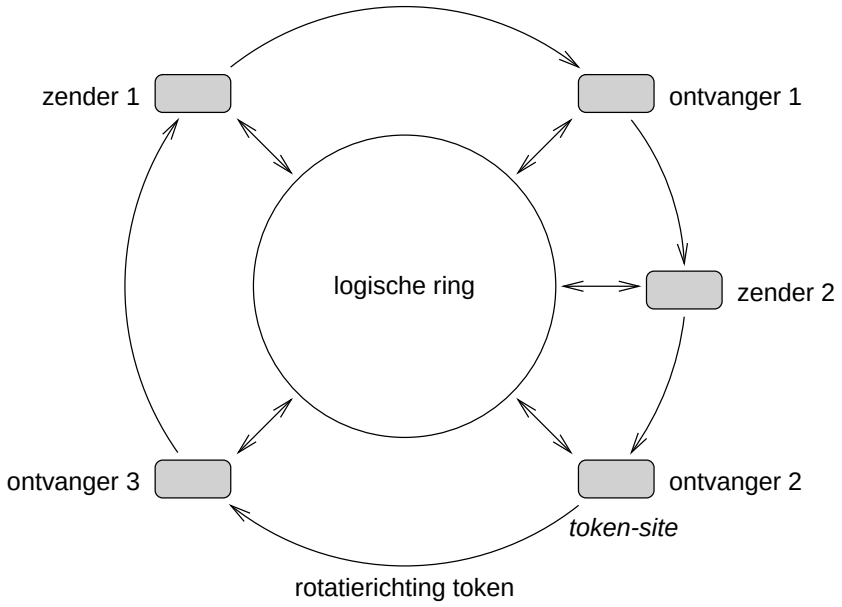
Het RMP-protocol maakt het mogelijk data in multicast naar een groep hosts te versturen, multicastdata te ontvangen, en dat een host zichzelf toevoegt aan of verwijdert uit een RMP-groep. Elke RMP-groep heeft een tekstuele naam, en er is een eenduidig verband tussen RMP-groepen en IP-multicastgroepen.

Intern ordent RMP alle hosts uit een RMP-groep in een ring. In deze ring circuleert een token, dat door elke host aan zijn opvolger doorgegeven wordt – zie ook figuur 4.3. Het token bevat ook een aanduiding van hoeveel pakketten door de voorgaande hosts werden verstuurd. De host die het token ontvangt zal nagaan of alle datagrammen sinds het vorige bezit van het token correct ontvangen werden, en zal ontvangst bevestigen van al deze datagrammen door in multicast een ACK te versturen naar alle hosts van de groep. Als een host daarentegen het verlies van een pakket detecteert, zal die een NACK multicasten, waarop een andere host dit pakket opnieuw zal versturen. Bovendien kan in de meeste gevallen het token samen met een datapakket verstuurd worden. Op deze wijze wordt in het RMP-protocol de hoeveelheid te versturen controle-informatie sterk beperkt.

RMP biedt volgende modes aan voor de verzending van data naar een groep met $N$ hosts:

- Onbetrouwbare verzending, vergelijkbaar met UDP.
- Betrouwbare verzending zonder garantie van de volgorde van aflevering.
- Afzendergeordende of *source-ordered* verzending, waarbij datagrammen worden afgeleverd in dezelfde volgorde als deze waarin ze verzonden werden.
- Totaal geordende aflevering, waarbij alle pakketten op elke bestemming in dezelfde volgorde worden afgeleverd.
- *K-resilient* aflevering, waarbij een pakket als afgeleverd wordt beschouwd als $K$ ontvangers het pakket correct hebben ontvangen.
- meerderheids-*resilient*, zie *K-resilient* met $K = \lceil (N+1)/2 \rceil$
- totaal *resilient*, zie *K-resilient* met $K = N$.

Er is een verband tussen de verzendingswijzen in paragraaf 3.9.3 en bovenstaande verzendingswijzen in RMP. *Betrouwbare verzending* in RMP garandeert dat elk verzonden bericht ooit aankomt, zonder dat de volgorde van aankomst gespecificeerd wordt. Dit stemt overeen met *asynchrone berichtenverzending* (AM). *Afzendergeordende verzending* betekent dat indien twee berichten worden verstuurd door een zelfde proces, en ook worden ontvangen door een zelfde proces, dat deze berichten dan gelijk geordend zijn. Dit is het *FIFO-communicatiemodel*




**Figuur 4.3:** Een logische ring voor een RMP-groep bestaande uit vijf hosts.

(FM). *Totaal geordende* aflevering komt erop neer dat alle verzonden berichten bij alle processen in dezelfde totale ordening worden ontvangen. Dit communicatiemodel is ook gekend als *synchroon* (SM). De *onbetrouwbare verzending* uit RMP heeft geen equivalent bij de modellen in paragraaf 3.9.3.

## 4.4 Parallelwerking via berichtencommunicatie

Als een netwerk van werkstations wordt ingezet als parallelle computer, dan zijn er een aantal moeilijkheden waarmee elke applicatieprogrammeur te maken krijgt. Deze moeilijkheden zijn:

- Werkstations van verschillende fabrikanten kunnen verschillende binaire voorstellingen hanteren voor data, wat gegevensuitwisseling bemoeilijkt.

- Er is functionaliteit nodig om een proces op een ander werkstation te kunnen starten.

- Verschillende soorten netwerken vragen elk een specifieke aanpak om het netwerk maximaal te kunnen benutten.

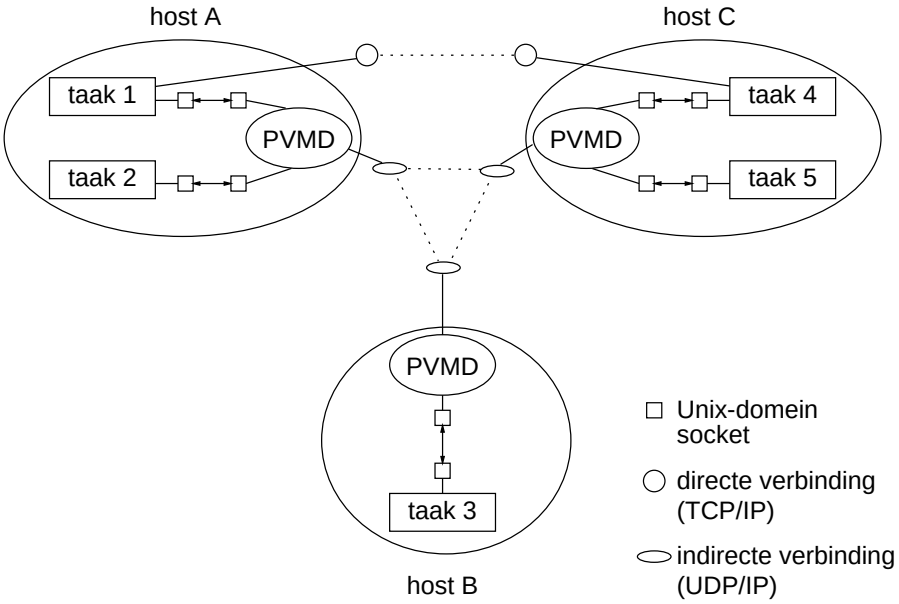
Om deze redenen maakt zo goed als elke NOW-applicatie gebruik van een extra softwarelaag die zorgt voor dataformaatconversie, het starten van taken op een ander werkstation en communicatieoptimalisatie volgens netwerktopologie. De meest bekende voorbeelden van dergelijke softwarelagen zijn PVM [SGDM94] en MPI [DOSW96]. Alhoewel PVM en MPI vergelijkbare functionaliteit aanbieden, verschillen zij hoofdzakelijk in de soorten platformen die zij ondersteunen. Alhoewel beide softwarelagen één-naar-meerdere communicatie ondersteunen in hun applicatie-interface, maakt voor zover mij bekend noch PVM noch MPI gebruik van multicast op netwerkniveau.

Er zijn structurele overeenkomsten tussen PVM als MPI: beide bestaan zowel uit een bibliotheek die moet meegelinkt worden met de NOW-applicatie en ook uit een daemon resp. server waarvan er juist één exemplaar actief moet zijn per werkstation waar taken van de NOW-applicatie lopen. Dit betekent dat de uitbreiding met multicast op netwerkniveau gelijkaardig is voor PVM en MPI. Besluiten i.v.m. de implementatiewijze en prestaties voor één van de twee berichtencommunicatiesystemen kunnen dus naar het andere berichtencommunicatiesysteem uitgebreid worden. Omwille van praktische redenen werd ervoor gekozen PVM uit te breiden met multicast op netwerkniveau.

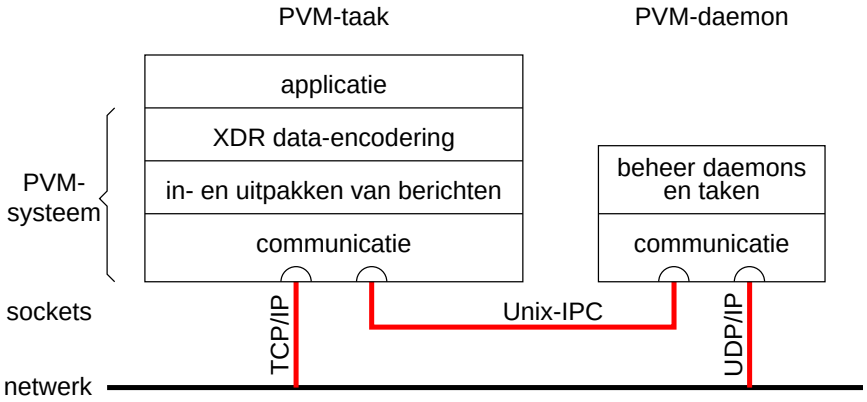
### 4.4.1 Structuur van PVM

PVM staat voor *Parallel Virtual Machine*, en is een softwaresysteem dat het voor applicaties mogelijk maakt een netwerk van werkstations als één enkele virtuele machine in te zetten. Een virtuele machine bestaat uit een aantal samenwerkende werkstations. Op elk werkstation loopt een PVM-daemon, en samen vormen deze daemons de ruggengraat van de virtuele machine. Naast de PVM-daemon lopen op elk werkstation nul of meerdere PVM-taken. Deze afzonderlijke taken vormen samen een applicatie van een gebruiker – zie ook figuur 4.4. Samenwerking van taken binnen deze virtuele machine gebeurt op basis van berichtencommunicatie.

PVM biedt functionaliteit aan voor het starten en stoppen van taken, datacommunicatie en datavertaling. Alle gegevens in PVM-berichten worden omgezet naar het XDR-formaat (*external data representation*), en bij ontvangst worden deze vertaald naar het formaat eigen




**Figuur 4.4:** Voorbeeld van netwerkverbindingen tussen PVM-taken en -daemons.




**Figuur 4.5:** Lagenstructuur van een PVM-taak en de PVM-daemon op dezelfde host.

voor de host. Indien verzender en ontvanger hetzelfde dataformaat gebruiken, zijn deze vertalingsstappen dus redundant. De functionaliteit door PVM toegevoegd aan een PVM-taak kan onderverdeeld worden in drie lagen, zie ook figuur 4.5:

- Omzetting tussen host-afhankelijke en host-onafhankelijke datavoorstelling, XDR.
- Inpakken en uitpakken van data tot berichten (*pack / unpack*).
- Versturen en ontvangen van data via de socket-interface.

De XDR-laag is tevens de interface van PVM met de taak van de PVM-gebruiker. Hieronder worden twee implementaties van multicast besproken. In beide gevallen werd multicast toegevoegd op het niveau van de communicatielaag. Op deze manier blijft de functionaliteit van datavertaling en het in- en uitpakken van berichten behouden.

## 4.5 Uitbreiding van PVM met multicast

Uit de documentatie van PVM blijkt dat PVM data steeds verstuurt op betrouwbare wijze en met garantie van de afzenderordening. Vandaar dat deze verzendingswijze ook voor RMP zal worden gebruikt.

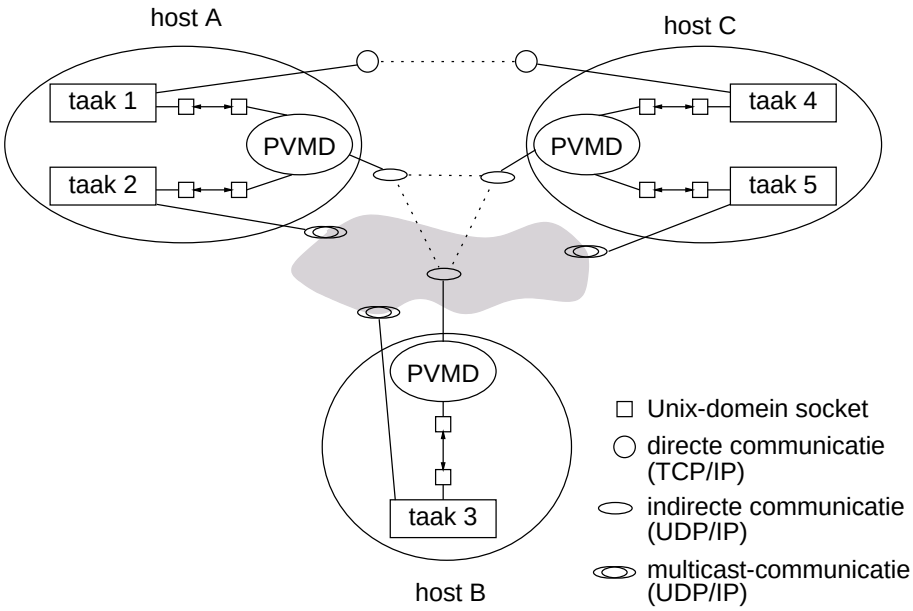
Data die vanuit een taak via PVM verstuurd worden, kunnen op twee wijzen verstuurd worden. De eerste mogelijkheid is van de taak naar de daemon op dezelfde host, dan van daemon naar daemon over het netwerk en tenslotte van daemon naar de bestemmingstaak. De tweede mogelijkheid is versturing direct van taak naar taak. De eerste mogelijkheid is standaard, terwijl de tweede mogelijkheid door de applicatie kan worden aangevraagd via de oproep pvm_setopt(PvmRouteDirect). Beide mogelijkheden zijn voorgesteld in figuur 4.4. Bij de eerste methode wordt het UDP-protocol gebruikt tussen de daemons met daar bovenop een eigen hertransmissieprotocol, terwijl bij de tweede methode gebruik wordt gemaakt van het TCP-protocol. Er werd proefondervindelijk aangetoond dat de tweede methode tot een factor twee sneller kan zijn dan de eerste methode. Daar staat tegenover dat voor de eerste methode een vast klein aantal sockets per taak volstaan, terwijl bij de tweede methode het aantal sockets per taak recht evenredig is met het aantal taken. De bestaansreden van de eerste methode is compatibiliteit met besturingssystemen die slechts in een klein aantal sockets per proces voorzien.

Terwijl voor de communicatie tussen de daemons de ontwerpers van PVM de keuze hadden tussen UDP en TCP, was voor rechtstreekse verbindingen tussen de taken het TCP-protocol de enige mogelijkheid. De reden hiervoor is dat gebruik van het UDP-protocol impliceert dat hertransmissie binnen strikte tijdslimieten hertransmissie moet kunnen worden gegarandeerd. Dit is mogelijk binnen de PVM-daemon, maar dit kan niet van een PVM-taak worden verwacht. Daar elk betrouwbaar multicastprotocol noodzakelijkerwijs gebaseerd is op communicatie met onbetrouwbare UDP-multicastpakketten, moet een betrouwbaar multicastprotocol dus voorzien in de mogelijkheid tot hertransmissie. Bij het implementeren van multicast op taakniveau zullen de PVM-taken dus op een of andere manier moeten meewerken om de hertransmissie correct te laten verlopen. Het alternatief voor implementatie van multicast op taakniveau is implementatie ervan in de PVM-daemon. Beide mogelijkheden worden hieronder meer in detail besproken.

### 4.5.1 Multicast in PVM-taken

Het hierboven besproken RMP-protocol eist dat de RMP-software minstens één maal per seconde geactiveerd wordt – dit is een conventie die beschreven wordt in de RMP-documentatie. Om RMP op taakniveau in te zetten zijn er twee alternatieven: ofwel ervoor zorgen dat in de PVM-taak RMP zo vaak als nodig geactiveerd wordt en enkel oproepen doen die voldoende kort zijn, ofwel het activeren van RMP parallel met de PVM-taak. Voorbeelden van acties die te lang duren om de goede werking van RMP te kunnen garanderen zijn: uitgebreide berekeningen, bepaalde systeemoproepen en het wachten op de aankomst van PVM-berichten. Er zijn twee mogelijkheden om RMP te activeren tijdens langdurige acties: ofwel een draad starten die enkel deze taak heeft, ofwel langdurige acties opsplitsen en activeringen inlassen. Er dient hierbij opgemerkt te worden dat draden nog niet op elk besturingssysteem beschikbaar zijn waarvoor PVM beschikbaar is. De tweede methode heeft als nadeel dat langdurende systeemoproepen uitgesloten worden. Het is aan de applicatieontwikkelaar om hierin een keuze te maken.

In het kader van dit doctoraat werd een softwarebibliotheek ontwikkeld bovenop PVM en RMP die beide integreert [VA97], en die de keuze open laat wat betreft de manier waarop RMP geactiveerd wordt. Deze bibliotheek vervangt volgende functies uit PVM: pvm_recv(),

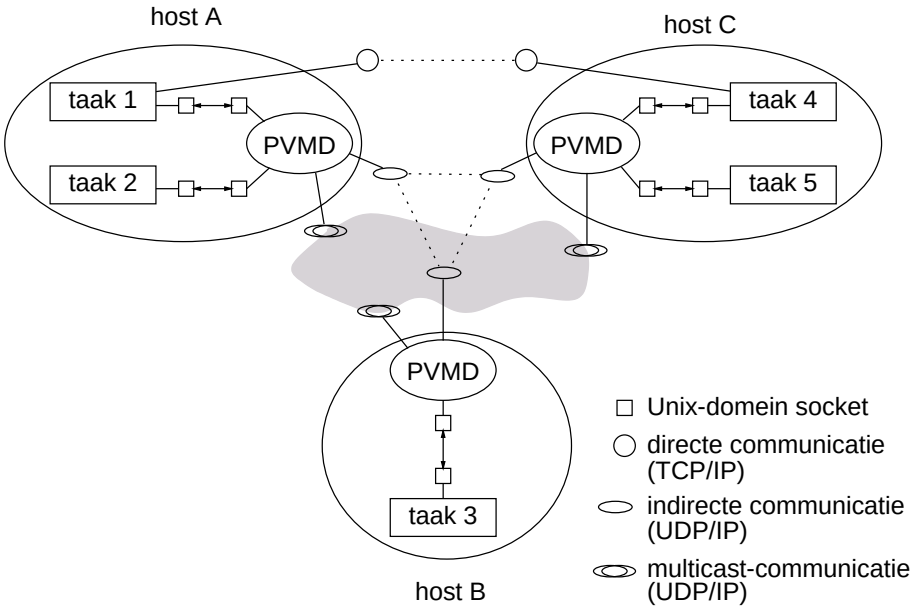



**Figuur 4.6:** Voorbeeld van netwerkverbindingen tussen PVM-taken en -daemons bij implementatie van multicast in de PVM-taken.

