Universiteit Gent
Faculteit van de Toegepaste Wetenschappen

Seminarie voor Toegepaste Wiskunde en Biometrie
Directeur: Prof. Dr. ir. G.C. Vansteenkiste

HGPSS: Object-georiënteerde "process-interaction" simulatie

door
Filip Claeys

Promotor: Prof. Dr. ir. G.C. Vansteenkiste
Thesisbegeleider: H. Vangheluwe

Deel I

Afstudeerwerk ingediend tot het behalen van de graad van
Licentiaat in de Informatica

Academiejaar 1991-1992

Dankwoord

De auteur wenst alle medewerkers van het Seminarie voor Toegepaste Wiskunde en Biometrie te bedanken voor hun bijdrage tot het welslagen van dit eindwerk.

Een bijzondere dankbetuiging gaat uit naar Hans Vangheluwe die als thesisbegeleider steeds klaarstond om het project in goede banen te leiden en te stimuleren door het leveren van opbouwende kritiek.

Toelating

Contents

List of Figures

List of Tables

Chapter 1

Inleiding

1.1 Probleemstelling

Simulatie is vandaag als probleemoplossende methode algemeen aanvaard. Ze biedt uitkomst in die gevallen waar een analytische oplossing voor een probleem onmogelijk of te tijdrovend is. Nieuwe ontwikkelingen op het gebied van software en hardware hebben het spectrum van toepassingen waarbinnen simulatie-technieken kunnen ingezet worden, nog uitgebreid.

Vele commerciële, zowel algemene als eerder probleemspecifieke simulatie-softwarepakketten zijn op de markt. Ondanks de kracht van vele van deze pakketten kunnen toch een aantal tekortkomingen onderscheiden worden.

- Ten eerste zijn de meeste pakketten gericht op simulatie van hetzij continue, hetzij discrete systemen. De systemen binnen de reële wereld situeren zich echter hoofdzakelijk in het spectrum dat ligt tussen de zuiver continue en de zuiver discrete systemen. Simulatie van een dergelijk hybride systeem met een pakket dat zich richt op één van de uiteinden van het spectrum, zal vereisen dat er binnen het model van het te simuleren systeem abstractie gemaakt wordt van de met de aard van het pakket niet compatibele eigenschappen. Men kan de niet compatibele eigenschappen ook trachten te vertalen naar eigenschappen die wel compatibel zijn met het binnen het pakket gebruikte formalisme. Een dergelijke werkwijze is echter onnatuurlijk en zal de duidelijkheid niet stimuleren. Voorts zal een aanzienlijk verlies aan performantie in vergelijking met een pakket dat wel geschikt is, niet irreëel zijn. Voor de oplossing van een hybride probleem zou kunnen gedacht worden aan de combinatie van een pakket specifiek gericht op continue simulatie en een pakket gericht op discrete simulatie. Deze koppeling blijkt echter in de praktijk verre van eenvoudig te zijn.

- In tweede instantie blijken er in vele gevallen problemen op te duiken bij het incorporeren van uitbreidingen als een aantrekkelijke gebruiksomgeving of andere additionele personaliserende software, in een simulatie-pakket. Concreet is het koppelen van de simulatie-software aan door de gebruiker of door derden in één of andere klassieke programmeertaal ontwikkelde programmatuur voor pre- of postprocessing, een moeilijke zaak.

- Het ontbreken van voorzieningen voor hiërarchische modellering in vooral oudere simulatie-pakketten, is een derde te onderkennen probleem. Principes als abstractie, het verschuilen van informatie en object-oriëntatie hebben hun nut bewezen bij de ontwikkeling van steeds complexere software. Ze zouden ook moeten kunnen toegepast worden bij de modellering van systemen. De noodzaak

van het gebruik van deze principes kan ook doorgetrokken worden naar de implementatie van het simulatie-pakket ansich.

- Uiteindelijk is het soms wenselijk om direct te interageren met de low-level aspecten van het simulatie-proces. In vele gevallen zijn deze aspecten volledig afgeschermd van de gebruiker zodat deze op geen enkele manier kan ingrijpen of uitbreiden waar nodig.

1.2 Oplossing

Samengevat zijn de vereisten gesteld aan een simulatie-pakket dat de gestelde tekortkomingen oplost, de volgende:

- Modellering van hybride systemen moet op een natuurlijke wijze uit te voeren zijn.
- Uitbreiden en personaliseren van het pakket moet op een efficiënte manier door te voeren zijn en mag geen onoverkomelijke moeilijkheden met zich meebrengen.
- Object-georiënteerde technieken moeten consistent toegepast worden, zowel bij de implementatie van de simulatie-engine als bij het modelleren.
- Het geheel van routines die de uiteindelijke simulatie ondersteunen moeten toegankelijk blijven voor de gebruiker.