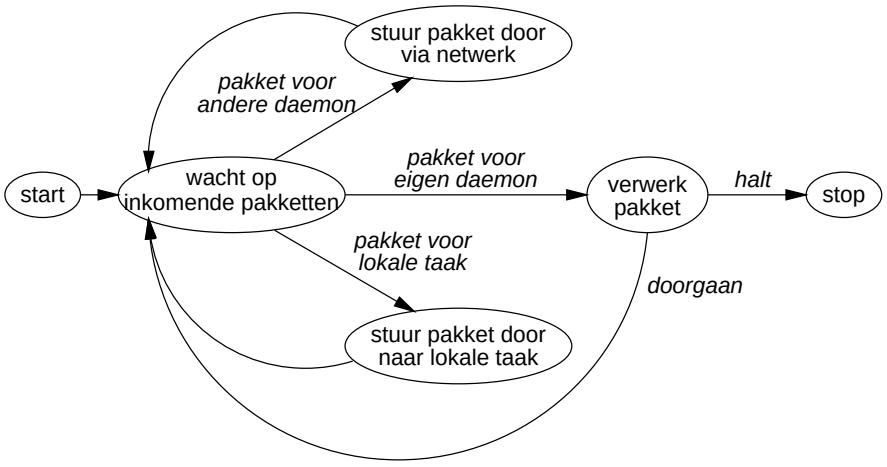
pvm_send(), pvm_exit() en uiteraard pvm_mcast(). Bovendien werden functies toegevoegd voor het verwerken van RMP-gebeurtenissen en voor initialisatie – de PVM-bibliotheek zelf heeft namelijk geen expliciete initialisatieoproep.

### 4.5.2 Multicast in PVM-daemon

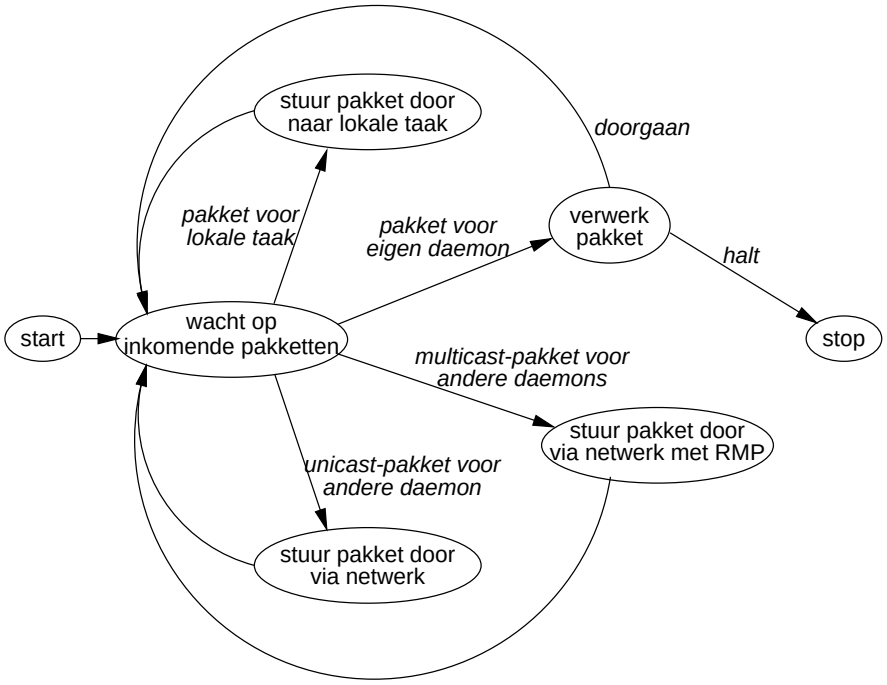
Intern is de PVM-daemon opgebouwd als een gebeurtenisverwerkend systeem: de daemon wacht op binnenkomende gebeurtenissen, waarna deze zo snel mogelijk verwerkt worden – zie ook figuur 4.8. De daemon verwerkt data in fragmenten van vaste grootte (*PvmFragSize*), zodat de tijd nodig om deze te versturen of te ontvangen beperkt blijft. De intervallen tussen twee opeenvolgende wachtopdrachten zijn dus kort. Het vervangen van de wachtopdracht door een wachtopdracht die eveneens de RMP-wachtrij verwerkt is dus voldoende voor de correcte werking van RMP binnen de PVM-daemon – zie ook fig. 4.9. Voor een gedetailleerde bespreking van de implementatie door lic. Kristof Lievens verwijzen we naar [Lie99].




**Figuur 4.7:** Voorbeeld van netwerkverbindingen tussen PVM-taken en -daemons bij implementatie van multicast in de PVM-daemon.




**Figuur 4.8:** Toestandsdiagram van de ongewijzigde PVM-daemon.




**Figuur 4.9:** Toestandsdiagram van de PVM-daemon uitgebreid met multicast.

| Test-programma | Rekentijd | Datahoeveelheid unicast | Datahoeveelheid multicast | Reductie met $p = 8$ |
|---|---|---|---|---|
| MV | $\mathcal{O}(n^3)$ | $(3(p-1)+1)n^2$ | $4n^2$ | 5.5 |
| GJ | $\mathcal{O}(n^3)$ | $(1.5(p-1)+0.5)n^2$ | $2n^2$ | 5.5 |

**Tabel 4.1:** Reductie hoeveelheid te verzenden data in multicast t.o.v. unicast. Hierbij staat $n$ voor de probleemdimensie en $p$ voor het aantal meerekenende processors.

## 4.6 Testprogramma's

Om de prestaties van de multicastimplementaties na te gaan gebruiken we twee programmakernen: de parallelle versies van een matrixvermenigvuldigingsprogramma (MV) en een programma voor het oplossen van een stelsel vergelijkingen met het Gauss-Jordan algoritme (GJ). Er worden programmakernen gebruikt i.p.v. een complexe applicatie omdat dit beter toelaat uit de resultaten af te leiden welke de invloedsfactoren op de resultaten zijn.

De geparallelliseerde versie van beide programmakernen heeft dezelfde structuur: telkens is er een hoofdtaak en één of meerdere neventaken. De hoofdtaak voert alle sequentiële code uit, en stuurt de neventaken. De parallelliseerbare gedeelten worden volledig door de neventaken uitgevoerd. Dit betekent dat de hoofdtaak eerst de initialisatie van de data uitvoert en deze dan doorstuurt naar de neventaken. De neventaken voeren de berekeningen uit en sturen tenslotte hun resultaten terug naar de hoofdtaak. De gemeten uitvoeringstijden voor zowel de sequentiële als de parallelle versies omvatten zowel de initialisatie, de eigenlijke berekeningen en de ontvangst van de resultaten.

Het MV-programma initialiseert twee matrices $A$ en $B$, beide $n \times n$, en vermenigvuldigt deze in een derde matrix $C$. Het GJ-programma start van een lineair stelsel met $n$ onafhankelijke vergelijkingen, waarvan in elke iteratiestap één vergelijking wordt geëlimineerd. De reken- en communicatietijden van het MV- en GJ-programma worden vermeld in tabel 4.1. Het is belangrijk daarbij te vermelden dat terwijl voor het MV-programma er enkel communicatie is aan het begin en het einde van het programma, er in het GJ-programma communicatie is tijdens elke iteratie van de buitenste lus.

In alle gevallen werden de XDR-datavertaling en de in- en uitpakfunctionaliteit van PVM gebruikt om berichten te versturen. Unicast-

| Naam configuratie | Type werkstation | Grootte L2-cache | Netwerksnelheid Mbit/s |
|---|---|---|---|
| A | HP 9000/712 | 0.25 MB | 10 |
| B | HP 9000/778 | 1.00 MB | 100 |

**Tabel 4.2:** Netwerkconfiguraties waarop de testprogramma's werden uitgevoerd.

data werden rechtstreeks tussen taken verstuurd door gebruik van de optie PvmRouteDirect. Data verstuurd met pvm_mcast() werden verstuurd in unicast tussen de taken, in multicast tussen de taken of in multicast tussen de daemons naargelang resp. de originele PVM-implementatie, de implementatie van multicast in de taken of de implementatie van multicast in de daemons werd gebruikt.

## 4.7 Resultaten

Het MV-programma en het GJ-programma werden uitgevoerd op twee verschillende testplatformen – zie tabel 4.2. In de werkstations was voldoende geheugen aanwezig om alle tussenresultaten bij te houden. Zowel de werkstations als het netwerk waren nagenoeg onbelast tijdens de metingen. De termen grote en kleine datablokken verwijzen naar hoeveelheden data groter resp. kleiner dan de grensgrootte vermeld in tabel 4.3. Als benadering voor de transfertijd $t$ van een blok $b$ bytes groot wordt volgend model gebruikt: $t = l + b/s$, met $l$ de latentie en $s$ de doorvoersnelheid van het netwerk.

**Eigenschappen netwerk** Uit de metingen in tabel 4.3 kunnen we enkele belangrijke eigenschappen afleiden over het netwerk en de gebruikte protocollen. De verschillende metingen gebeurden zowel op configuraties A en B, en zowel in unicast als in multicast. De multicastcijfers verwijzen naar verzending van 1 naar $p = 7$ andere hosts. Om de multicastcijfers en de unicastcijfers te vergelijken dienen de multicastcijfers nog met een factor $p = 7$ te worden vermenigvuldigd.

Uit de doorvoersnelheden gemeten zonder XDR kunnen we besluiten dat voor configuratie A zowel bij verzending in unicast (TCP) als bij verzending in multicast (RMP) de gehaalde snelheden dicht bij de maximaal haalbare snelheid van 10 Mbit/s / 8 = 1.25 MB/s voor deze

| | | Configuratie → | A | | B | |
|---|---|---|---|---|---|---|
| | $p$ | Verzendingswijze | u.c. | m.c. | u.c. | m.c. |
| zonder | 1 | Doorvoersnelheid, MB/s | 0.9 | 1.1 | 10.0 | 5.0 |
| XDR | 7 | Doorvoersnelheid, MB/s | 1.0 | 0.9 | 9.0 | 4.2 |
| | 1 | Doorvoersnelheid $r_1$, MB/s | 0.40 | 0.54 | 2.5 | 1.9 |
| met | 7 | Doorvoersnelheid $r_7$, MB/s | 0.54 | 0.48 | 6.6 | 1.8 |
| XDR | 7 | Latentie $l_7$, ms | 8.0 | 6.0 | 2.0 | 1.0 |
| | 7 | Grensgrootte $l_7 r_7$, KB | 4.3 | 2.9 | 13.2 | 1.8 |
| | 7 | Winstfactor m.c. $p \cdot r_{m,7}/r_{u,7}$ | 1.0 | 6.2 | 1.0 | 1.9 |

**Tabel 4.3:** Metingen transfersnelheid op de verschillende configuraties. Het symbool $p$ stelt het aantal bestemmelingen voor, en de afkortingen u.c. en m.c. staan voor unicast resp. multicast.

configuratie liggen. Dit geldt ook voor verzending met unicast onder configuratie B, maar niet voor de verzending met multicast onder deze configuratie. Er blijkt dat het RMP-protocol beter presteert voor het tragere netwerk uit configuratie A dan het snellere netwerk uit configuratie B. Door de cijfers voor de multicastsnelheden bij $p = 1$ en $p = 7$ in tabel 4.3 te vergelijken volgt dat de multicastsnelheden licht afnemen naarmate het aantal bestemmelingen stijgt. Dit is normaal: hoe meer bestemmelingen, hoe groter de RMP-ring en hoe meer protocoloverlast.

Zoals verwacht vallen de gemeten doorvoersnelheden terug naar een lagere waarde bij gebruik van XDR, vanwege de extra rekenlast die deze omzetting meebrengt. Daar deze rekenlast slechts eenmaal dient te worden aangerekend bij een verzending van data naar meerdere bestemmelingen, is de doorvoersnelheid voor unicastverzending met $p = 7$ bestemmelingen voor configuratie B een stuk hoger dan de corresponderende waarde voor een bestemmeling. Dit blijkt echter niet te gelden voor multicastverzending.

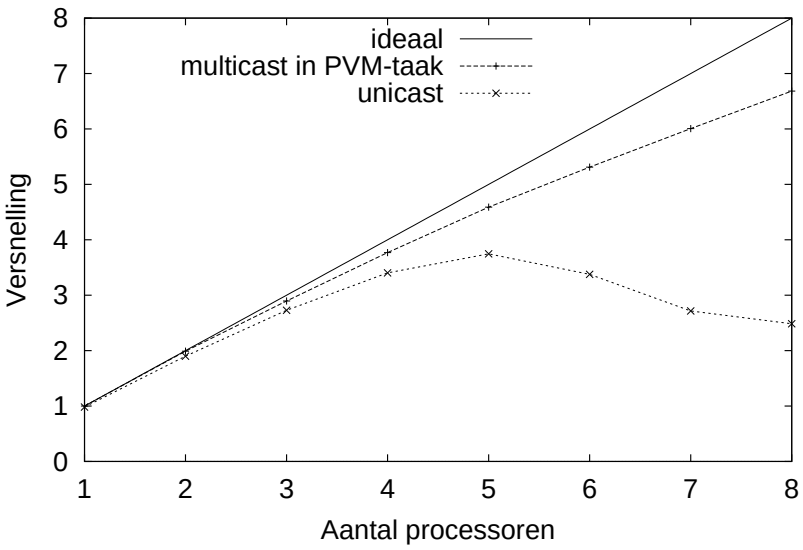
De laatste kolom, winstfactor multicast genaamd, geeft aan wat de maximaal te verwachten vermindering in communicatietijd door het toepassen van multicast in combinatie met XDR is. De basis voor het berekenen van deze winstfactor is de doorvoersnelheid met unicast. Deze factor blijkt aanzienlijk groter voor configuratie A dan voor configuratie B, wegens de mindere prestaties van het RMP-protocol voor configuratie B.

**Resultaten testprogramma's** De uitvoeringstijd van de sequentiële versie van beide programma's werd als referentie genomen: voor $n = 800$ zijn deze tijden 548 s en 236 s op platform A, en 154 s en 66 s op platform B voor het MV-programma resp. het GJ-programma. De meetresultaten voor de MV- en GJ-programma's op beide platformen zijn samengevat in figuren 4.10, 4.11, 4.12 en 4.13 en ook in tabel 4.4. Er zijn geen resultaten op platform A voor de multicastimplementatie in de PVM-daemon omdat deze implementatie pas gebeurde nadat platform A niet meer beschikbaar was.

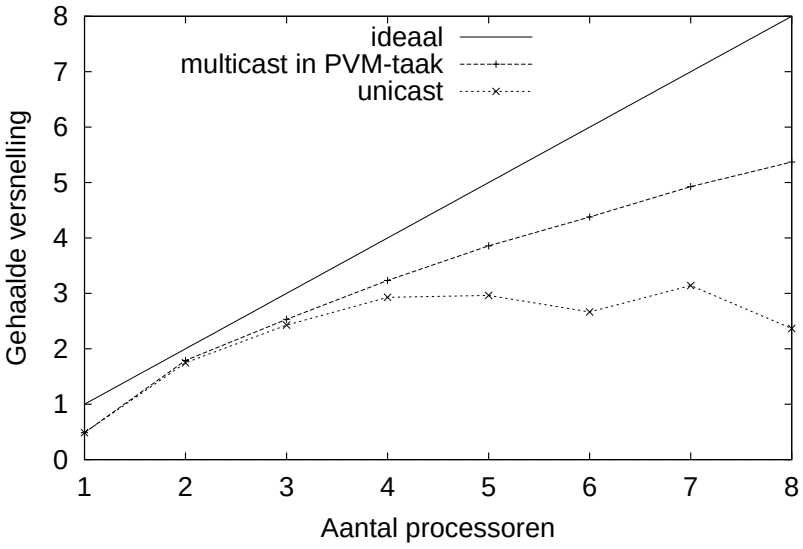
Er wordt verondersteld dat de distributie van de uitvoeringstijd benaderd wordt door $\phi(t) = \mu(t - t_0)\lambda \exp^{-\lambda(t-t0)}$, of een exponentiële distributie met ondergrens $t_0$ en gemiddelde $t_0 + \lambda^{-1}$. Hierbij is $\mu(t)$ de stapfunctie. Deze verdeling geldt voor deterministische programma's met gegeven invoer. Uit de experimenten volgt dat de standaardafwijking van de metingen sterk toeneemt samen met het aantal processors. Om nog een verband te kunnen leggen tussen de metingen en de implementatie van het communicatieprotocol wordt er gebruik gemaakt van het minimum van de uitvoeringstijden i.p.v. het gemiddelde. Elke uitvoeringstijd werd minstens 15 maal gemeten. De standaardafwijking op de gemeten tijden is 3% voor platform A en 0.2% voor platform B. Hoe groter de standaardafwijking, hoe minder monotoon de curve in de grafiek – dit effect is bijvoorbeeld merkbaar in de grafiek van de versnelling gehaald voor PVM in figuur 4.11.

Multicast is voordelig voor het versturen van initiële data naar de neventaken, maar niet voor het verzamelen van de resultaten door de hoofdtaak. Het MV- en het GJ-programma versturen initieel ong. 8 MB resp. 4 MB data naar de neventaken. Het communicatiepatroon van deze programma's is sterk verschillend: terwijl het programma MV zijn data in een groot blok verstuurt, verstuurt het GJ programma bij elke iteratie een deel van de data – gemiddeld 1600 bytes per iteratie. Doordat multicast interessanter wordt naarmate de hoeveelheid data per verzending toeneemt, geeft multicast een sterkere reductie van de communicatietijd voor MV dan voor GJ – zie tabel 4.4.

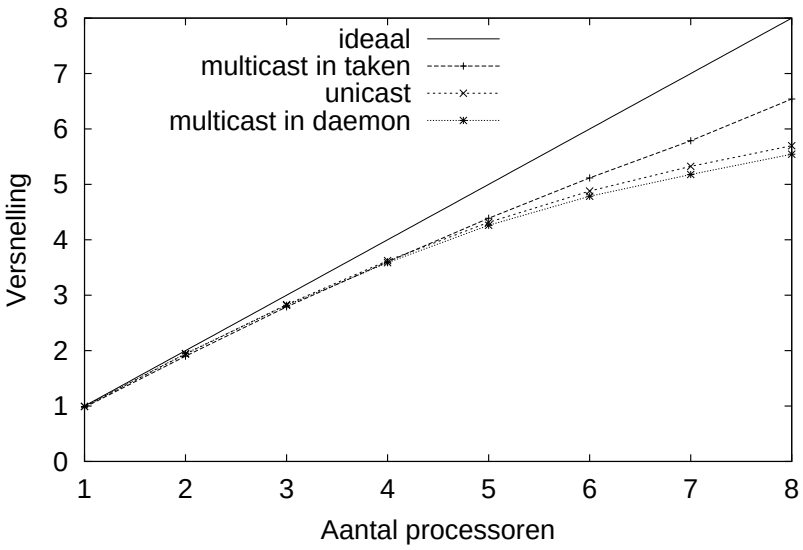
Uit dezelfde tabel 4.4 kunnen we verder besluiten dat de winst door het gebruik van multicast duidelijk kleiner is voor configuratie B dan voor configuratie A. Dit is een verwacht resultaat op basis van de in de vorige paragraaf besproken winstfactoren. Mits de foutmarge op de reductiefactoren in tabel 4.4 in acht wordt genomen is hun waarde de verwachte waarde uit tabel 4.3. Verder valt op dat de uitvoeringstijd



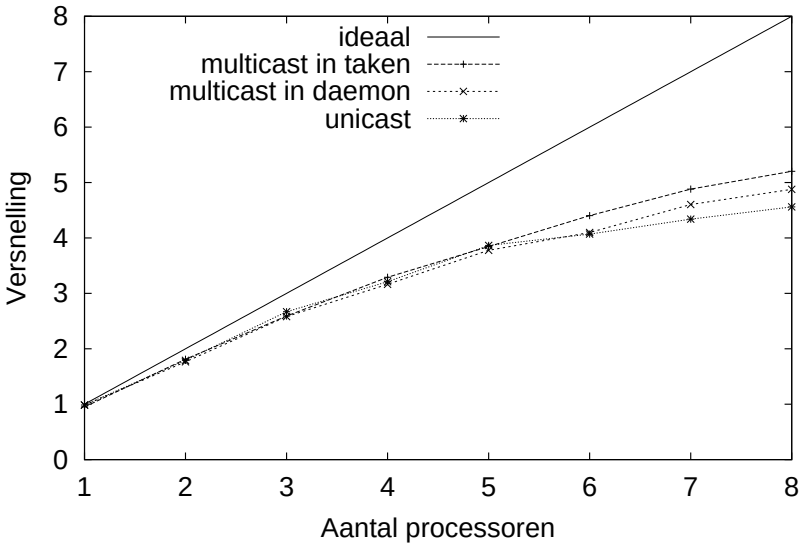

**Figuur 4.10:** Gehaalde versnelling voor de vermenigvuldiging van twee $800 \times 800$ matrices op een 10 Mbit/s netwerk.




**Figuur 4.11:** Gehaalde versnelling voor het oplossen van 800 lineaire vergelijkingen op een 10 Mbit/s netwerk.




**Figuur 4.12:** Gehaalde versnelling voor de vermenigvuldiging van twee $800 \times 800$ matrices op een 100 Mbit/s netwerk.




**Figuur 4.13:** Gehaalde versnelling voor het oplossen van 800 lineaire vergelijkingen op een 100 Mbit/s netwerk.

| configuratie | testprogramma | sequentiële tijd $t_s$ | | versnelling $v$ | communicatietijd $t_c = t_s/v - t_s/p$ | reductie communicatietijd $t_{u,c}/t_c$ |
|---|---|---|---|---|---|---|
| A | MV | 550.0 | unicast | 2.6 | 144.6 | 1.0 |
| | | | multicast in taken | 6.7 | 13.5 | 10.7 |
| A | GJ | 236.0 | unicast | 2.4 | 70.2 | 1.0 |
| | | | multicast in taken | 5.4 | 14.4 | 4.9 |
| B | MV | 153.7 | unicast | 5.7 | 7.8 | 1.0 |
| | | | multicast in daemons | 5.5 | 8.5 | 0.9 |
| | | | multicast in taken | 6.5 | 4.3 | 1.8 |
| B | GJ | 66.4 | unicast | 4.6 | 6.3 | 1.0 |
| | | | multicast in daemons | 4.9 | 5.3 | 1.2 |
| | | | multicast in taken | 5.2 | 4.5 | 1.4 |

**Tabel 4.4:** Gehaalde versnelling bij de uitvoering van het matrixvermenigvuldigings- (MV) en het Gauss-Jordan-programma (GJ) op een 10 Mbit/s (A) en 100 Mbit/s (B) netwerk en op $p = 7$ processors. De relatieve fout op de vermelde uitvoeringstijden is 3% (A) resp. 0.2% (B), en de relatieve fout op de berekende reductie is 40% (A) resp. 3% (B).

voor de implementatie van multicast in de PVM-daemons voor MV groter is dan de uitvoeringstijd door toepassing van unicast. De oorzaak hiervan ligt bij de implementatie van multicast in de daemons, die namelijk alle data in blokken van 4 KB verstuurt. Deze implementatie kan dus geen voordeel halen uit het effect dat de doorvoersnelheid voor het RMP-protocol toeneemt bij toename van de blokgrootte, en kan ook niet eenvoudig aangepast worden om blokken samen te nemen tot grotere gehelen.

## 4.8 Verwant werk

In deze paragraaf wordt dieper ingegaan op de verschillende mogelijkheden voor het optimaliseren van de communicatie tussen taken op verschillende hosts: optimalisatie van de software die de berichten behandelt, optimalisatie van het besturingssysteem, overgaan naar een sneller netwerk en het toepassen van multicast.

**Optimalisatie van de communicatiesoftware** Er zijn naast de gedistribueerde applicatie twee niveaus waarop software betrokken is bij berichtencommunicatie: de berichtenbehandelingssoftware die de berichten tussen de applicatie en het besturingssysteem doorgeeft (zoals PVM of MPI), en ook het besturingssysteem dat de berichten doorgeeft tussen de netwerkhardware en de berichtenbehandelingssoftware. Wat betreft het niveau van het besturingssysteem is er de toevoeging aan het besturingssysteem van multicast voor het ATM-protocol door G. Armitage [Arm97]. Door optimalisatie van de laag met de berichtencommunicatie voor homogene netwerken bereikte S. White aanzienlijk hogere prestaties met PVM [WAS95]. S. Chang daarentegen optimaliseerde de één-naar-meerdere communicatie binnen PVM voor een ATM-netwerk [CDH+95]. K. Hall implementeerde multicast in PVM, maar haalde geen versnelling [Hal94]. Vermoedelijk lag dit aan de kleine pakketgrootte van 1 KB in combinatie met het gebruik van een WAN. Multicast kan niet alleen met voordeel ingezet worden in een berichtengebaseerd systeem, maar ook in een DSM-systeem. In het DSM-systeem Brazos heeft het toepassen van multicast aanleiding gegeven tot een aanzienlijke prestatieverbetering: deze varieert tussen 0% en 100% met een gemiddelde 38%, afhankelijk van de beschouwde toepassing [SB97, SB98].

**Aanpassing besturingssysteem** DSM-systemen bestaan niet alleen als laag tussen applicatie en besturingssysteem, maar ook als onderdeel van het besturingssysteem. Zo past bijvoorbeeld het gedistribueerde besturingssysteem Amoeba [MvT+90] multicast toe om een sequentieel consistent DSM-systeem te implementeren. Nog een andere aanpak wordt gevolgd in de recent gestandaardiseerde VI-architectuur (*virtual interface*), die toelaat berichten zonder tussenkomst van het besturingssysteem over het netwerk te versturen [DRM+98, SAB99]. Deze techniek staat ook bekend onder de namen *zero-copy* of *virtual memory-mapped* communicatie, en werd reeds eerder toegepast voor U-net-netwerken [WBHvE98] en voor Myrinet-netwerken [BGM99]. Op dit ogenblik levert deze techniek de hoogst gekende prestaties voor berichtencommunicatie.

**Snelle netwerkhardware** Voor de netwerkhardware maken we onderscheid tussen verbeteringen die alleen de multicastprestaties verhogen of verbeteringen die zowel de unicast- als de multicastsnelheid verhogen. Tot de eerste categorie behoort het onderzoek om bij ATM-switches in hardwareondersteuning voor multicast te voorzien, samengevat door M. Guo in [GC99, TY98]. Tot de tweede categorie behoren de snellere types netwerken met bijhorende netwerkkaarten. Voorbeelden hiervan zijn het Myrinet-netwerk [BGM99] en het ParaStation-netwerk [WBT98]. Beide netwerken hebben bijhorende berichtencommunicatiesoftware die zoveel mogelijk het besturingssysteem omzeilt voor berichtencommunicatie. Een recent voorbeeld is het cLAN-netwerk (*cluster LAN, maximaal 1.25 Gbit/s*) van de onderneming GigaNet, dat gebaseerd is op de VI-architectuur [vV98].

## 4.9 Besluit

Alhoewel het inzetten van multicast in een bestaande PVM-applicatie slechts kleine veranderingen vereist, kan multicast voor een sterke prestatietoename zorgen. De prestatietoename verhoogt naarmate volgende invloedsfactoren verbeteren:

- Het aandeel van de unicastcommunicatie dat door multicastcommunicatie kan worden vervangen.
- Het aantal ingezette processors.

- Het aandeel van de communicatietijd in de totale uitvoeringstijd.
- De grootte van de in multicast te versturen datablokken.
- De prestatieverhouding van het toegepaste multicastprotocol t.o.v. het toegepaste unicastprotocol.

Verder geldt dat:

- De latentie van de berichtenversturing dient zo klein mogelijk te zijn. Het belang om XDR-datavertaling uit te schakelen neemt toe naarmate de netwerksnelheid toeneemt.
- Het toegepaste multicastprotocol dient voordeel te kunnen halen uit de transmissie van grote datablokken t.o.v. kleine datablokken.

# Hoofdstuk 5

# Besluit

## 5.1 Besluiten van het onderzoek

In dit proefschrift werden twee facetten van het verband tussen enerzijds communicatie in gedistribueerde systemen en anderzijds berichtendoorgave en gemeenschappelijk geheugen uitgewerkt. Het eerste facet is een systematische studie van geheugenmodellen voor gedistribueerde gemeenschappelijk-geheugensystemen. Een geheugenmodel bepaalt zowel de programmeringswijze van het systeem als de hoeveelheid gegenereerde communicatie. Er wordt een beschrijvingsmethode voorgesteld die de beschrijving en vergelijking van geheugenmodellen eenvoudiger maakt dan met bestaande methodes. Het tweede facet is het onderzoek naar de invloed van de communicatietechniek genaamd multicast op de prestaties van gedistribueerde programma's gebaseerd op berichtendoorgave.

Het in hoofdstuk 3 voorgestelde formalisme laat toe om geheugenmodellen en hun eigenschappen nauwkeurig te beschrijven. Geheugenmodellen zijn fundamenteel voor parallelwerking op basis van gemeenschappelijk geheugen. Het formalisme omvat naast lees- en schrijfopdrachten ook atomische lees/wijzig/schrijf-opdrachten, vier soorten etiketten die controle over herordening van opdrachten binnen een proces mogelijk maken, en synchronisatieopdrachten. Het beschrijven van geheugenmodellen in dit formalisme laat toe geheugenmodellen onderling eenvoudiger te vergelijken, en de voorgestelde aanpak is bovendien geschikt voor automatische bewijsvoering. Ook werd er onderscheid gemaakt tussen modelspecifieke en modelonafhankelijke eigenschappen, wat de eenvoud van een specificatie ten

goede komt. Ook werd de geschiktheid aangetoond van het formalisme om berichtencommunicatiemodellen te definiëren.

Alhoewel het inzetten van multicast in een bestaande PVM-applicatie slechts kleine aanpassingen vereist, kan multicast voor een sterke prestatietoename zorgen. In hoofdstuk 4 werd aangetoond welke factoren in de verschillende lagen van het communicatieprotocol gunstig of ongunstig kunnen bijdragen tot de prestaties van een programma. Dit onderzoek werd uitgevoerd voor lokale netwerken.

## 5.2 Verder onderzoek

Naast de onderzochte onderwerpen blijven er nog een aantal onderzoeksrichtingen open, die hieronder kort worden vermeld.

In dit werk werden verschillende geheugenmodellen met elkaar vergeleken en werden de factoren aangewezen die tot een reductie in communicatie kunnen leiden. Nadat gekozen werd welk geheugenmodel te implementeren, is er nog een grote keuzevrijheid in de wijze van implementeren. Alhoewel reeds geruime tijd onderzoek verricht wordt op dit terrein, zowel voor multiprocessors als voor gedistribueerde systemen, is het laatste woord hierover zeker nog niet geschreven.

Voor grootschalige parallelle programma's kan het nodig zijn gebruik te maken van een interlokaal netwerk (WAN) i.p.v. een lokaal netwerk (LAN). Dit brengt extra vertragingen mee in de berichtencommunicatie, wat gevolgen heeft voor de prestaties van parallelle programma's. Het vervangen van een lokaal door een interlokaal netwerk kan meebrengen dat voor optimale resultaten het multicastprotocol aan de nieuwe situatie dient te worden aangepast. Verder is het nuttig de invloed na te gaan van de toegepaste software voor berichtencommunicatie, waar in deze thesis werd uitgegaan van PVM.

# Bibliografie


[ABC+99] A. Agarwal, R. Bianchini, D. Chaiken, F. T. Chong, K. L. Johnson, D. Kranz, J. D. Kubiatowicz, B. H. Lim, K. Mackenzie, and D. Yeung. The mit alewife machine. *Proceedings of the IEEE*, 87(3):430–444, March 1999.

[ABJN93] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, and Gil Neiger. The power of processor consistency. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures (SPAA'93)*, 1993.

[ABM93] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

[ACD+96] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, pages 18–28, February 1996.

[ACFW98] H. Attiya, S. Chaudhuri, R. Friedman, and J. L. Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. *SIAM Journal on Computing*, 27(1):65–89, February 1998.

[Adv93] Sarita V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, December 1993.

[AF98] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors. *SIAM Journal on Computing*, 27(6):1637–1670, December 1998.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, February 1996.

[AGU72] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1:131–137, 1972.

[AH90a] S. V. Adve and M. D. Hill. Implementing sequential consistency in cache-based systems. In *Proc. of the 1990 Int'l Conf. on Parallel Processing (ICPP'90)*, pages 47–50, August 1990.

[AH90b] S. V. Adve and M. D. Hill. Weak ordering — A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, ACM SIGARCH Computer Architecture News*, pages 2–14, June 1990.

[AH93] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[AH98] S. V. Adve and M. D. Hill. *A Retrospective on "Weak Ordering–A New Definition"*, pages 60–62. Selected Papers from the First 25 International Symposia on Computer Architecture. ACM Press, 1998.

[ANK+95] M. Ahamad, G. Neiger, P. Kohli, J. E. Burns, and P. W. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9:37–49, 1995.

[APR99] Sarita V. Adve, S. V. Pai, and P. Ranganathan. Recent advances in memory consistency models for hardware shared-memory systems. *Proceedings of the IEEE*, 87(3):445–455, March 1999.

[Arm97] G. J. Armitage. IP multicasting over atm networks. *IEEE Journal on Selected Areas in Communications*, 15(3):445–457, April 1997.

[AW94] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[BBD+87] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., New York, NY, 1987.

[BF99] Gregory T. Bird and Michael J. Flynn. Producer-consumer communication in distributed shared memory multiprocessors. *Proceedings of the IEEE*, 87(3):456–466, March 1999.

[BFL+94] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, London, 1994. ISBN 3-540-19813-X.

[BGM99] A. Barak, I. Gilderman, and I. Metrik. Performance of the communication layers of TCP/IP with the myrinet gigabit LAN. *Computer Communications*, 22(11):989–997, July, 15 1999.

[BHW+93] C. Bendtsen, P. C. Hansen, J. Wasniewski, J. B. Hansen, J. N. Sorensen, and Z. Zlatev. Experience with the KSR-1 parallel computer. *Supercomputer*, 10(6):34–43, 1993.

[Bic98] Juan C. Bicarregui, editor. *Proof in VDM: Case studies*. FACIT – Formal Approaches to Computing and Information Technology. Springer-Verlag, London, March 1998.

[BR91] Juan Bicarregui and Brian Ritchie. Reasoning about VDM developments using the VDM support tool in Mural. In S. Prehn and W. J. Toetenel, editors, *Lecture Notes in Computer Science – VDM '91 – Formal Software Development Methods*, volume 551, pages 371–388. Springer-Verlag, New York, October 1991.

[BS72] J. Bruno and K. Steiglitz. The expression of algorithms by charts. *Journal of the ACM*, 19(3):517–525, July 1972.

[Car95] J. B. Carter. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29(2):219–227, September 1995.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Pro-*

*ceedings of 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[CDH+95] Sheue-Ling Chang, David Hung-Chang Du, Jenwei Hsieh, Rose P. Tsang, and Mengjou Lin. Enhanced PVM communications over a high-speed LAN. *IEEE Parallel and Distributed Technology*, 3(3):20–32, Fall 1995.

[CHPS99] A. Condon, M. Hill, M. Plakal, and D. Sorin. Using lamport clocks to reason about relaxed memory models. In *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture (HPCA-5)*, January 1999.

[CL94] R. P. Colwell and R. A. Lethin. Latent design faults in the development of the multiflow trace/200. *IEEE Transactions on Reliability*, 43(4):557–565, December 1994.

[CM69] D. Chazan and W. L. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2:199–222, 1969.

[Col92] William W. Collier. *Reasoning about Parallel Architectures*. Prentice-Hall, 1992.

[DCZ96] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. An integrated compile-time / run-time software distributed shared memory system. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 186–197, New York, October 1–5, 1996. ACM.

[Dee89] Steve E. Deering. *Host Extensions for IP Multicasting: RFC 1112*. Network Information Center, SRI International, Menlo Park, CA, August 1989. 17 pages.

[DH98] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. Technical report, Cisco Systems and Nokia, December 1998. RFC 2460.

[Dij65] E. W. Dijkstra. Solution to a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[DM98] L. Dagum and R. Menon. OpenMP: An industry standard api for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, January–March 1998.

[DOSW96] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A message passing standard for mpp and workstations. *Communications of the ACM*, 39(7):84–90, July 1996.

[DRM+98] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, 18(2):66–76, March–April 1998.

[DSB86] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, volume 14 (2), pages 434–442, Tokyo, Japan, June 2–5, 1986. IEEE Computer Society TCCA, ACM SIGARCH, and the Information Processing Society of Japan.

[Eri94] Hans Eriksson. MBONE: The multicast backbone. *Communications of the ACM*, 37(8):54–60, August 1994.

[Fly66] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, September 1972.

[GAG+92] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15:399–407, August 1992.

[GAG+93] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Specifying system requirements for memory consistency models. Technical Report CSL-TR-93-594, Computer Systems Laboratory, Stanford University, 1993.

[Gar96] Vijay K. Garg. *Principles of Distributed Systems*. Kluwer Academic Publishers, 1996.

[GC99] M. H. Guo and R. S. Chang. Design issues for multicast atm switches. *Computer Communications*, 22(9):771–777, June, 15 1999.

[GGH91a] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, Architecture, pages I–355–I–364, Boca Raton, FL, August 1991. CRC Press.

[GGH91b] Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. Performance evaluation of memory consistency models for shared memory multiprocessors. *ACM SIGPLAN Notices*, 26(4):245–257, April 1991.

[Gha95] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Computer Systems Laboratory, Stanford University, December 1995.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Henessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[GM92] Phillip B. Gibbons and Michael Merritt. Specifying nonblocking shared memories. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 306–315, 1992.

[GMG91] Phillip B. Gibbons, Michael Merritt, and Kourosh Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proceedings of the 3th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 292–303, 1991.

[Goo89] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, March 1989.

[HA90] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proc. of the 10th Int'l Conf. on Distributed Computing Systems (ICDCS-10)*, pages 302–311, May 1990.

[Hal94] Kara Ann Hall. The implementation and evaluation of reliable IP multicast. Master's thesis, Univer-

sity of Tennessee, Knoxville, December 1994. URL: http://www.epm.ornl.gov/pvm/hall_thesis.ps.

[HD98] R. Hinden and S. Deering. Ip version 6 addressing architecture. Technical report, Nokia and Cisco Systems, July 1998. RFC 2373.

[HH89] S. Haridi and E. Hagersten. The cache coherence protocol of the data diffusion machine. In *Lecture Notes in Computer Science*, volume 365, pages 1–18, 1989.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*, pages 708–721. Morgan Kaufmann Publishers, Inc., second edition, 1996.

[HW90] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[Int99] *IA-64 Application Architecture Developers Guide*. Intel Corporation, May 1999. Revision 1.0.

[IS99] L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. *Proceedings of the IEEE*, 87(3):498–507, March 1999.

[ISL98] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. *Theory of Computing Systems*, 31(4):451–473, July–August 1998.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *Mural: A formal development support system*. Springer-Verlag, London, 1991.

[KCDZ95] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. An evaluation of software-based release consistent protocols. *Journal of Parallel and Distributed Computing*, 29(2):126–141, September 1995.

[KCZ92] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *18th International Symposium on Computer Architecture*, pages 13–21. IEEE, 1992.

[KDCZ94] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Winter 1994 USENIX Conference*, pages 115–132, January 1994.

[KW97] M. Kurreck and S. Wittig. A comparative study of pressure correction and block-implicit finite volume algorithms on parallel computers. *International Journal for Numerical Methods in Fluids*, 24:1111–1128, 1997.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 9:690–691, September 1979.

[Lam97] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7), July 1997.

[LC99] J. D. Lin and R. S. Chang. A comparison of the internet multicast routing protocols. *Computer Communications*, 22(2):144–155, January 25 1999.

[LDCZ95] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Proceedings of Supercomputing '95*, December 1995.

[LDCZ97] H. H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Quantifying the performance differences between pvm and treadmarks. *Journal of Parallel and Distributed Computing*, 43(2):65–78, June 1997.