Om deze desiderata te verwezelijken zou een volledig nieuwe simulatie-taal en bijbehorende run-time omgeving kunnen geconcipieerd worden die simulatie van hybride systemen gecombineerd met adaptatie toelaat. Het incorporeren van voorzieningen voor het simuleren van discrete en continue systemen binnen eenzelfde pakket impliceert echter heel wat compromissen, zodat noch het discrete, noch het continue deel optimaal zullen zijn. De afzonderlijke ontwikkeling en latere koppeling van een pakket voor de simulatie van continue systemen en één voor de simulatie van discrete systemen, is een betere oplossing. De vraag stelt zich echter of het noodzakelijk is om voor beide partners een nieuwe simulatie-taal te ontwikkelen. Het nut van nieuwe talen is in het licht van het grote aantal reeds bestaande talen twijfelachtig. Een nieuwe taal zou te kampen hebben met de problemen waarmee nieuwe programmeer-talen over het algemeen te kampen hebben. Zo wordt de geschiktheid van bepaalde constructies binnen de taal pas bewezen na extensief gebruik. Voorts worden grote hoeveelheden software ontwikkeld in reeds bestaande talen onbruikbaar. Een betere oplossing is om een bestaande simulatie-taal voor de simulatie van continue systemen en één voor de simulatie van discrete systemen zodanig uit te breiden en aan te passen dat gebruik kan gemaakt worden van de eventueel jarenlange expertise in verband met deze talen, zonder echter de vooropgestelde desiderata uit het oog te verliezen. Dit is dan ook de strategie die gevolgd werd.

1.3 Implementatie van de oplossing

Bij de implementatie van een oplossing voor de gestelde problemen werd concreet volgende werkwijze gevolgd:

- Een bestaande taal voor het modelleren en simuleren van discrete systemen werd aangepast en uitgebreid om de koppeling met een continue simulatie-partner mogelijk te maken en het incorporeren van andere externe software niet uit te sluiten. Een analoge maar duale operatie werd uitgevoerd op een taal voor de simulatie van continue systemen.

- Ter ondersteuning van de talen werden twee verzamelingen simulatie-routines in een object-georiënteerde general-purpose programmeertaal ontwikkeld en als een open systeem geconcipieerd.
- Als interface tussen simulatie-taal en ondersteuningsroutines fungeert een precompiler die constructies uit de simulatie-taal omzet naar statements in de general-purpose programmeertaal.

De wijze waarop de continue simulatie-partner werd geïmplementeerd, wordt besproken in [Vanwijnsberghe 1992]. Dit document richt zich op de beschrijving van de door de auteur ontwikkelde discrete simulatie-partner.

De discrete simulatie-partner werd verwezenlijkt door het als volgt geschetste procédé toe te passen.

- De wijd verspreide *discrete event* simulatie-taal *GPSS* (General-Purpose Simulation System) en meer specifiek de variant GPSS/360, werd uitgebreid met een aantal constructies om aan de gestelde eisen tegemoet te komen.
  - Voorzieningen voor de koppeling met een continue simulatie-partner werden ingebouwd.
  - Aangezien GPSS/360 het hiërarchisch modelleren niet conceptueel ondersteunt, werden een aantal primitieven voorzien om dit euvel uit de wereld te helpen.
  - Het incorporeren van externe software werd mogelijk gemaakt door het inbedden toe te laten van stukken software geschreven in de taal waarin de onsteuningsroutines geïmplementeerd werden, in simulatie-programma's.

  De uitgebreide versie van GPSS/360 werd *HGPSS* (Hierarchical General-Purpose Simulation System) gedoopt.
- Een verzameling van voor de gebruiker transparante routines ter ondersteuning van hiërarchische discrete-event simulatie werd ontwikkeld in de klassieke object-georiënteerde programmeertaal *C++* en samengebundeld in een zogenaamde *simulatie-kernel*. Gepoogd werd om deze kernel zou nauw mogelijk te laten annleunen bij HGPSS om de semantische kloof tussen kernel en HGPSS zo klein mogelijk te houden. Aangezien de kernel zowel elementen van HGPSS als van C++ in zich draagt, werd *HGPSS++* (Hierarchical General-Purpose Object Oriented Simulation System) als naam gekozen.
- Als interface tussen HGPSS en HGPSS++ werd een precompiler ontwikkeld die constructies uit de simulatie-taal HGPSS omzet naar aanroepen van functies die deel uitmaken van de kernel. De verzameling C++-functies die instaat voor de communicatie met de kernel, kan als een taal op zich beschouwd worden en als de *HGPSS++-taal* aangeduid worden. Op microscopisch gebied wordt een programma bestaande uit HGPSS-statements en ingebedde C++-statements door de precompiler omgezet naar een programma bestaande uit C++-statements en ingebedde HGPSS++-statements. C++ kan beschouwd worden als gasttaal voor HGPSS.

Chapter 2

Discrete-event simulatie

2.1 Terminologie

Simulatie [Spriet & Vansteenkiste 1982, Kreutzer 1986] is een ruim begrip. Niet enkel het uitvoeren van een simulatie met het oog op het verzamelen van nuttige resultaten zit erin vervat. Ook het proces leidende van de analyse van een probleem tot de implementatie ervan, resulterende in de *simulator*, is een facet van simulatie. Het simulator-constructieproces kan in twee fasen onderverdeeld worden.