[LG98] B. N. Levine and J. J. Garcia Luna Aceves. A comparison of reliable multicast protocols. *Multimedia Systems*, 6(5):334–348, September 1998.

[LH94] Daniel H. Linder and James C. Harden. Access graphs: A model for investigating memory consistency. *IEEE Transactions on Parallel and Distributed Systems*, 5(1):39–52, 1 1994.

[Lie99] Kristof Lievens. Uitbreiding van de PVM-daemon met multicast-functionaliteit. Afstudeerwerk, Vakgroep ELIS, Universiteit Gent, June 1999.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In Jean-Loup Baer and Larry Snyder, editors, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, Seattle, WA, June 1990. IEEE Computer Society Press.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[LLJ+92] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Implementation and performance. In *19th International Symposium on Computer Architecture*, pages 92–103, 1992.

[Lov93] David B. Loveman. High performance fortran. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):25–42, 1993.

[LW95] Daniel E. Lenoski and Wolf-Dietrich Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann Publishers, 1995.

[Mis86] Jayadev Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, January 1986.

[Mos93] David Mosberger. Memory consistency models. *ACM Operating Systems Review*, 27(1):18–26, January 1993.

[MRZ98] P. Mehrotra, J. Van Rosendale, and H. Zima. High performance fortran: History, status and future. *Parallel Computing*, 24(3–4):325–354, May 1998.

[MvT+90] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[NM92] Robert H. B. Netzer and Barton P. Miller. What are race conditions? Some issues and formalizations. *ACM Letters*

*on Programming Languages and Systems*, 1(1):74–88, March 1992.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS – A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Pau98] Sanjoy Paul. *Multicasting on the Internet and its Applications*. Kluwer Academic Publisher, Norwell, USA, 1998.

[PBA+98] F. Pong, M. Browne, G. Aybay, A. Nowatzyk, and M. Dubois. Design verification of the S3.mp cache-coherent shared-memory system. *IEEE Transactions on Computers*, 47(1):135–140, January 1998.

[PD95] Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, August 1995.

[PD98] Fong Pong and Michel Dubois. Formal verification of complex coherence protocols using symbolic state models. *Journal of the ACM*, 45(4):557–587, July 1998.

[Pos81] Jon Postel. Internet protocol – DARPA internet program protocol specification. Technical report, Information Sciences Institute University of Southern California, October 1981. RFC 791 (IPv4).

[PSLB97] S. Paul, K. K. Sabnani, J. C. H. Lin, and S. Bhattacharyya. Reliable multicast transport protocol (RMTP). *IEEE Journal on Selected Areas in Communications*, 15(3):407–421, April 1997.

[Ron99] Michiel Ronsse. *Racedetectie in Parallelle Programma's door Gecontroleerde Heruitvoering*. PhD thesis, Universiteit Gent, May 1999.

[RS97] M. Raynal and A. Schiper. A suite of definitions for consistency criteria in distributed shared memories. *Annales Des Telecommunications*, 52(11–12):652–661, November–December 1997.

[SAB99] W. E. Speight, H. Abdel-Shafi, and J. K. Bennett. Realizing the performance potential of the virtual interface architecture. In *Proceedings of the 13th ACM-SIGARCH International Conference on Supercomputing (ICS'99)*, June 1999.

[SAR99] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-reconcile & fences (CRF): A new memory model for architects and compiler writers. In *26th International Symposium on Computer Architecture – ISCA'99*, pages 150–161, May 1999.

[SB97] Evan Speight and John K. Bennett. Brazos: A third generation DSM system. In USENIX, editor, *The USENIX Windows NT Workshop*, pages 95–106, Berkeley, CA, USA, August11–13 1997. USENIX.

[SB98] W. E. Speight and J. K. Bennett. Using multicast and multithreading to reduce communication in software DSM systems. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pages 312–322, February 1998.

[SBE+99] Tony Speakman, Nidhi Bhaskar, Richard Edmonstone, Dino Farinacci, Steven Lin, Alex Tweedly, Lorenzo Vicisano, and Jim Gemmell. Pgm reliable transport protocol specification. Technical report, Cisco Systems and Microsoft, June, 24 1999. Internet draft. Expires 24 December 1999.

[SBKK98] Yangfeng Su, Amit Bhaya, Eugenius Kaszkurewicz, and Victor S. Kozyakin. Further results on convergence of asynchronous linear iterations. *Linear Algebra and its Applications*, 281(1–3):11–24, September 1998.

[SFC92] P. S. Sindhu, J-M. Frailong, and M. Cekleov. Formal specification of memory models. In M. Dubois and S. S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*, pages 25–41. Kluwer Academic Publishers, 1992.

[SGDM94] V.S. Sunderam, G.A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences and trends. *Parallel Computing*, 20:531–545, 1994.

[SI95] Terunao Soneoka and Toshihide Ibaraki. Logically instantaneous message passing in asynchronous distributed

systems. *IEEE Transactions on Parallel and Distributed Systems*, 43(5):513–527, May 1995.

[Sin95] Ambuj K. Singh. A framework for programming with nonatomic memories. *Journal of Parallel and Distributed Computing*, 26:211–224, May 1995.

[Sla96] Noemie Slaats. The mural theory for shared memory synchronization. Technical Report R96010, Vakgroep Zuivere Wiskunde en Computeralgebra (CAGE), Universiteit Gent, September 1996.

[SMS96] T. Sterling, P. Merkey, and D. Svarese. Improving application performance on the HP/Convex Exemplar. *Computer*, 29(12):50–, December 1996.

[SPA92] SPARC International Inc. *The SPARC Architecture Manual: Version 8*, 1992.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[SS93] M. A. Schuette and J. P. Shen. Instruction-level experimental evaluation of the multiflow trace-14-300 VLIW computer. *Journal of Supercomputing*, 7(1–2):249–271, May 1993.

[Str97] John C. Strikwerda. A convergence theorem for chaotic asynchronous relaxation. *Linear Algebra and its Applications*, 253(1–3):15–24, March 1997.

[SVAH98] Noemie Slaats, Bart Van Assche, and Albert Hoogewijs. *Shared Memory Synchronization*, chapter 4, pages 123–156. FACIT – Formal Approaches to Computing and Information Technology. Springer-Verlag, March 1998.

[TY98] J. Turner and N. Yamanaka. Architectural choices in large scale atm switches. *IEICE Transactions on Communications*, 2:120–137, February 1998.

[VA95] Bart Van Assche. Een implementatie van gedistribueerd gedeeld geheugen onder het Mach besturingssysteem, May 1995. Afstudeerwerk ingediend tot het behalen van

de academische graad van burgerlijk ingenieur in de computerwetenschappen.

[VA96] Bart Van Assche. DSM under Mach: A quantitative approach. In *Parallel Computing: State-of-the-Art and Perspectives*, pages 463–470. Elsevier, September 1996.

[VA97] Bart Van Assche. IP multicast for PVM on bus based networks. In *Proceedings of Parallel Computing Conference*, volume 12, pages 403–410. Elsevier, Amsterdam, September 1997.

[VAD96a] Bart Van Assche and Erik H. D'Hollander. Code generation for networks of workstations. In *Parallel Computing Seminar*, pages 120–124, Noordwijk, October 1996.

[VAD96b] Bart Van Assche and Erik H. D'Hollander. Message passing versus distributed shared memory. In *Symposium on Knowledge and Information Technology*, pages 61–68, Gent, September 23–24, 1996.

[VAD97] Bart Van Assche and Erik H. D'Hollander. *Networks of Workstations: DSM or MP Strategy ?*, pages 89–94. T.U. Clausthal, May 1997.

[VAD00] Bart Van Assche and Erik D'Hollander. A framework for the investigation of shared memory systems. In *Fifth International Conference on Computer Science and Informatics (CS&I-2000)*, pages 512–518, 27 February – 3 March 2000.

[vV98] Thorsten von Eicken and Werner Vogels. Evolution of the Virtual Interface Architecture. *Computer*, 31(11):61–68, November 1998.

[WAS95] S. White, A. Alund, and V. S. Sunderam. Performance of the NAS parallel benchmarks on PVM-based networks. *Journal of Parallel and Distributed Computing*, 26(1):61–71, April 1995.

[WBHvE98] M. Welsh, A. Basu, X. W. Huang, and T. von Eicken. Memory management for user-level network interfaces. *IEEE Micro*, 18(2):77–82, March–April 1998.

[WBT98] T. M. Warschko, J. M. Blum, and W. F. Tichy. The ParaStation project: Using workstations as building blocks for parallel computing. *Information Sciences*, 106(3–4):277–292, May 1998.

[WMK95] Brian Whetten, Todd Montgomery, and Simon Kaplan. *A high performance totally ordered multicast protocol*, volume 938 of *Lecture Notes in Computer Science – Theory and Practice in Distributed Systems*, pages 33–57. Springer-Verlag, 1995.

# Bijlage A

# Wiskundige relaties

## A.1 Definities

Een **relatie** $R$ tussen de verzamelingen $A$ en $B$ geldt voor de elementen $a \in A$ en $b \in B$, met als notatie $aRb$, of geldt niet, met als notatie $\neg(aRb)$. Met elke relatie $R$ is een verzameling $\{(a,b)|aRb\}$ geassocieerd. Voor deze verzameling wordt ook de notatie $R$ gebruikt. De **unie** van twee relaties $R_1$ en $R_2$ wordt gegeven door $R_1 \cup R_2$. Relatie $R_1$ is een **deelrelatie** van relatie $R_2$ als en slechts als $R_1 \subset R_2$.

Het **domein** van een relatie $R$ is de verzameling elementen waarvoor tenminste een ander element bestaat zodanig dat $R$ geldt: $dom(R) \triangleq \{a|\exists b:\ aRb \vee bRa\}$.

Een relatie $R$ over $A$ is **reflexief**, **symmetrisch**, **anti-symmetrisch** of **transitief** als resp. $\forall\, a \in A:\ aRa$, $\forall\, a_1, a_2 \in A:\ a_1Ra_2 \iff a_2Ra_1$ $\forall\, a_1, a_2 \in A:\ a_1Ra_2 \wedge a_2Ra_1 \implies a_1 = a_2$ of $\forall\, a_1, a_2, a_3 \in A:\ a_1Ra_2 \wedge a_2Ra_3 \implies a_1Ra_3$ geldt.

Een **partiële ordening** (p.o.) is een relatie die reflexief is, anti-symmetrisch en transitief. Een relatie $R$ is **totaal in** de verzameling $A$ als en slechts als $\forall\, a_1, a_2 \in A:\ a_1Ra_2 \vee a_2Ra_1 \vee a_1 = a_2$. Een relatie $R$ is **totaal** als en slechts als de relatie $R$ totaal is in haar domein. Een **totale ordening** (t.o.) is een partiële ordeningsrelatie die ook totaal is. In dit proefschrift zijn partiële en totale ordeningsrelaties reflexief, tenzij anders wordt vermeld.

Een **equivalentierelatie** $Q$ over de verzameling $A$ is een relatie die reflexief, symmetrisch, transitief en totaal is in $A$. Deze equivalentierelatie creëert een partitie in die verzameling. De **partitie** wordt geno-

teerd als $A/Q$, met $A/Q = \{S[a]|a \in A\}$, en met $S[a]$ de equivalentieklasse **equivalentieklasse** voor het element $a \in A$, gedefinieerd als $S[a] = \{a' \in A|aQa'\}$.

De **quotiënt-relatie** van een gegeven relatie $R$ in $A$ en quotiëntrelatie $Q$ in $A$ wordt genoteerd als $R/Q$ en is gedefinieerd als
$R/Q = \{(S_1[a_1], S_2[a_2])|a_1Ra_2\}$.

De relaties $R_1 \ldots R_n$ zijn **consistent** over de verzameling $A$ als en slechts als de relaties $R_1$ t.e.m. $R_n$ identiek zijn wanneer hun restrictie tot $A$ wordt beschouwd: $R_1 \cap A^2 = \ldots = R_n \cap A^2$. Relaties zijn consistent als ze consistent zijn over de verzameling $dom(R_1) \cup \ldots \cup dom(R_n)$.

De relaties $R_1$ t.e.m. $R_n$ zijn **sequentieel consistent** (s.c.) over de verzameling $A$ als en slechts als $R_1$ t.e.m. $R_n$ consistent zijn en tevens $R_1$ t.e.m. $R_n$ totale ordeningen zijn in $A$.

Twee relaties $R_1$ en $R_2$ **conflicteren** als en slechts als $\exists a, b : \ aR_1b \wedge bR_2a$.

Twee relaties zijn **disjunct** als en slechts als hun domeinen disjunct zijn.

De **samenstelling** van twee relaties, genoteerd als $(R_1; R_2)$, wordt gegeven door $\forall a_1, a_3 : \ a_1(R_1; R_2)a_3 \iff \exists a_2 : \ a_1R_1a_2 \wedge a_2R_2a_3$.

Een relatie $R$ verheven tot de **macht** $n \in \mathbb{N}$ wordt als volgt recursief gedefinieerd:

$$R^0 = \{(a, a)|a \in dom(R)\}$$
$$\forall\, n \in \mathbb{N} : \ R^{n+1} = (R^n; R).$$

De **reflexieve en transitieve sluiting** $R^\star$ van de relatie $R$ wordt gegeven door $R^\star \triangleq \bigcup_{n=0}^{\infty} R^n$.

Een **lineaire extensie** van een anti-symmetrische relatie $R$ over de verzameling $A$ wordt genoteerd als $\text{linext}_A\, R$ en is een partiële ordening, is buiten $A$ identiek aan $R$ en is binnen $A$ een totale ordening. Een lineaire extensie modulo de equivalentierelatie $Q$ van een anti-symmetrische relatie $R$ over de verzameling $A$ wordt als volgt gedefinieerd:

$$T = \text{linext}_A^N\, R \iff (R/N \subset T/N) \wedge (T/N \text{ is totaal in } A/N).$$

Voor een volledige definitie van **transitieve reductie** verwijzen we naar Aho [AGU72]. We vermelden hier alleen de belangrijkste eigenschap, nl. dat de transitieve reductie van een transitieve relatie toelaat deze relatie met minder koppels voor te stellen, terwijl het nog mogelijk is de originele relatie te reconstrueren. In het algemeen geldt dat als

de relatie $R^-$ is een transitieve reductie is van de relatie $R$, dan voldoet deze aan de eigenschap $\{R^-\}^\star = R^\star$. Als $R$ reflexief en transitief is, dan is $\{R^-\}^\star = R$.

Een functie $P$ is een **permutatie** in de verzameling $A$ als $P$ bijectief is in $A$. De inverse permutatie wordt genoteerd als $P^{-1}$, en heeft als definiërende eigenschap $\forall\, a_1, a_2 \in A :\ P(a_1) = a_2 \Longrightarrow P^{-1}(a_2) = a_1$.

De toepassing van een permutatie $P$ op een relatie $R$ wordt genoteerd als $P(R)$, en wordt als volgt gedefinieerd: $\forall a_1, a_2 :\ a_1 R a_2 \iff P(a_1)P(R)P(a_2)$. $P(R)$ is een relatie in $dom(R)$ met dezelfde eigenschappen als $R$ betreffende reflexiviteit, irreflexiviteit, symmetrie, anti-symmetrie, transitiviteit en totaal zijn.

## A.2 Eigenschappen

In deze paragraaf stelt de relatie $R_1$ een totale ordening voor over de verzameling $A_1$, stelt de relatie $R_2$ een partiële ordening over $A_2$, is de verzameling $A$ de unie van de verzamelingen $A_1$ en $A_2$, en de relaties $R_1$ en $R_2$ conflicteren niet. De relatie $R$ is gedefinieerd als $R_1 \cup R_2$.

**Lemma A.2.1** *Als $R_1$ een partiële ordening is en $R_2 \subset R_1$, dan is $R_2^\star$ een partiële ordening en bovendien is het een deelrelatie van $R_1$.*

**Bewijs**

$R_2^\star$ is per definitie reflexief en transitief. Daar $R_2 \subset R_1$, volgt er dat $\forall\, n \in \mathbb{N} :\ R_2^n \subset R_1^n$. Uit de definitie van transitieve sluiting en uit de transitiviteit van $R_1$ volgt dat $R_2^\star \subset R_1^\star$. Omdat $R_1^\star = R_1$ is $R_2^\star \subset R_1$ en dus anti-symmetrisch.
□

**Lemma A.2.2** $\forall\, a_1, a_2, a_3, a_4 \in A :\ (a_1 R_1 a_2 \wedge a_2 R_2 a_3 \wedge a_3 R_1 a_4 \Longrightarrow a_1 R_1 a_4)$

**Bewijs**

$$
\begin{array}{ll}
 & a_1 R_1 a_2 \wedge a_2 R_2 a_3 \wedge a_3 R_1 a_4 \\
(R_1 \text{ is totaal}) & \iff a_1 R_1 a_2 \wedge a_2 R_2 a_3 \wedge a_3 R_1 a_4 \\
 & \quad \wedge (a_2 R_1 a_3 \vee a_3 R_1 a_2) \\
(\text{distr. } \wedge) & \iff a_1 R_1 a_2 \wedge a_3 R_1 a_4 \\
 & \quad \wedge ((a_2 R_2 a_3 \wedge a_2 R_1 a_3) \vee (a_2 R_2 a_3 \wedge a_3 R_1 a_2)) \\
(R_1, R_2 \text{ conflictvrij}) & \iff a_1 R_1 a_2 \wedge a_3 R_1 a_4 \\
 & \quad \wedge (a_2 = a_3 \vee (a_2 R_2 a_3 \wedge a_3 R_1 a_2)) \\
(\text{refl. } R_1, R_2) & \iff a_1 R_1 a_2 \wedge a_3 R_1 a_4 \wedge a_2 R_2 a_3 \\
 & \quad \wedge a_2 R_1 a_3 \\
(\text{trans. } R_1) & \implies a_1 R_1 a_4
\end{array}
$$

**Lemma A.2.3** $\exists\, a_2, a_3, a_4 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_4 \wedge a_4 R_1 a_5$
$\iff \exists\, a_2 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_5$

**Bewijs**

Van links naar rechts, door toepassen van lemma A.2.2:

$$
\begin{array}{ll}
 & \exists\, a_2, a_3, a_4 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_4 \wedge a_4 R_1 a_5 \\
(\text{lemma A.2.2}) & \implies \exists\, a_2 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_5
\end{array}
$$

Van rechts naar links:

$$
\begin{array}{ll}
 & \exists\, a_2 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_5 \\
(\text{refl. } R_1, R_2) & \implies \exists\, a_2 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_2 \wedge a_2 R_2 a_2 \wedge a_2 R_1 a_5 \\
(\text{eig. } \exists) & \implies \exists\, a_2, a_3, a_4, a_5 \in A : \\
 & \quad a_1 R_2 a_2 \wedge a_2 R_1 a_2 \wedge a_2 R_2 a_2 \wedge a_2 R_1 a_5
\end{array}
$$

**Lemma A.2.4**
$(\exists\, n \in \mathbb{N} :\ a_1 R^n a_5) \iff \exists\, a_2, a_3 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_5$.

**Bewijs**

• Inductiebasis
Als $n = 0$, dan is het linkergedeelte $a_1 R^n a_5$ identiek aan $a_1 = a_5$. Met $a_2 = a_1$ en $a_3 = a_5$ geldt het rechtergedeelte omdat $R_1$ en $R_2$ reflexief zijn. Dus de inductiebasis geldt.
• Inductiehypothese
Met $n \in \mathbb{N}$ is de inductiehypothese als volgt:

$$
\begin{array}{l}
(a_1 R^n a_5 \iff \exists\, a_2, a_3 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_5) \\
\implies (a_1 R^{n+1} a_5 \iff \exists\, a_2, a_3 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_5)
\end{array}
$$

• Bewijs

Met $n \in \mathbb{N}$ volgt er voor van links naar rechts:

$$
\begin{array}{ll}
 & a_1 R^{n+1} a_5 \\
\text{(def. } R^{n+1}\text{)} & \iff \exists\, a_4 \in A :\ a_1 R^n a_4 \wedge a_4 R a_5 \\
\text{(ind. basis, def. } R\text{)} & \iff \exists\, a_2, a_3, a_4 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_4 \\
 & \qquad\qquad \wedge (a_4 R_1 a_5 \vee a_4 R_2 a_5) \\
\text{(distr. } \wedge\text{)} & \iff \exists\, a_2, a_3, a_4 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \\
 & \qquad\qquad \wedge a_3 R_2 a_4 \wedge a_4 R_1 a_5 \\
 & \qquad \vee\, a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_4 \wedge a_4 R_2 a_5 \\
\text{(eig. } \exists\text{, eig. } \vee\text{)} & \iff \exists\, a_2, a_3, a_4 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \\
 & \qquad\qquad \wedge a_3 R_2 a_4 \wedge a_4 R_1 a_5 \\
 & \quad \vee \exists\, a_2, a_3, a_4 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \\
 & \qquad\qquad \wedge a_3 R_2 a_4 \wedge a_4 R_2 a_5 \\
\text{(lemma A.2.3,} & \implies \exists\, a_2 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_5 \\
\text{trans. } R_2\text{)} & \quad \vee \exists\, a_2, a_3 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_5 \\
\text{(refl. } R_2\text{, eig. } \exists\text{)} & \implies \exists\, a_2, a_3 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_5
\end{array}
$$

Van rechts naar links:

$$
\begin{array}{ll}
 & \exists\, a_2, a_3 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_5 \\
\text{(def. } R\text{)} & \implies \exists\, a_2, a_3 \in A :\ a_1 R a_2 \wedge a_2 R a_3 \wedge a_3 R a_5 \\
\text{(def. } R^n\text{)} & \implies a_1 R^3 a_5 \\
\text{(eig. } \exists\text{)} & \implies \exists\, n \in \mathbb{N} :\ a_1 R^n a_5
\end{array}
$$

Dit bewijst de inductiehypothese.
□

**Lemma A.2.5** *$R = (R_1 \cup R_2)$ is anti-symmetrisch en reflexief.*

**Bewijs**

Zowel de anti-symmetrie als de reflexiviteit kunnen samen bewezen worden door aan te tonen dat $a_1 R a_2 \wedge a_2 R a_1 \implies a_1 = a_2$ geldt. Dit wordt hieronder aangetoond:

$$
\begin{array}{ll}
 & a_1 R a_2 \wedge a_2 R a_1 \\
\text{(def. } R\text{)} & \implies a_1 (R_1 \cup R_2) a_2 \wedge a_2 (R_1 \cup R_2) a_1 \\
\text{(def. } \cup\text{)} & \implies (a_1 R_1 a_2 \vee a_1 R_2 a_2) \wedge (a_2 R_1 a_1 \vee a_2 R_2 a_1) \\
\text{(distr. } \wedge\text{)} & \implies (a_1 R_1 a_2 \wedge a_2 R_1 a_1) \vee (a_1 R_1 a_2 \wedge a_2 R_2 a_1) \\
 & \qquad \vee (a_1 R_2 a_2 \wedge a_2 R_1 a_1) \vee (a_1 R_2 a_2 \wedge a_2 R_2 a_1) \\
(R_1, R_2 \text{ conflictvrij}) & \implies a_1 = a_2 \vee a_1 = a_2 \vee a_1 = a_2 \vee a_1 = a_2 \\
\text{(eig. } \vee\text{)} & \implies a_1 = a_2
\end{array}
$$

**Lemma A.2.6** *$R^n$ is anti-symmetrisch and reflexief voor elke $n \in \mathbb{N}$.*

**Bewijs**

Opnieuw tonen we anti-symmetrie en reflexiviteit samen aan door te bewijzen dat $\forall\, n \in \mathbb{N} :\ a_1 R^n a_4 \wedge a_4 R^n a_1 \Longrightarrow a_1 = a_4$ geldt:

$$
\begin{array}{ll}
 & a_1 R^n a_4 \wedge a_4 R^n a_1 \\
\text{(lemma A.2.4)} & \Longrightarrow \exists\, a_2, a_3, a_5, a_6 \in A : \\
 & \qquad a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_4 \\
 & \qquad \wedge a_4 R_2 a_5 \wedge a_5 R_1 a_6 \wedge a_6 R_2 a_1 \\
\text{(trans. } R_2\text{)} & \Longrightarrow \exists\, a_2, a_3, a_5, a_6 \in A : \\
 & \qquad a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_4 \\
 & \qquad \wedge a_4 R_2 a_5 \wedge a_5 R_1 a_6 \wedge a_6 R_2 a_1 \\
 & \qquad \wedge\, a_3 R_2 a_5 \wedge a_6 R_2 a_2 \\
\text{(def. ; en } R\text{)} & \Longrightarrow \exists\, a_2, a_3, a_5, a_6 \in A : \\
 & \qquad a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_4 \\
 & \qquad \wedge a_4 R_2 a_5 \wedge a_5 R_1 a_6 \wedge a_6 R_2 a_1 \\
 & \qquad \wedge\, a_3 R_2 a_5 \wedge a_6 R_2 a_2 \\
 & \qquad \wedge a_2 R a_5 \wedge a_5 R a_2 \\
\text{(lemma A.2.5)} & \Longrightarrow \exists\, a_2, a_3, a_5, a_6 \in A : \\
 & \qquad a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_4 \\
 & \qquad \wedge a_4 R_2 a_2 \wedge a_2 R_1 a_6 \wedge a_6 R_2 a_1 \\
 & \qquad \wedge\, a_3 R_2 a_2 \wedge a_6 R_2 a_2 \wedge a_2 = a_5 \\
(R_1, R_2 \text{ conflictvrij} & \Longrightarrow \exists\, a_2, a_3, a_5, a_6 \in A : \\
\text{voor } a_2, a_3 \text{ en } a_2, a_6) & \qquad a_1 R_2 a_2 \wedge a_2 R_2 a_4 \wedge a_4 R_2 a_2 \wedge a_2 R_2 a_1 \\
 & \qquad \wedge\, a_2 = a_3 = a_5 = a_6 \\
\text{(anti-symm. } R_2\text{)} & \Longrightarrow \exists\, a_2, a_3, a_5, a_6 \in A : \\
 & \qquad a_2 = a_3 = a_5 = a_6 = a_1 = a_4 \\
\text{(eig. } \exists\text{)} & \Longrightarrow a_1 = a_4
\end{array}
$$

**Lemma A.2.7** *$R^\star$ is een partiële ordening.*

**Bewijs**

Uit lemma A.2.6 volgt dat $R^\star$ anti-symmetrisch is. $R^\star$ is per definitie reflexief en transitief. Er volgt dus dat $R^\star$ een partiële ordening is.

**Lemma A.2.8** *Als $R_1$ een p.o. is in $A_1^2$, en $R_2$ is een p.o. in $A_1 \times A_2$, $A_1 \cap A_2 = \{\}$, $R = (R_1 \cup R_2)^\star$, $a_1 \in A_1$, $a_3 \in A_2$ en $a_1 R a_3$ geldt, dan volgt er dat $\exists\, a_2 \in A_1 :\ a_1 R a_2 \wedge a_2 R_2 a_3$.*

**Bewijs**

Met $R_3 = R_1 \cup R_2$ volgt uit de definitie van transitieve sluiting dat $R = R_3^0 \cup R_3^1 \cup R_3^2 \cup \ldots$. Als $a_1 R a_3$ geldt, dan volgt er dat $a_1 = a_3 \vee$

$(a_1R_1a_3 \vee a_1R_2a_3) \vee \exists\, a_2 \in A : (a_1R_1a_2 \vee a_1R_2a_2) \wedge (a_2R_1a_3 \vee a_2R_2a_3) \vee \ldots$, of ook dat $a_1R_2a_3 \vee \exists\, a_2 \in A :\ a_1R_1a_2 \wedge a_2R_2a_3 \vee \exists\, a_2, a_4 \in A :\ a_1R_2a_2 \wedge a_2R_1a_4 \wedge a_4R_2a_3 \vee \ldots$. Daar $R_2; R_1 = \{\}$ volgt er dus dat $\exists\, a_2 \in A_1 : a_1Ra_2 \wedge a_2R_2a_3$.

**Lemma A.2.9** *Als relatie $R_1$ een partiële ordening is in $A$ en totaal in $B \subset A$, en als $R_2$ een relatie is in $(A \setminus B) \times B$ zodanig dat $\forall\, (a_1, a_2) \in A : a_1R_2a_2 \Longrightarrow \neg a_1R_1a_2 \wedge \neg a_2R_1a_1$, dan is $R \triangleq (R_1 \cup R_2)^\star$ een partiële ordening in $A$ en totaal in $A^2 \setminus (A \setminus B)^2$.*

**Bewijs**

In het bewijs worden volgende stappen aangetoond:

1. $(R_1; R_2) \subset R_2$.
2. $(R_2; R_1; R_2) \subset (R_2; R_1)$.
3. $(R_1; R_2) \subset (R_1 \cup R_2)$.
4. $\forall\, n \in \mathbb{N} :\ n \geq 1 \Longrightarrow (R_1 \cup R_2)^n \subset (R_1 \cup (R_2; R_1))$.
5. $R = R_1 \cup (R_2; R_1)$.
6. $R$ is totaal in $A^2 \setminus (A \setminus B)^2$.
7. $R$ is reflexief.
8. $R$ is anti-symmetrisch.
9. $R$ is transitief.

Het eigenlijke bewijs is als volgt:

1. Met $a_1$ en $a_3 \in A$ is het bewijs van $(R_1; R_2) \subset R_2$ is als volgt:

$$
\begin{array}{l}
a_1(R_1; R_2)a_3 \\
\Longrightarrow \exists\, a_2 \in A :\ a_1R_1a_2 \wedge a_2R_2a_3 \\
\Longrightarrow \exists\, a_2 \in A :\ a_1R_1a_2 \wedge a_2 \notin B \wedge a_3 \in B \wedge \neg a_2R_1a_3 \wedge \neg a_3R_1a_2 \\
\Longrightarrow \exists\, a_2 \in A :\ a_1R_1a_2 \wedge a_2 \notin B \wedge a_3 \in B \wedge \neg a_2R_1a_3 \wedge \neg a_3R_1a_2 \\
\wedge (a_1 \notin B \vee a_1R_1a_3 \vee a_3R_1a_1) \\
\Longrightarrow \exists\, a_2 \in A :\ a_1R_1a_2 \wedge a_2 \notin B \wedge a_3 \in B \\
\wedge \neg a_2R_1a_3 \wedge \neg a_3R_1a_2 \wedge a_1 \notin B \\
\Longrightarrow \exists\, a_2 \in A :\ a_1R_1a_2 \wedge a_2 \notin B \wedge a_3 \in B \wedge \neg a_2R_1a_3 \wedge \neg a_3R_1a_2 \\
\wedge a_1 \notin B \wedge a_1R_2a_3 \\
\Longrightarrow a_1R_2a_3
\end{array}
$$

2. Het bewijs van $(R_2; R_1; R_2) \subset (R_2; R_1)$ is als volgt:

$$
\begin{array}{l}
a_1(R_2; R_1; R_2)a_4 \\
\Longrightarrow \exists\, a_2, a_3 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_4 \\
\Longrightarrow \exists\, a_2, a_3 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_3 R_2 a_4 \wedge (a_2 R_1 a_4 \vee a_4 R_1 a_2) \\
\Longrightarrow \exists\, a_2 \in A :\ a_1 R_2 a_2 \wedge a_2 R_1 a_4 \\
\Longrightarrow a_1(R_2; R_1)a_4
\end{array}
$$

3. Eigenschap $(R_1; R_2) \subset (R_1 \cup R_2)$ geldt omdat:

$$
\begin{array}{l}
a_1(R_1; R_2)a_3 \\
\Longrightarrow \exists\, a_2 \in A :\ a_1 R_1 a_2 \wedge a_2 R_2 a_3 \\
\Longrightarrow \exists\, a_2 \in A :\ a_1 R_1 a_2 \wedge a_2 R_2 a_3 \\
\wedge (a_1 \notin B \wedge \neg a_1 R_1 a_3 \wedge \neg a_3 R_1 a_1 \vee a_1 R_1 a_3 \vee a_3 R_1 a_1) \\
\Longrightarrow a_1 R_1 a_3 \vee a_1 R_2 a_3
\end{array}
$$

4. De eigenschap $\forall\, n \in \mathbb{N} :\ n \geq 1 \Longrightarrow (R_1 \cup R_2)^n \subset R_1 \cup (R_2; R_1)$ wordt aangetoond via inductie:

   • Inductiebasis
   $(R_1 \cup R_2)^1 = R_1 \cup R_2 = R_1 \cup R_2; R_1$
   • Inductiehypothese
   $(R_1 \cup R_2)^n \subset R_1 \cup (R_2; R_1) \Longrightarrow (R_1 \cup R_2)^{n+1} \subset R_1 \cup (R_2; R_1)$
   • Bewijs

$$
\begin{array}{l}
(R_1 \cup R_2)^{n+1} \\
= (R_1 \cup R_2)^n; (R_1 \cup R_2) \\
\subset (R_1 \cup (R_2; R_1)); (R_1 \cup R_2) \\
\subset (R_1; R_1) \cup (R_2; R_1; R_1) \\
\cup (R_1; R_2) \cup (R_2; R_1; R_2) \\
\subset R_1 \cup (R_2; R_1) \cup (R_1 \cup R_2) \cup (R_2; R_1) \\
\subset R_1 \cup (R_2; R_1)
\end{array}
$$

   □

5. Uit de voorgaande stap en uit de definitie van transitieve sluiting volgt er dat $R \subset R_1 \cup (R_2; R_1)$. Daar tevens $R_1 \cup (R_2; R_1) \subset (R_1 \cup R_2)^2 \subset R$, volgt er dat $R = R_1 \cup (R_2; R_1)$.

6. Voor elke $a_1, a_3 \in A$ geldt er:

$$
\begin{array}{l}
a_1 R a_3 \vee a_3 R a_1 \\
\Longrightarrow (a_1 R_1 a_3 \vee \exists\, a_2 \in B :\ a_1 R_2 a_2 \wedge a_2 R_1 a_3) \\
\vee (a_3 R_1 a_1 \vee \exists\, a_4 \in B :\ a_3 R_2 a_4 \wedge a_4 R_1 a_1) \\
\Longrightarrow \exists\, a_2, a_4 \in B :\ a_1 R_1 a_3 \vee a_3 R_1 a_1 \\
\vee (a_3 R_2 a_4 \wedge a_4 R_1 a_1) \vee (a_1 R_2 a_2 \wedge a_2 R_1 a_3) \\
\Longrightarrow \exists\, a_2, a_4 \in B :\ (a_1, a_3) \in B^2 a_1 R_1 a_3 \vee a_3 R_1 a_1 \\
\vee ((a_3 R_2 a_4 \wedge a_4 R_1 a_1) \vee (a_1 R_2 a_2 \wedge a_2 R_1 a_3)) \\
\wedge (a_1 R_2 a_3 \vee a_3 R_2 a_1 \vee (a_1, a_3) \in (A \setminus B)^2) \\
\Longrightarrow \exists\, a_2, a_4 \in B :\ (a_1, a_3) \in B^2 a_1 R_1 a_3 \vee a_3 R_1 a_1 \\
\vee (a_3 R_2 a_4 \wedge a_4 R_1 a_1 \wedge a_3 R_2 a_1) \vee (a_1 R_2 a_2 \wedge a_2 R_1 a_3 \wedge a_1 R_2 a_3) \\
\Longrightarrow (a_1, a_3) \in B^2 \vee a_1 R_1 a_3 \vee a_3 R_1 a_1 \\
\vee a_3 R_2 a_1 \vee a_1 R_2 a_3 \\
\Longrightarrow (a_1, a_3) \in A^2 \setminus (A \setminus B)^2
\end{array}
$$

   Dit bewijst dat relatie $R$ totaal is in $A^2 \setminus (A \setminus B)^2$.

7. $R$ is per definitie reflexief.

8. $R$ is anti-symmetrisch:

$$
\begin{array}{l}
a_1 R a_3 \wedge a_3 R a_1 \\
\Longrightarrow \exists\, a_2, a_4 \in A :\ (a_1 R_1 a_3 \vee a_1 R_2 a_2 \wedge a_2 R_2 a_3) \\
\wedge (a_3 R_1 a_1 \vee a_3 R_2 a_4 \wedge a_4 R_1 a_1) \\
\Longrightarrow \exists\, a_2, a_4 \in A : \\
(a_1 R_1 a_3 \wedge a_3 R_2 a_4 \wedge a_4 R_1 a_1) \vee (a_1 R_2 a_2 \wedge a_2 R_2 a_3 \wedge a_3 R_1 a_1) \\
\Longrightarrow \textit{false}.
\end{array}
$$

9. $R$ is per definitie transitief.

Besluit: $(R_1 \cup R_2)^\star$ is een partiële ordening in $A$, en totaal in $A^2 \setminus (A \setminus B)^2$.

**Lemma A.2.10** *Als $A$ een relatie is, $B$ en $C$ verzamelingen zijn, $R_1$ een totale ordening is in $A = B^2 \setminus C^2$, $R_1 \subset R_2$ en $R_2$ een partiële ordening is, dan is $R_1 \cap A = R_2 \cap A$.*

**Bewijs**

Uit $R_1 \subset R_2$ volgt dat $(R_1 \cap A) \subset (R_2 \cap A)$. Stel dat $(R_2 \cap A) \subset (R_1 \cap A)$ niet zou gelden. Dan bestaat er minstens een koppel $(a_1, a_2) \in A$ waarvoor $a_1 R_2 a_2$ geldt en $a_1 R_1 a_2$ niet geldt. Daar $R_1$ totaal is in $A$ volgt $a_2 R_1 a_1$, en wegens het gegeven ook dat $a_2 R_2 a_1$. Dit is echter strijdig met de anti-symmetrie van $R_2$, dus $R_2 \cap A \subset R_1 \cap A$ en ook $R_1 \cap A = R_2 \cap A$. $\square$

**Lemma A.2.11** *Als relaties $R_1$, $R_2$ en $R_3$ partiële ordeningen zijn, met $R_1 \subset R_3$ en ook $R_2 \subset R_3$, dan volgt dat ook $R \stackrel{\triangle}{=} (R_1 \cup R_2)^\star$ een partiële ordening is.*

**Bewijs**

Relatie $R$ is per definitie reflexief en transitief. Omdat $R_1 \cup R_2 \subset R_3$, volgt er dat $R = (R_1 \cup R_2)^\star \subset R_3^\star$. Omdat $R_3$ een partiële ordening is, is $R_3 = R_3^\star$. Daar $R_3$ anti-symmetrisch is, is $R$ dat ook. $R$ is dus een partiële ordening.

# Bijlage B

# Bewijzen eigenschappen geheugenmodellen

## B.1 Equivalentie van PSO en PSO$^\star$

Voor het PSO-geheugenmodel gelden de volgende restricties: $Op \subset L_{ord} \cup S_{ord} \cup F_{ord} \cup Bar$, $\forall\, op \in Bar : \; mem(op) = Mem$ en $\forall\, b_1, b_2 \in Bar : b_1 N b_2 \implies b_1 = b_2$. De onderstaande bewijzen gelden voor de geheugenmodellen PSO versus PSO$^\star$ met bovenstaande restricties in acht genomen. De equivalentie van deze geheugenmodellen wordt bewezen door wederzijdse inclusie aan te tonen.

### B.1.1 Inclusie van PSO in PSO$^\star$

**Lemma B.1.1** *Elke PSO-uitvoering met opdrachtenverzameling Op, programmaordening* `;` *en SPARC-geheugenordening* $\leq$ *voldoet aan het gegeheugenmodel PSO$^\star$. Dit kan ook als volgt geformuleerd worden: er bestaan relaties* $\xrightarrow{mo\,1} \ldots \xrightarrow{mo\,n}$ *zodanig dat de uitvoering* $E' = (Op, \leq, \xrightarrow{mo\,1} \ldots \xrightarrow{mo\,n})$ *voldoet aan de eigenschappen (b), (c), (m), (3.3), (3.8), (3.9) en (r). Via de eigenschappen in tabel 3.19 volgt dat ook de eigenschappen (f), (g), (i), (j) en (o) gelden.*

**Bewijs**

We vertrekken van een uitvoering $E$ met opdrachten $Op$, programmaordening `;` en SPARC-geheugenordening $\leq$, geldig in het PSO-geheugenmodel. De relatie `;` is per definitie een geldige programmaordening, dus definiëren we $\xrightarrow{\text{po}} \triangleq$ `;`.

Beschouw nu volgende definitie voor de permutatie $T_m : Op_m \rightarrow Op_m$, met $op_1$ en $op_2 \in Op_m$:

- Als er geldt dat $op_1 \in S_m \wedge \exists\, op_2 \in L_m : \ (op_1 ;^{-} op_2 \wedge op_2 \leq op_1)$, dan is $T_m(op_1) = op_2$;
- Als er geldt dat $op_1 \in L_m \wedge \exists\, op_2 \in S_m : \ (op_2 ;^{-} op_1 \wedge op_1 \leq op_2)$, dan is $T_m(op_1) = op_2$;
- Anders is $T_m(op_1) = op_1$.

De bovenstaande definitie is zinvol omdat geen van de drie gevallen overlapt met een ander geval. $T_m$ is een permutatie omdat uit $T_m(op_1) = op_2$ volgt dat $T_m(op_2) = op_1$. De permutatie $T$ definiëren we als de unie van de afzonderlijke $T_m$-permutaties: $\forall\, op \in LSF : \ T(op) = T_{memsmall_{(op)}}(op) \wedge \forall\, op \in Op \setminus LSF : \ T(op) = op$. Merk op dat voor het gegeven geheugenmodel geldt dat $\#mem(op) = 1$. Verder definiëren we volgende relaties, met $m \in Mem$ en $p \in P$:

$$
\begin{aligned}
R_1 &\triangleq \{(l, s) \in (L \times SF) | \neg l \leq s \wedge \neg s \leq l\} \\
R_2 &\triangleq (\ \leq \ \cup R_1)^{\star} \\
R_3 &\triangleq \text{linext}^{N}_{Op}\, R_2 \\
R_4 &\triangleq T^{-1}(R_3)
\end{aligned}
$$

De verschillende stappen van het bewijs staan hieronder.

1. Daar relatie $\leq$ wegens het PSO-axioma Geheugenordening een totale ordening is in de verzameling $SF$ en wegens de definitie van $R_1$ is lemma A.2.9 van toepassing op de relaties $\leq$ en $R_1$. Er volgt er dat relatie $R_2$ een totale ordening is in $LSF^2 \setminus L^2$, dat $\leq \ \subset R_2$ en ook dat $R_2 = (\ \leq \ \cup (R_1;\ \leq))$.
2. Gevolg van de voorgaande stap: $R_2 \cap (SF \times L) = \ \leq \ \cap (SF \times L)$, wegens:

$$
\begin{aligned}
& R_2 \cap (SF \times L) \\
&= (\ \leq \ \cup (R_1;\ \leq)) \cap (SF \times L) \\
&= (\ \leq \ \cap (SF \times L)) \cup ((R_1;\ \leq) \cap (SF \times L)) \\
&= \ \leq \ \cap (SF \times L).
\end{aligned}
$$

3. Uit de definitie van relatie $R_3$ volgt dat $R_3/N$ een totale ordening is in $Op/N$, en dat $R_2 \subset R_3$.

4. Uit de toepassing van lemma A.2.10 op relaties $R_2$ en $R_3$ volgt dat $(R_2 \cap (LSF^2 \setminus L^2)) = (R_3 \cap (LSF^2 \setminus L^2))$. Daar $(SF \times L) \subset (LSF^2 \setminus L^2)$ en uit stap 2 volgt verder dat $R_3 \cap (SF \times L) = \leq \cap (SF \times L)$.

5. Daar wegens PSO-axioma Schrijf-Schrijf de relatie $\leq$ een totale ordening is in $SF^2$, en daar $\leq \subset R_2 \subset R_3$, volgt er wegens lemma A.2.10 dat $(\leq \cap SF^2) = (R_3 \cap SF^2)$.

6. Uit stappen 4 en 5 volgt dat $R_3 \cap (SF^2 \cup (SF \times L)) = \leq \cap (SF^2 \cup (SF \times L))$.

7. Door toepassing van de PSO-axioma's Lees-Opdracht en Schrijf-Schrijf-Zelfde en via de definitie van permutatie $T$ volgt er voor $op_1$ en $op_2 \in Op_m$:

$$
\begin{aligned}
& op_1 \,;\, op_2 \wedge T(op_2) R_3 T(op_1) \wedge (op_1, op_2) \notin (S_m \times L_m) \\
& \Longrightarrow op_1 \,;\, op_2 \wedge T(op_2) R_3 T(op_1) \\
& \wedge (T(op_1) \,;\, op_1 \wedge op_1 \in L \vee T(op_1) = op_1 \vee T(op_1) = op_2) \\
& \wedge (op_2 \,;\, T(op_2) \wedge op_2 \in S \vee T(op_2) = op_2 \vee T(op_2) = op_1) \\
& \wedge (op_1, op_2) \notin (S_m \times L_m) \\
& \Longrightarrow op_1 \,;\, op_2 \wedge T(op_2) R_3 T(op_1) \\
& \wedge (T(op_1) \,;\, op_1 \wedge op_1 \in L_m \vee T(op_1) = op_1) \\
& \wedge (op_2 \,;\, T(op_2) \wedge op_2 \in S_m \vee T(op_2) = op_2) \\
& \wedge (op_1, op_2) \notin (S_m \times L_m) \\
& \Longrightarrow op_1 \,;\, op_2 \wedge T(op_2) R_3 T(op_1) \\
& \wedge (T(op_1) \,;\, T(op_2)) \\
& \wedge (op_1, op_2) \notin (S_m \times L_m) \\
& \Longrightarrow op_1 \,;\, op_2 \wedge T(op_2) R_3 T(op_1) \wedge T(op_1) \leq T(op_2) \\
& \Longrightarrow op_1 \,;\, op_2 \wedge T(op_2) R_3 T(op_1) \wedge T(op_1) R_3 T(op_2) \\
& \Longrightarrow op_1 = op_2.
\end{aligned}
$$

8. Uit stap 7 volgt dat relaties `;` en $R_4$ conflictvrij zijn in $Op_m$:

$$
\begin{aligned}
& op_1 \,;\, op_2 \wedge op_2 R_4 op_1 \\
& \Longrightarrow op_1 \,;\, op_2 \wedge T(op_2) R_3 T(op_1) \\
& \Longrightarrow op_1 \,;\, op_2 \wedge T(op_2) R_3 T(op_1) \\
& \wedge((op_1, op_2) \in (S_m \times L_m) \\
& \wedge T(op_1) = op_2 \wedge T(op_2) = op_1 \wedge op_2 \leq op_1 \\
& \vee(op_1, op_2) \in (S_m \times L_m) \wedge T(op_1) = op_1 \wedge T(op_2) = op_2 \\
& \vee(op_1, op_2) \notin (S_m \times L_m)) \\
& \Longrightarrow op_1 \,;\, op_2 \wedge op_2 \leq op_1 \\
& \vee op_1 \,;\, op_2 \wedge op_2 R_3 op_1 \wedge (op_1, op_2) \notin (S_m \times L_m) \\
& \Longrightarrow op_1 \,;\, op_2 \wedge op_2 \leq op_1 \\
& \vee op_2 R_3 op_1 \wedge op_1 \leq op_2 \\
& \Longrightarrow op_1 = op_2.
\end{aligned}
$$

9. Uit de definitie van de permutatie $T$ volgt rechtstreeks dat $R_4 \cap SF^2 = \leq \cap SF^2$.

10. Stel dat $(op_1, op_2) \in (SF_m \times L_{m,p})$ en dat $proc(op_1) = proc(op_2)$. Dan volgt uit het conflictvrij zijn van `;` en $R_4$ en ook uit het totaal zijn van `;` voor de opdrachten per processor dat $op_1 R_4 op_2 \Longrightarrow op_1 \,;\, op_2$. Verder geldt dat:

$$
\begin{aligned}
& op_1 \,;\, op_2 \\
& \Longrightarrow op_1 \leq op_2 \vee op_2 \leq op_1 \vee (\neg op_1 \leq op_2 \wedge \neg op_2 \leq op_1) \\
& \Longrightarrow op_1 R_4 op_2 \vee op_1 R_1 op_2 \wedge op_2 R_1 op_1 \\
& \Longrightarrow op_1 R_4 op_2 \vee op_1 R_3 op_2 \wedge op_2 R_3 op_1 \\
& \Longrightarrow op_1 R_4 op_2 \vee op_1 R_4 op_2 \\
& \Longrightarrow op_1 R_4 op_2.
\end{aligned}
$$

Gevolg:

$$
\begin{aligned}
& ((op_1, op_2) \in (SF_m \times L_m) \wedge proc(op_1) = proc(op_2)) \\
& \Longrightarrow (op_1 R_4 op_2 \Longleftrightarrow op_1 \,;\, op_2)\,.
\end{aligned}
$$

11. Uit Lees-Opdracht, Schrijf-Schrijf-Zelfde en stap 10 volgt dat $(\,; \cap LSF_m^2) \subset R_4$.

12. Met $(op_1, op_2) \in (LF \times Op)$ en $op_1 \,;^- op_2$ geldt er dat:

$$\begin{array}{l}
\exists\, m \in Mem :\ (op_1, op_2) \in (S_m \times L_m) \\
\vee \forall\, m \in Mem :\ (op_1, op_2) \notin (S_m \times L_m) \\
\Longrightarrow \exists\, m \in Mem :\ (op_1, op_2) \in (S_m \times L_m) \\
\wedge (op_1 \leq op_2 \wedge T(op_1) = op_1 \\
\vee op_2 \leq op_1 \wedge T(op_1) = op_2 \wedge T(op_2) = op_1) \\
\vee T(op_1) \,;\, T(op_2) \\
\Longrightarrow (op_1, op_2) \in (S_m \times L_m) \\
\wedge (T(op_1) \leq T(op_2) \vee T(op_1) \leq T(Op_2)) \\
\vee T(op_1) \leq T(op_2) \\
\Longrightarrow T(op_1) \leq T(op_2) \\
\Longrightarrow op_1 T^{-1}(\leq) op_2.
\end{array}$$

    Via inductie volgt er dat $op_1 \,;\, op_2 \Longrightarrow op_1 T^{-1}(\leq) op_2$, en ook dat $(\,; \cap (LF \times Op)) \subset (T^{-1}(\leq) \cap (LF \times Op)) \subset (R_4 \cap (LF \times Op))$.

13. Met $(op_1, op_2) \in (SF_m \times L_{m,p})$ en $proc(op_1) \neq proc(op_2)$ volgt uit de definitie van $T$ dat $T(op_1) = op_1$ en $T(op_2) = op_2$. Gevolg:

$$\begin{array}{l}
((op_1, op_2) \in (SF_m \times L_{m,p}) \wedge proc(op_1) \neq proc(op_2)) \\
\Longrightarrow (op_1 R_4 op_2 \Longleftrightarrow op_1 \leq op_2)\,.
\end{array}$$

14. Definieer $A_{1,m} \triangleq SF_m \times L_m$, $A_{2,m} \triangleq A_{1,m} \cap \bigcup_{p \in P} Op_{m,p}$ en $A_{3,m} \triangleq A_{1,m} \setminus A_{2,m}$, waarbij dus $A_{1,m} = A_{2,m} \cup A_{3,m}$.

15. Wegens stappen 9, 10 en 13 geldt:

$$\begin{array}{l}
R_4 \cap (SF_m^2 \cup A_{1,m}) \\
= R_4 \cap (SF_m^2 \cup A_{2,m} \cup A_{3,m}) \\
= (R_4 \cap SF_m^2) \cup (R_4 \cap A_{2,m}) \cup (R_4 \cap A_{3,m}) \\
= (\leq \cap SF_m^2) \cup (\,; \cap A_{2,m}) \cup (\leq \cap A_{3,m}).
\end{array}$$

16. Met $m \in Mem$, $l \in LF_m$ en $p = proc(l)$ volgt door toepassen van PSO-axioma Lees-Opdracht uit de definitie van $pred_{SPARC,\leq}(l)$ dat $pred_{SPARC,\leq}(l) = \{s \in SF_{mem(l)} | s \neq l \wedge (s \,;\, l \vee s \leq l \wedge proc(s) \neq proc(l))\}$. In combinatie met het PSO-axioma Waarde volgt er dat de waarde van de leesopdrachten in het PSO-geheugenmodel volledig bepaald wordt door de relatie $(\leq \cap SF_m^2) \cup (\,; \cap A_{2,m}) \cup (\leq \cap A_{3,m})$.

17. Definieer $E' = (Op, \xrightarrow{\text{po}}, R_4 \dots R_4)$. Uit stap 11 volgt dat (3.1) geldt voor $E'$. Omdat het PSO-geheugenmodel geen etiketten definieert en geen lock- of unlock-opdrachten kent, omdat een PSO-barrière steeds bestaat uit één grensopdracht en omwille van PSO-axioma Schrijf-Schrijf-Zelfde gelden (3.2), (3.5), (3.7), (3.4) en (3.6). Uit stappen 15 en 16 volgt dat $E'$ voldoet aan (3.3). Eigenschap (3.8) geldt omwille van PSO-axioma Beëindiging. (3.9) volgt uit het gegeven dat alle geheugenordening gelijk zijn en $R_4$ een totale ordening is. Uit stap 12 volgt dat eigenschap (m) geldt.

### B.1.2 Inclusie van PSO$^\star$ in PSO

Verder wordt bewezen dat elke TSO$^\star$-uitvoering voldoet aan het TSO-geheugenmodel. Laten we uit het gegeven van dat bewijs de eigenschap (i) weg, dan is het resultaat dat voor de bekomen SPARC-uitvoering $E'$ bestaande uit *Op*, **;** en $\leq$ dat alle eigenschappen behalve Schrijf-Schrijf gelden. Eigenschap Schrijf-Schrijf-Zelfde volgt uit (3.1). $E'$ voldoet dus aan PSO. Gevolg: elke PSO$^\star$-uitvoering voldoet aan het geheugenmodel PSO.

## B.2 Equivalentie van TSO en TSO$^\star$

In deze paragraaf wordt de equivalentie bewezen van TSO en TSO$^\star$ onder de voorwaarden die van toepassing zijn voor het geheugenmodel TSO. Deze voorwaarden zijn: $Op \subset L_{ord} \cup S_{ord} \cup F_{ord}$, $op \in LSF \implies \#mem(op) = 1$, $op \in Bar_{ss} \implies mem(op) = Mem$ en $\forall\, b_1, b_2 \in Bar : b_1 N b_2 \implies b_1 = b_2$. We bewijzen de equivalentie door wederzijdse inclusie aan te tonen.

### B.2.1 Inclusie van TSO in TSO$^\star$

**Lemma B.2.1** *Elke TSO-uitvoering met opdrachtenverzameling Op, programmaordening* **;** *en SPARC-geheugenordening* $\leq$ *voldoet aan het gegeheugenmodel TSO$^\star$. Dit kan ook geformuleerd worden als er bestaan relaties* $\xrightarrow{mo\,1} \dots \xrightarrow{mo\,n}$ *zodanig dat de uitvoering* $E' = (Op, \leq, \xrightarrow{mo\,1} \dots \xrightarrow{mo\,n})$ *voldoet aan de eigenschappen (b), (c), (m), (l), (3.3), (3.8), (3.9) en (r). Als direct gevolg van deze laatste eigenschappen gelden ook de eigenschappen (f), (g), (i), (j), (o) en (q)– zie ook tabel 3.19.*

**Bewijs**

Daar $Op$, ; en $\leq$ samen een geldige TSO-uitvoering vormen, vormen deze ook een geldige PSO-uitvoering. Met $R_i$ dezelfde relatie als in lemma B.1.1 volgt er dus dat $E' = (Op, ;, R_i)$ een geldige PSO$^\star$ uitvoering is. Omdat het TSO-axioma Schrijf-Schrijf geldt, volgt er dat ook (l) geldt. $E'$ voldoet dus aan het geheugenmodel TSO$^\star$.

### B.2.2 Inclusie van TSO$^\star$ in TSO

Uitgaande van de TSO$^\star$-uitvoering $E = (Op, \xrightarrow{\text{po}}, \xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n})$, waarbij $\xrightarrow{\text{mo}\,1} = \ldots = \xrightarrow{\text{mo}\,n} = \xrightarrow{\text{mo}}$, definiëren we volgende relaties:

$$
\begin{aligned}
R_{1,m} &\triangleq \xrightarrow{\text{po}} \cap\, Op_m^2 \\
R_{2,m} &\triangleq \xrightarrow{\text{mo}} \cap\, Op_m^2 \\
R_3 &\triangleq (\textstyle\bigcup_{m \in Mem} R_{2,m})^\star \\
R_4 &\triangleq \xrightarrow{\text{po}} \cap\, (LF \times Op) \\
R_5 &\triangleq \xrightarrow{\text{po}} \cap\, SF^2 \\
R_6 &\triangleq (R_4 \cup R_3)^\star \\
R_7 &\triangleq (R_6 \cup R_5)^\star \\
R_8 &\triangleq \xrightarrow{\text{mo}} \cap\, SF^2 \\
R_9 &\triangleq (R_8 \cup R_7)^\star
\end{aligned}
$$

Daar de uitvoering $E = (Op, \xrightarrow{\text{po}}, \xrightarrow{\text{mo}})$ een geldige TSO$^\star$-uitvoering is, volgt uit definitie van TSO$^\star$ dat $R_{1,m} \subset R_{2,m}$, dat $R_{2,m}$ een totale ordening is in $Op_m$, dat $\forall\, p \in P :\ R_{2,m} \subset \xrightarrow{\text{mo}\,p}$, $\forall\, p \in P :\ R_4 \subset \xrightarrow{\text{mo}\,p}$ en dat $R_5 = R_8$ totale ordeningen zijn in $SF$.

Uit lemma A.2.1 volgt dat relatie $R_3, R_6, R_7$ en $R_8$ partiële ordeningen zijn en dat $R_3$, $R_6, R_7$ en $R_8$ deelrelaties zijn van $\xrightarrow{\text{mo}\,p}$. Relatie $R_{1,m}$ is per definitie een totale ordening in $Op_m$. Ook is $R_{1,m} \subset R_{2,m}$ en er geldt dat $R_{1,m} \cap Op_m^2 \subset R4, m \cap Op_m^2$. Daar ook $R_{2,m}$ een partiële ordening is in $Op_m$, volgt dat $(R_{1,m} \cap Op_m^2) = (R_{2,m} \cap Op_m^2)$.

Wegens de definitie van $R_4$ geldt dat $\forall\, p \in P :\ \forall\, m \in Mem :\ R_4 \subset \xrightarrow{\text{mo}\,p}$. In combinatie met $R_{2,m} \subset \xrightarrow{\text{mo}\,p}$ leidt dit tot $R_{5,Mem} \subset \xrightarrow{\text{mo}\,p}$. Verder volgt dat $R_6 \subset \xrightarrow{\text{mo}\,p}$ en dus ook dat $R_6$ een partiële ordening is in $Op$. Uit lemma A.2.7 volgt er dat $R_9$ een partiële ordening is in $Op$.

$$
\begin{aligned}
&(\xrightarrow{\text{po}} \cap A_{2,m}) \cup (R_9 \cap (SF_m^2 \cup A_{3,m})) \\
&= (R_{1,m} \cap A_{2,m}) \cup (R_9 \cap (SF_m^2 \cup A_{3,m})) \\
&= (R_{2,m} \cap A_{2,m}) \cup (R_9 \cap (SF_m^2 \cup A_{3,m})) \\
&= (R_9 \cap (SF_m^2 \cup A_{2,m} \cup A_{3,m})) \\
&= (R_9 \cap (SF_m^2 \cup A_{1,m}) \\
&= \xrightarrow{\text{mo}} \cap (SF_m^2 \cup A_{1,m})
\end{aligned}
$$

Er volgt dat $pred_{SPARC,R_9}(l)$ en $pred_{\xrightarrow{\text{mo}}}(l)$ dezelfde verzamelingen zijn, waarbij $l \in LF$.

Beschouw nu de SPARC-uitvoering met opdrachtenverzameling *Op*, programmaordening $; = \xrightarrow{\text{po}}$ en SPARC-geheugenordening $\leq = R_9$. Deze uitvoering is een geldige TSO-uitvoering omdat:

1. $;$ and $\leq$ per definitie partiële ordeningen zijn in verzameling *Op*.
2. programmaordening geldt wegens $; = \xrightarrow{\text{po}}$.
3. Geheugenordening geldt wegens $R_8 \subset R_9 = \leq$.
4. Ondeelbaarheid geldt wegens eigenschap 3.3.
5. Beëindiging geldt wegens eigenschap 3.8.
6. Waarde geldt wegens $(\xrightarrow{\text{po}} \cap A_{2,m}) \cup (R_9 \cap (SF_m^2 \cup A_{3,m})) = \xrightarrow{\text{mo}} \cap (SF_m^2 \cup A_{1,m})$.
7. Lees-Opdracht geldt omdat $R_4 \subset R_9 = \leq$.
8. Schrijf-Schrijf geldt omwille van eigenschap 3.6.
9. Schrijf-Schrijf-Zelfde geldt omdat $\forall\, m \in Mem :\ R_{2,m} \subset R_9 = \leq$.

## B.3 Bewijzen van eigenschappen uit paragraaf 3.9.2

Hieronder wordt een eigenschap bewezen die vermeld werd op blz. 74.

**Lemma B.3.1** *Onderstaande eigenschap geldt voor berichtencommunicatiemodellen:*

$$
\begin{aligned}
&\forall\, m \in Mem :\ \forall\, s_1, s_2 \in SF_m :\ \forall\, l_1, l_2 \in LF_m : \\
&s_1 \neq s_2 \wedge s_1\, SR\, l_1 \wedge s_2\, SR\, l_2 \\
&\Longrightarrow (s_1 \xrightarrow{mo} s_2 \wedge l_1 \xrightarrow{mo} l_2) \vee (s_2 \xrightarrow{mo} s_1 \wedge l_2 \xrightarrow{mo} l_1)
\end{aligned}
$$

**Bewijs**

Stel dat voor gegeven $m$, $s_1$ en $s_2$ en met $s_1$ en $s_2$ in $SF_m$ geldt dat $s_1 \xrightarrow{\text{mo}} s_2$. Dan volgt uit $s_1 \neq s_2$ in combinatie met het eerste gevolg van eigenschap (3.12) dat $C_m(s_1) < C_m(s_2)$. Uit het gegeven $s_1 \,\text{SR}\, l_1 \wedge s_2 \,\text{SR}\, l_2$ volgt er dat $C_m(s_1) = C_m(l_1) \wedge C_m(s_2) = C_m(l_2)$. Door deze uitdrukkingen te combineren volgt er dat $C_m(l_1) < C_m(l_2)$. Door toepassing van het tweede gevolg van eigenschap (3.12) volgt hieruit dat $l_1 \xrightarrow{\text{mo}} l_2$. Samengevat: uit het gegeven en $s_1 \xrightarrow{\text{mo}} s_2$ volgt dat $l_1 \xrightarrow{\text{mo}} l_2$. Analoog volgt uit $s_2 \xrightarrow{\text{mo}} s_1$ dat $l_2 \xrightarrow{\text{mo}} l_1$. Wegens eigenschap 3.11 volgt er dat $s_1 \xrightarrow{\text{mo}} s_2 \vee s_2 \xrightarrow{\text{mo}} s_1$. Hieruit besluiten we dat $(s_1 \xrightarrow{\text{mo}} s_2 \wedge l_1 \xrightarrow{\text{mo}} l_2) \vee (s_2 \xrightarrow{\text{mo}} s_1 \wedge l_2 \xrightarrow{\text{mo}} l_1)$.

Met het volgende lemma wordt een eigenschap uit tabel 3.19 op blz. 84 bewezen:

**Lemma B.3.2** *(t) ∧ (m) ⟹ (u)*

**Bewijs**

Gegeven: $m \in \mathit{Mem}, s_1, s_2 \in SF_m, l_1, l_2 \in LF_m$. De schrijfopdrachten in $SF_m$ zijn totaal geordend. Er zijn drie mogelijkheden: $C_m(s_1)$ is ofwel kleiner dan, ofwel gelijk aan, ofwel groter dan $C_m(s_2)$. Als $C_m(s_1) = C_m(s_2)$ volgt er dat $s_1 = s_2$. Met $s_1 = s_2$ en gebruik makend van (m) volgt er:

$$
\begin{aligned}
& s_1 \,\text{SR}\, l_1 \wedge s_2 \,\text{SR}\, l_2 \\
\Longrightarrow\ & s_1 (\xrightarrow{\text{mo}} \setminus L^2)^- l_1 \wedge s_2 (\xrightarrow{\text{mo}} \setminus L^2)^- l_2 \\
\Longrightarrow\ & s_1 \xrightarrow{\text{mo}} l_1 \wedge s_2 \xrightarrow{\text{mo}} l_2 \\
\Longrightarrow\ & s_1 \xrightarrow{\text{mo}} l_1 \wedge s_1 \xrightarrow{\text{mo}} l_2 \\
\Longrightarrow\ & \mathit{proc}(l_1) \neq \mathit{proc}(l_2) \\
\Longrightarrow\ & \neg(l_1 \xrightarrow{\text{po}} l_2) \wedge \neg(l_2 \xrightarrow{\text{po}} l_1) \\
\Longrightarrow\ & (u).
\end{aligned}
$$

Voor het geval $C_m(s_1) < C_m(s_2)$ volgt er uit de definities van eigenschap $(t)$ en relatie SR:

$$
\begin{aligned}
& s_1 \,\text{SR}\, l_1 \wedge s_2 \,\text{SR}\, l_2 \\
\Longrightarrow\ & s_1 \xrightarrow{\text{mo}} l_1 \wedge s_2 \xrightarrow{\text{mo}} l_2 \\
\Longrightarrow\ & s_1 \xrightarrow{\text{mo}} l_1 \xrightarrow{\text{mo}} s_2 \xrightarrow{\text{mo}} l_2 \\
\Longrightarrow\ & l_1 \xrightarrow{\text{mo}} l_2 \\
\Longrightarrow\ & \neg l_2 \xrightarrow{\text{mo}} l_1 \\
\Longrightarrow\ & \neg l_2 \xrightarrow{\text{po}} l_1 \\
\Longrightarrow\ & (u).
\end{aligned}
$$

Het geval $C_m(s_2) < C_m(s_1)$ is analoog.

## B.4 Bewijzen van eigenschappen uit paragraaf 3.10

**Lemma B.4.1** *Als een uitvoering $E = (Op, \overset{po}{\longrightarrow})$ voldoet aan de definiërende eigenschappen van de geheugenmodellen CC$^\star$ en CCA$^\star$, nl. (c), (e), (3.3), (3.8) en (r), dan voldoet dezelfde uitvoering ook aan de eigenschappen (f), (k), (l), (m), (n) en (o).*

**Bewijs**

De geheugenordeningen horende bij uitvoering $E$ duiden we aan met $\overset{\text{mo}\,1}{\longrightarrow}$ t.e.m. $\overset{\text{mo}\,n}{\longrightarrow}$. Wegens eigenschap (e) geldt ook eigenschap (n), met als gevolg dat $\overset{\text{po}\,p}{\longrightarrow} \in \overset{\text{mo}\,p}{\longrightarrow}$. De uitvoering $E$ voldoet aan volgende eigenschappen:

- Omwille van de definitie van CC$^\star$ en CCA$^\star$ geldt $\overset{\text{po}}{\longrightarrow} \subset \overset{\text{mo}\,p}{\longrightarrow}$, en dus ook eigenschap (k).
- Via tabel 3.19 op blz. 84 volgen eigenschappen (c), (l), (m), (n) en (o) onmiddellijk uit (f) en (k).

De uitvoering $E$ voldoet dus aan de gestelde eigenschappen.

**Lemma B.4.2** *De geheugenmodellen CC en CCA zijn equivalent voor uitvoeringen die een eindig aantal opdrachten bevatten.*

**Bewijs**

Dit volgt uit de combinatie van lemma's B.4.3 en B.4.4.

**Lemma B.4.3** *Als uitvoering $E = (Op, \overset{po}{\longrightarrow}, \overset{mo}{\longrightarrow})$ voldoet aan het geheugenmodel causale consistentie of CC, dan voldoet deze uitvoering ook aan causale consistentie zoals gedefinieerd door Ahamad.*

**Bewijs**

Met het geheugenmodel CC wordt het geheugenmodel bedoeld zoals gedefinieerd in tabel 3.5. Omdat uitvoering $E$ voldoet aan geheugenmodel CC gelden o.m. eigenschappen (c), (k), (3.3), (3.8) en (r). Beschouw nu onderstaande definities:

$$
\begin{aligned}
\mapsto &\triangleq \bigcup_{p\in P}\bigcup_{m\in Mem}(\overset{\text{mo}\,p}{\longrightarrow} \cap (SF_m \times LF_{m,p}))^- \\
\rightsquigarrow &\triangleq (\overset{\text{po}}{\longrightarrow} \cup \mapsto)^\star \\
Q_p &\triangleq \overset{\text{mo}\,p}{\longrightarrow} \cap (Op_p \cup SF)^2
\end{aligned}
$$

De uitvoering $(Op, \xrightarrow{\text{po}}, \mapsto, \rightsquigarrow, Q_1 \ldots Q_n)$ is causaal consistent volgens de definitie van Ahamad omdat:

1. De relatie $\mapsto$ ordent per definitie enkel paren van schrijf- en leesopdrachten.
2. Wegens eigenschap (c) is elke relatie $\xrightarrow{\text{mo}\,p}$ een totale ordening in $Op$. Als gevolg geldt dat $op_1 \mapsto op_3 \wedge op_2 \mapsto op_3 \Longrightarrow op_1 = op_2$.
3. Als $op_1 \mapsto op_2$ geldt, dan volgt uit (3.3) samen met het gegeven dat $\xrightarrow{\text{mo}\,p}$ een totale ordening is, dat $Val(op_1) = Val(op_2)$.
4. Uit de definitie van CC volgt rechtstreeks dat $\forall\, p \in P : \rightsquigarrow \subset \xrightarrow{\text{mo}\,p}$ en dus ook dat $\rightsquigarrow$ een p.o. is.
5. Uit de definitie van $Q_p$ volgt dat $\xrightarrow{\text{mo}\,p} \cap (SF^2 \cup (SF \times Op_p)) = Q_p \cap (SF^2 \cup (SF \times Op_p))$.
6. De ordening $Q_p$ is per definitie een totale ordening.
7. Uit stap 5 en uit het gevolg van eigenschap (3.3) volgt dat in $Q_p$ de waarde van een leesopdracht de waarde is geschreven door de meest recente schrijfopdracht, of eigenschap (3.3) geldt voor $Q_p$.
8. Wegens $\rightsquigarrow \subset \xrightarrow{\text{mo}\,p}$ volgt er dat $\rightsquigarrow \cap (Op_p \cup SF)^2 \subset Q_p$, en dus dat relatie $Q_p$ relatie $\rightsquigarrow$ respecteert.

Gevolg: de relatie $\mapsto$ voldoet aan Ahamads voorwaarden voor een dergelijke relatie, de relatie $\rightsquigarrow$ is op dezelfde wijze gedefinieerd als door Ahamad, en $Q_p$ respecteert $\rightsquigarrow$ en voldoet aan eigenschap (3.3). De uitvoering $(Op, \xrightarrow{\text{po}}, \mapsto, \rightsquigarrow, Q_1 \ldots Q_n)$ voldoet dus aan Ahamads definitie van een causaal consistente uitvoering.

**Lemma B.4.4** *Als uitvoering $E = (Op, \xrightarrow{po}, \mapsto, \rightsquigarrow, Q_1 \ldots Q_n)$ voldoet aan het geheugenmodel causale consistentie volgens Ahamad of CCA, en de opdrachtenverzameling Op is eindig, dan voldoet deze uitvoering ook aan causale consistentie of CC.*

**Bewijs**

Voor een gegeven uitvoering $E = (Op, \xrightarrow{\text{po}}, \mapsto, \rightsquigarrow, Q_1 \ldots Q_n)$ die voldoet aan causale consistentie volgens Ahamad, definiëren we de relaties $R_1$ t.e.m. $R_n$ als volgt, met $p \in P$:

$$R_p \triangleq \text{linext}^N_{Op}(Q_p \cup \xrightarrow{\text{po}})^\star.$$

Daar uitvoering $E$ voldoet aan CCA, geldt er dat relatie $\rightsquigarrow = (\mapsto \cup \xrightarrow{\text{po}})^\star$ een partiële ordening is. Omdat relatie $Q_p$ de relatie $\rightsquigarrow$ respecteert geldt dat $\rightsquigarrow \cap dom(Q_p)^2 \subset Q_p$, met $dom(Q_p) = Op_p \cup SF$.

1. Omdat $\xrightarrow{\text{po}} \cap dom(Q_p)^2 \subset Q_p$ geldt is lemma A.2.7 van toepassing is op relaties $Q_p$ en $\xrightarrow{\text{po}}$. Dus is de relatie $(Q_p \cup \xrightarrow{\text{po}})^\star$ een partiële ordening. Gevolg: elke relatie $R_p$ is een totale ordening in $Op/N$, of eigenschap (c) geldt.

2. Uit de definitie van $R_p$ volgt dat $Q_p \subset R_p$. Omdat $(SF \cup LF_p) \subset dom(Q_p)$, en omdat $Q_p$ een totale ordening is in $dom(Q_p)$, volgt er dat $R_p \cap (SF \cup LF_p)^2 = Q_p \cap (SF \cup LF_p)^2$. Gevolg: $R_p \cap (SF_m \times LF_{m,p}) = Q_p \cap (SF_m \times LF_{m,p})$.

3. Omdat $Q_p$ voldoet aan eigenschap (3.3) en ook omwille van stap 2 volgt er dat $Q_p \cap (SF \times LF_p) = \mapsto \cap (SF \times LF_p)$. Gevolg: $R_p \cap (SF_m \times LF_{m,p}) = \mapsto \cap (SF_m \times LF_{m,p})$.

4. Als gevolg van de vorige stap geldt er dat $\bigcup_{p \in P} \bigcup_{m \in Mem} R_p \cap (SF_m \times LF_{m,p}) = \mapsto$. Dit is eigenschap (e).

5. Uit stap 2 volgt dat $R_p \cap (SF_m^2 \cup (SF_m \times L_{m,p})) = Q_p \cap (SF_m^2 \cup (SF_m \times L_{m,p}))$. Omdat de relaties $Q_p$ voldoen aan eigenschap (3.3) en wegens het gevolg van eigenschap (3.3) geldt dat ook de relaties $R_p$ voldoen aan (3.3).

6. Eigenschap (3.8) geldt omdat de verzameling $Op$ eindig is.

7. Omdat uitvoering $E$ voldoet aan uniprocessorcorrectheid geldt deze eigenschap (3.1) ook voor uitvoering $E'$. Uitvoeringen $E$ en $E'$ bevatten geen speciale opdrachten of synchronisatieopdrachten, dus eigenschappen (3.2), (3.4), (3.5), (3.6) en (3.7) gelden voor $E'$. Dit heeft als gevolg dat eigenschap (r) geldt voor $E'$.

Besluit: de uitvoering $E' = (Op, \xrightarrow{\text{po}}, R_1 \ldots R_n)$ voldoet aan de eigenschappen (c), (e), (3.3), (3.8) en (r), en is dus een geldige uitvoering onder het geheugenmodel CC.

**Lemma B.4.5** *Voor elke uitvoering $E = (Op, \xrightarrow{po})$ die voldoet aan de definitie van PRAM$^\star$, waarvoor dus eigenschappen (c), (n), (3.3), (3.8) en (r) gelden, gelden ook de eigenschappen (c), (f), (n),(o), (3.3) en (3.8).*

**Bewijs**

Daar de uitvoering $E$ een PRAM$^\star$-uitvoering is, bestaan er geheugenordeningen $\xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n}$ zodanig dat eigenschappen (c), (n), (3.3) en (3.8) gelden. Uit de eigenschappen in tabel 3.19 volgt onmiddellijk het gestelde.

**Lemma B.4.6** *Als de uitvoering $E = (Op, \xrightarrow{po})$ een eindig aantal opdrachten bevat en voldoet aan de definiërende eigenschappen van het veralgemeende geheugenmodel PC$^\star$, nl. (j), (l), (n), (3.3), (3.8), (3.9) en (r), dan voldoet dezelfde uitvoering ook aan de eigenschap (c). Via tabel 3.19 volgt onmiddellijk dat dan ook de overige eigenschappen (f), (g), en (o) gelden.*

**Bewijs**

Omdat de uitvoering $E = (Op, \xrightarrow{\text{po}})$ voldoet aan het geheugenmodel PC$^\star$, bestaan er geheugenordeningen $\xrightarrow{\text{mo}\,1}$ t.e.m. $\xrightarrow{\text{mo}\,n}$ die voldoen aan de definiërende eigenschappen van PC$^\star$. Vertrekkende van de uitvoering $E = (Op, \xrightarrow{\text{po}}, \xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n})$ definiëren we volgende verzameling en relaties, met $p \in P$ en $m \in Mem$:

$$
\begin{aligned}
A_p &\triangleq SF \times (SF \cup L_p) \\
R_{1,m} &\triangleq \xrightarrow{\text{mo}\,1} \cap\, SF_m^2 \\
R_{2,p} &\triangleq \xrightarrow{\text{po}} \cap\, SF^2 \\
R_{3,p} &\triangleq \xrightarrow{\text{po}\,p} \cap\, LSF^2 \\
R_{4,p} &\triangleq \xrightarrow{\text{mo}\,p} \cap\, (Op^2 \setminus LSF^2) \\
R_{5,p} &\triangleq (R_{1,p} \cup R_{2,p} \cup R_{3,p} \cup R_{4,p})^\star \\
R_{6,p} &\triangleq \{(l,s) \in (L \times SF) | \neg l R_{5,p} s \wedge \neg s R_{5,p} l\} \\
R_{7,p} &\triangleq (R_{5,p} \cup R_{6,p})^\star \\
R_{8,p} &\triangleq \text{linext}_{Op}^{N}\, R_{7,p}
\end{aligned}
$$

Op basis van bovenstaande relaties definiëren we de uitvoering $E'$ als volgt: $E' = (Op, \xrightarrow{\text{po}}, R_{8,1} \ldots R_{8,n})$.

1. De relaties $R_{1,m}$, $R_{2,p}$, $R_{3,p}$ en $R_{4,p}$ zijn deelrelaties van $\xrightarrow{\text{mo}\,p}$.
   Dit volgt respectievelijk uit de eigenschappen (j), (l), (n) en uit de definitie van $R_{4,p}$.

2. $R_{5,p}$ is een partiële ordening, en een deelrelatie van $\xrightarrow{\text{mo}\,p}$.
   Dit volgt uit de transitiviteit van de operator transitieve sluiting en uit de herhaalde toepassing van lemma A.2.11.

3. $R_{7,p}$ is een partiële ordening in $Op$, en een totale ordening in $Op^2 \setminus (Op \setminus SF)^2$.

Dit volgt rechtstreeks uit lemma A.2.9. Gevolg: $R_{7,p} = R_{5,p} \cup (R_{6,p}; R_{5,p})$, en dus geldt dat $R_{7,p} \cap A_p = R_{5,p} \cap A_p$.

4. Gevolg: elke relatie $R_{8,p}$ is een totale ordening in *Op*, of eigenschap (c) geldt voor uitvoering $E'$.

5. $R_{8,p} \cap A_p = \overset{\text{mo}\,p}{\longrightarrow} \cap A_p$.
Omdat $R_{7,p}$ een totale ordening is in $Op^2 \setminus (Op \setminus SF)^2$, en omdat $A_p \subset Op^2 \setminus (Op \setminus SF)^2$, geldt dat $R_{8,p} \cap A_p = R_{7,p} \cap A_p$. Wegens stap 3 geldt ook dat $R_{7,p} \cap A_p = R_{5,p} \cap A_p$, en uit stap 2 volgt verder dat $R_{5,p} \cap A_p = \overset{\text{mo}\,p}{\longrightarrow} \cap A_p$. Gevolg: $R_{8,p} \cap A_p = \overset{\text{mo}\,p}{\longrightarrow} \cap A_p$.

6. Omdat $R_{1,m} \subset R_{8,p}$ ongeacht $p$, geldt eigenschap (j) voor $E'$.

7. Omdat $R_{2,p} \subset R_{8,p}$ ongeacht $p$, geldt eigenschap (l) voor $E'$.

8. Omdat $R_{3,p} \subset R_{8,p}$ ongeacht $p$, geldt eigenschap (m) voor $E'$.

9. Omdat $R_{8,p} \cap A_p = \overset{\text{mo}\,p}{\longrightarrow} \cap A_p$ en omdat eigenschap (3.3) geldt voor $E$, geldt deze eigenschap ook voor $E'$.

10. Omdat *Op* een eindige verzameling is geldt eigenschap (3.8) voor $E'$.

11. Omdat de eigenschap coherentie of (3.9) geldt voor uitvoering $E$, en omwille van de definitie van de relaties $R_{8,p}$, geldt deze eigenschap ook voor uitvoering $E'$.

12. Wegens stap 7 geldt eigenschap (3.1) voor uitvoering $E'$.

13. Omdat eigenschap (3.2) geldt voor uitvoering $E$, wegens stap 7 en omdat $R_{4,p} \subset R_{8,p}$ gelden eigenschappen (3.2), (3.4), (3.5), (3.6) en (3.7) ook voor uitvoering $E'$.

14. Gevolg van de twee voorgaande stappen: eigenschap (r) geldt ook voor uitvoering $E'$.

**Lemma B.4.7** *Als de uitvoering $E = (Op, \overset{po}{\longrightarrow})$ voldoet aan de definiërende eigenschappen van het PCD*-model, nl. (i), (l), (m), (3.3), (3.8), (3.9) en (r), en als bovendien Op een eindige verzameling is, dan voldoet dezelfde uitvoering E ook aan de eigenschap (c). Via tabel 3.19 volgt onmiddellijk dat dan ook de overige eigenschappen (d), (f), (g), (h), (j), (o) en (q) gelden.*

**Bewijs**

Omdat de uitvoering $E = (Op, \xrightarrow{\text{po}})$ voldoet aan het geheugenmodel PCD$^\star$, bestaan er geheugenordeningen $\xrightarrow{\text{mo}\,1}$ t.e.m. $\xrightarrow{\text{mo}\,n}$ die voldoen aan de definiërende eigenschappen van PCD$^\star$. Vertrekkende van de uitvoering $E = (Op, \xrightarrow{\text{po}}, \xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n})$ definiëren we volgende verzameling en relaties, met $p \in P$ en $m \in \mathit{Mem}$:

$$
\begin{aligned}
A_p &\triangleq \mathit{SF} \times (\mathit{SF} \cup L_p) \\
R_1 &\triangleq \xrightarrow{\text{mo}\,1} \cap\, \mathit{SF}^2 \\
R_2 &\triangleq \xrightarrow{\text{po}} \cap\, \mathit{SF}^2 \\
R_3 &\triangleq \xrightarrow{\text{po}} \cap\, (\mathit{LF} \times \mathit{LSF}) \\
R_{4,p} &\triangleq \xrightarrow{\text{mo}\,p} \cap\, (S \times L_p) \\
R_{5,p} &\triangleq \xrightarrow{\text{mo}\,p} \cap\, (\mathit{Op}^2 \setminus \mathit{LSF}^2) \\
R_{6,p} &\triangleq (R_1 \cup R_2 \cup R_3 \cup R_{4,p} \cup R_{5,p})^\star \\
R_{7,p} &\triangleq \{(l, s) \in (L \times \mathit{SF}) | \neg l R_{6,p} s \wedge \neg s R_{6,p} l\} \\
R_{8,p} &\triangleq (R_{6,p} \cup R_{7,p})^\star \\
R_{9,p} &\triangleq \text{linext}^N_{Op}\, R_{8,p}
\end{aligned}
$$

Op basis van bovenstaande relaties definiëren we de uitvoering $E'$ als $E' = (Op, \xrightarrow{\text{po}}, R_{9,1} \ldots R_{9,n})$. Het bewijs dat $E'$ een geldige uitvoering is die aan de gevraagde eigenschappen voldoet is als volgt:

1. $R_1$, $R_2$, $R_3$, $R_{4,p}$ en $R_{5,p}$ zijn deelrelaties van $\xrightarrow{\text{mo}\,p}$ en $R_2 \subset R_1$.
   Voor de relaties $R_1$, $R_{2,p}$ en $R_{3,p}$ volgt het gestelde uit het gegeven dat de uitvoering $E$ voldoet aan de respectieve eigenschappen (i), (l) en (m). Relaties $R_{4,p}$ en $R_{5,p}$ zijn per definitie deelrelaties van $\xrightarrow{\text{mo}\,p}$. Uit (i) volgt dat $R_2 \subset R_1$.

2. $\xrightarrow{\text{mo}\,p} \cap\, A_p = R_1 \cup R_{4,p}$.
   Dit volgt uit de definities van $R_1$, $R_{4,p}$ en $A_p$.

3. Elke relatie $R_{6,p}$ is een partiële ordening.
   Dit volgt uit de herhaalde toepassing van lemma A.2.11.

4. Elke relatie $R_{6,p}$ is een totale ordening in *SF*. Dit volgt uit (i) en $R_1 \subset R_{6,p}$.

5. $R_{6,p} \cap A_p = \xrightarrow{\text{mo}\,p} \cap\, A_p$.
   Uit $R_{6,p} \subset \xrightarrow{\text{mo}\,p}$ volgt dat $R_{6,p} \cap A_p \subset \xrightarrow{\text{mo}\,p} \cap A_p$. Uit de definitie van $R_{6,p}$ en uit $\xrightarrow{\text{mo}\,p} \cap A_p = R_1 \cup R_{4,p}$ volgt dat $\xrightarrow{\text{mo}\,p} \cap A_p \subset R_{6,p} \cap A_p$. .

6. $R_{8,p}$ is een partiële ordening, en totaal in $Op^2 \setminus (Op \setminus SF)^2$.
   Dit volgt rechtstreeks uit de toepassing van lemma A.2.9 op de relaties $R_{6,p}$ en $R_{7,p}$, voor de verzamelingen $A = Op$ en $B = SF$. Uit datzelfde lemma volgt ook dat $R_{8,p} = R_{6,p} \cup (R_{7,p}; R_{6,p})$. Gevolg: $R_{8,p} \cap A_p = R_{6,p} \cap A_p$.

7. $\xrightarrow{\text{po}\,p} \cap\, LSF_m^2 \subset R_{8,p}$.
   Omdat de eigenschap uniprocessorcorrectheid (3.1) geldt voor uitvoering $E$ volgt dat $\xrightarrow{\text{po}\,p} \cap\, (LSF_m^2 \setminus L_m^2) \subset \xrightarrow{\text{mo}\,p}$. Omdat eigenschap (o) geldt voor uitvoering $E$ volgt dat $\xrightarrow{\text{po}\,p} \cap\, L_m^2 \subset \xrightarrow{\text{mo}\,p}$. Gevolg: $\xrightarrow{\text{po}\,p} \cap\, LSF_m^2 \subset \xrightarrow{\text{mo}\,p} \cap\, LSF_m^2 \subset R_{8,p}$.

8. $R_{9,p}$ is een partiële ordening in $Op$ omdat $R_{8,p}$ dat ook is.

9. $R_{9,p} \cap A_p = \xrightarrow{\text{mo}\,p} \cap A_p$.
   Omdat $R_{8,p}$ een totale ordening is in $Op^2 \setminus (Op \setminus SF)^2$, en omdat $A_p \subset Op^2 \setminus (Op \setminus SF)^2$, geldt dat $R_{9,p} \cap A_p = R_{8,p} \cap A_p$. Wegens stap 6 geldt ook dat $R_{8,p} \cap A_p = R_{6,p} \cap A_p$, en uit stap 5 volgt verder dat $R_{6,p} \cap A_p = \xrightarrow{\text{mo}\,p} \cap A_p$. Gevolg: $R_{9,p} \cap A_p = \xrightarrow{\text{mo}\,p} \cap A_p$.

10. Omdat $R_1 \subset R_{9,p}$ ongeacht $p$, geldt (i) voor $E'$.

11. Omdat $R_2 \subset R_{9,p}$ ongeacht $p$, geldt (l) voor $E'$.

12. Omdat $R_3 \subset R_{9,p}$ ongeacht $p$, geldt (m) voor $E'$.

13. Omdat $R_{9,p} \cap A_p = \xrightarrow{\text{mo}\,p} \cap A_p$ en omdat eigenschap (3.3) geldt voor $E$, geldt deze eigenschap ook voor $E'$.

14. Omdat $Op$ een eindige verzameling is geldt eigenschap (3.8) voor $E'$.

15. Omdat de eigenschap coherentie of (3.9) geldt voor uitvoering $E$, en omwille van de definitie van de relaties $R_{9,p}$, geldt deze eigenschap ook voor uitvoering $E'$.

16. Wegens stap 7 geldt eigenschap (3.1) voor uitvoering $E'$.

17. Omdat eigenschap (3.2) geldt voor uitvoering $E$, wegens stap 7 en omdat $R_{5,p} \subset R_{9,p}$ gelden eigenschappen (3.2), (3.4), (3.5), (3.6) en (3.7) ook voor uitvoering $E'$.

18. Gevolg van de twee voorgaande stappen: eigenschap (r) geldt ook voor uitvoering $E'$.

**Lemma B.4.8** *Als de uitvoering $E = (Op, \xrightarrow{po}, \xrightarrow{mo})$ een eindig aantal opdrachten bevat en voldoet aan de definiërende eigenschappen van het veralgemeende geheugenmodel IA-64$^\star$, nl. (g), (3.3), (3.8), (3.9) en (r), dan voldoet dezelfde uitvoering ook aan de eigenschap (c). Via tabel 3.19 volgt onmiddellijk dat dan ook de overige eigenschappen van het geheugenmodel IA-64$^\star$ gelden, nl. (f) en (j).*

**Bewijs**

Omdat de uitvoering $E = (Op, \xrightarrow{\text{po}})$ voldoet aan het geheugenmodel IA-64$^\star$, bestaan er geheugenordeningen $\xrightarrow{\text{mo}\,1}$ t.e.m. $\xrightarrow{\text{mo}\,n}$ die voldoen aan de definiërende eigenschappen van IA-64$^\star$. Vertrekkende van de uitvoering $E = (Op, \xrightarrow{\text{po}}, \xrightarrow{\text{mo}\,1} \ldots \xrightarrow{\text{mo}\,n})$ definiëren we volgende relaties, met $p \in P$:

$$R_p \triangleq \text{linext}_{Op}^{N} \xrightarrow{\text{mo}\,p}.$$

Op basis van bovenstaande relaties definiëren we de uitvoering $E'$ als volgt: $E' = (Op, \xrightarrow{\text{po}}, R_1 \ldots R_n)$.

1. Elke relatie $R_p$ is per definitie een totale ordening, dus geldt eigenschap (c) voor uitvoering $E'$.
2. Wegens eigenschap (g) geldt dat elke relatie $\xrightarrow{\text{mo}\,p}$ een totale ordening is in de verzameling $Op_m$. Gevolg: $R_p \cap Op_m^2 = \xrightarrow{\text{mo}\,p} \cap Op_m^2$. Omdat eigenschap (3.3) geldt voor uitvoering $E$ geldt deze eigenschap dus ook voor de uitvoering $E'$. Ook eigenschap (g) blijft dus behouden voor de uitvoering $E'$.
3. Omdat *Op* een eindige verzameling is geldt eigenschap (3.8) voor $E'$.
4. Omdat de eigenschappen coherentie (3.9) en ook (g) gelden voor uitvoering $E$, en omwille van de definitie van de relaties $R_p$, geldt deze eigenschap ook voor uitvoering $E'$.
5. Omdat de eigenschappen (3.1), (3.2), (3.4), (3.5), (3.6) en (3.7) gelden voor uitvoering $E$, en omdat $\xrightarrow{\text{mo}\,p} \subset R_p$, gelden deze eigenschappen ook voor uitvoering $E'$.
6. Gevolg van de voorgaan stap: eigenschap (r) geldt ook voor uitvoering $E'$.

**Lemma B.4.9** *Als de uitvoering $E = (Op, \xrightarrow{po}, \xrightarrow{mo})$ een eindig aantal opdrachten bevat en voldoet aan de definiërende eigenschappen van het veralgemeende berichtencommunicatiemodel causaal geordende berichten of CM$^\star$, nl.*

*(b), (f), (k), (3.8), (3.9), (s), (u) dan voldoet dezelfde uitvoering ook aan de overige eigenschappen van het berichtencommunicatiemodel AM*, nl. (c), (g), (i), (j), (l), (m), (n), (o), (p), (q), (3.8), en (3.9). en (t). Een analoge eigenschap geldt voor het berichtencommunicatiemodel synchrone berichtenordening of SM*, waarvan de definitie die van CM* is behalve dat eigenschap (u) door eigenschap (t) dient te worden vervangen.*

**Bewijs**

We definiëren eerst onderstaande relaties:

$$R \triangleq \text{linext}^{N}_{LSF}\ \xrightarrow{\text{mo}}$$

Relatie $R$ is per definitie een totale ordening in *LSF*, en een een partiële ordening in *Op*. Definieer nu de uitvoering $E'$ als $E' = (Op, \xrightarrow{\text{po}}, R)$. Eigenschap (b) geldt dus wegens de definitie van $E'$. Omdat $\xrightarrow{\text{mo}} \subset R$ voldoet $E'$ aan (h). Uit de definitie van $R$ volgt ook dat $\xrightarrow{\text{mo}} \cap (LSF^2 \setminus L^2) \subset R$, en dus dat de relaties SR verbonden aan de uitvoeringen $E$ en $E'$ identiek zijn. Gevolg: de eigenschappen (k), (3.12), (3.13) en (3.2), geldig voor $E$, gelden ook voor $E'$. Door de bovenstaande eigenschappen te combineren met tabel 3.19 volgt er dat voor de uitvoering $E'$ volgende eigenschappen gelden: (b), (c), (f), (3.8), (3.9), (g), (i), (j), (k), (l), (m), (n), (o), (p), (q), (3.8), (3.9) en (s). Omdat de relaties SR en $\xrightarrow{\text{mo}} \cap (LSF^2 \setminus L^2)$ identiek zijn voor uitvoeringen $E$ en $E'$, blijven de eigenschappen (t) en (u), indien geldig voor $E$, behouden voor uitvoering $E'$.

**Lemma B.4.10** *Als de uitvoering $E = (Op, \xrightarrow{po}, \xrightarrow{mo})$ een eindig aantal opdrachten bevat en voldoet aan de definiërende eigenschappen van het veralgemeende berichtencommunicatiemodel asynchrone berichtenordening of AM*, nl. (b), (f), (3.8), (3.9), en (s), dan voldoet dezelfde uitvoering ook aan de overige eigenschappen van het berichtencommunicatiemodel AM*, nl. (c), (g), (i), (j) en (l). Een analoge eigenschap geldt voor het berichtencommunicatiemodel FIFO-berichtenordening of FM*, dat eigenschap (v) als bijkomende definiërende eigenschap heeft.*

**Bewijs**

We definiëren volgende relaties:

$$
\begin{aligned}
R_1 &\triangleq \xrightarrow{\text{po}} \cap\, SF^2 \\
R_2 &\triangleq (\text{SR} \cup R_1)^\star \\
R_3 &\triangleq \bigcup_{m \in Mem} \xrightarrow{\text{mo}\,1} \cap\, LSF_m^2 \\
R_4 &\triangleq \bigcup_{m \in Mem} \{(l,s) \in (L_m \times SF_m) | \neg(l \xrightarrow{\text{mo}} s) \wedge \neg(s \xrightarrow{\text{mo}} l)\} \\
R_5 &\triangleq (R_3 \cup R_4)^\star \\
R_6 &\triangleq \text{linext}_{LSF}^N\, R_5
\end{aligned}
$$

De relatie $R_1$ is per definitie een partiële ordening. Uit eigenschappen (3.2) en SR $\subset \xrightarrow{\text{mo}}$ volgt resp. dat $R_1 \subset \xrightarrow{\text{mo}}$ en ook dat SR $\subset \xrightarrow{\text{mo}}$. Relatie $R_2$ is dus ook een partiële ordening, met $R_2 \subset \xrightarrow{\text{mo}}$. Relaties $R_3$ en $R_4$ voldoen aan de voorwaarden van lemma A.2.9 voor de verzamelingen $A = LSF$ en $B = SF$. Uit de toepassing van dat lemma volgt dat relatie $R_5$ een partiële ordening is in $LSF$, en een totale ordening in $LSF^2 \setminus L^2$. Relatie $R_6$ is dus een totale ordening in verzameling $LSF$. Uit lemma A.2.7 volgt verder dat relatie $R_6$ een partiële ordening is in $Op$. We definiëren de uitvoering $E'$ als volgt: $E' = (Op, \xrightarrow{\text{po}}, R_6)$. Omdat $R_3 \subset R_6$ geldt is met uitvoering $E'$ dezelfde relatie SR verbonden als met uitvoering $E$. Per definitie van $E'$ zijn voor uitvoering $E'$ alle geheugenordeningsrelaties gelijk, en dus geldt eigenschap (b) voor $E'$. Daar $R_6 \subset R_6$ geldt, en omdat $R_6$ een totale ordening is in $LSF$, geldt ook eigenschap (h). De eigenschap (3.8) en (3.9) zijn op triviale wijze voldaan omdat er een eindig aantal opdrachten is resp. er slechts één geheugenordeningsrelatie is. Omdat relatie SR behouden werd, blijven de eigenschappen (3.12) en (3.13) van uitvoering $E$ ook voor uitvoering $E'$ gelden. En omdat $R_1 \subset R_6$ geldt eigenschap (3.2) voor $E'$. Omdat $R_6$ een totale ordening is in $LSF$, en omdat $R_6 \subset R_6$, volgt er dat eigenschap (h) geldt. Uit de combinatie van bovenstaande eigenschappen en tabel 3.19 volgt dat eigenschappen (b), (c), (f), (3.8), (3.9), (g), (i), (j), (l) en (s) gelden voor $E'$, en dus dat $E'$ voldoet aan alle eigenschappen van het berichtencommunicatiemodel AM$^\star$. In het geval dat uitvoering $E$ voldoet aan de eigenschap (v), voldoet ook uitvoering $E'$ aan deze eigenschap omdat relaties $\xrightarrow{\text{po}}$ en SR in beide uitvoeringen identiek zijn. Gevolg: elke uitvoering die voldoet aan de definiërende eigenschappen van het berichtencommunicatiemodel FM$^\star$, voldoet ook aan de overige eigenschappen van datzelfde berichtencommunicatiemodel.