- In eerste instantie moet uitgaande van het probleem een *model* of een *formalisme* opgesteld worden. Deze fase wordt het *modelleren* genoemd. Het uitzicht van het model zal bepaald worden door de kennis van het probleem en door de uiteindelijke doelstelling van de simulatie. Kennis van het probleem kan a priori aanwezig zijn, of proefondervindelijk afgeleid worden.

- In de tweede fase, de *implementatie*, wordt de eigenlijke simulator aangemaakt, uitgaande van het model. Deze simulator zal een one-to-one mapping zijn van het initiële probleem, beperkt tot deze eigenschappen die een zekere mate van relevantie hebben. Bij de modelconstructie werd immers *abstractie* gemaakt van de irrelevante en ondergeschikte eigenschappen. De bruikbaarheid van de resultaten verkregen door simulatie zullen in hoge mate afhankelijk zijn van de geschiktheid van het model.

Bij de analyse van een systeem in de context van simulatie kan gesteld worden dat dit systeem bestaat uit een verzameling van *entiteiten* die met elkaar interageren. Een entiteit wordt gekenmerkt door een aantal *attributen.* Deze attributen zijn onveranderlijk in functie van de tijd, of juist niet. Dit resulteert in *statische* en *dynamische systemen.* De waarden van de attributen van een entiteit op een bepaald moment bepalen de *toestand* van de entiteit. De begrippen simulatie en model kunnen in functie van de noties entiteit en attribuut geherformuleerd worden.

- Simulatie kan beschouwd worden als de studie van de verandering van de attributen van entiteiten van een systeem gedurende een bepaalde periode.

- De simulatie wordt mogelijk door de constructie van een geschikt model voor de entiteiten, hun attributen en de wijze waarop ze interageren.

Discrete-event simulatie onderscheidt zich van *continue simulatie* door de gerichtheid op systemen uit een discrete-event wereld. Dit is een wereld waarin veranderingen van de waarde van entiteitsattributen slechts op discrete ogenblikken plaatsvinden. Entiteiten zullen dus slechts op discrete tijdstippen van toestand veranderen. In de tijdspannes tussen deze tijdstippen vinden geen toestandsveranderingen

plaats. Een dergelijke toestandsverandering wordt een *event* genoemd. Binnen de continue wereld kunnen deze veranderingen in principe op elk moment optreden.

Modellen voor discrete-event simulatie kunnen verder onderverdeeld worden naargelang een aantal van hun karakteristieken:

- Een model kan zuiver *numeriek* of zuiver *analytisch* zijn. Een combinatie van beiden is ook mogelijk.
- Een model kan net als een systeem *statisch* of *dynamisch* zijn. Statische modellen zijn tijdsinvariant voor gelijkblijvende stimuli.
- Uiteindelijk kan een model ook *deterministisch* of *stochastisch* zijn. In een determistisch model worden alle relaties beschreven door vaste mathematische regels. In een stochastisch model anderzijds, zijn één of meerdere relaties onderhevig aan een bepaalde mate van willekeur.

In de praktijk blijken de hoofdmoot van de discrete-event simulatie-modellen te catalogeren te zijn als numeriek, dynamisch en stochastisch.

Gezien binnen een discrete-event model toestandsveranderingen enkel op discrete punten binnen het verloop van de tijd plaatsvinden en hetgeen gebeurt tussen deze tijdspunten in, irrelevant is, kan discrete-event simulatie uitgevoerd worden door op een bepaald moment de noodzakelijke toestandsveranderingen door te voeren en de aandacht dan te verleggen naar het volgende relevante tijdspunt. Er wordt met andere woorden van actief tijdspunt naar actief tijdspunt gerekend door over de inactive tijd heen te springen. Deze aanpak wordt de *next event approach* genoemd.

2.2 Events, activiteiten en processen

Een systeem kan gemodelleerd worden door naast de entiteiten, een verzameling events en de verzameling tijdspunten waarop de events plaatsvinden te beschouwen. De events zijn gerelateerd aan veranderingen in de waarden van de entiteitsattributen. De sequentie van events geeft een zeer gedetailleerd maar relatief onduidelijk beeld van de toestandsveranderingen binnen een model. De reden voor die onduidelijkheid ligt in het feit dat de events van toepassing zijn op verschillende entiteiten of deelverzamelingen van entiteiten, zonder dat deze entiteiten enige mate van samenhang hoeven te vertonen.

Een hoger niveau van abstractie kan verkregen worden door niet de events op zich te beschouwen, maar events die betrekking hebben op een zelfde entiteit en die een bepaalde mate van chronologie vertonen, samen te bundelen en te beschouwen als een *proces*. Binnen een model kunnen meerdere processen concurrent evolueren.

Tussen het event- en het procesniveau kan een intermediair niveau ingevoerd worden. Dit niveau heeft betrekking op activiteiten. Een *activiteit* is een collectie van gebeurtenissen die de toestand van een entiteit veranderen. Activiteiten zijn die toestandstransformaties die in de reële wereld tijd in beslag nemen. Een activiteit wordt typisch geïnitieerd als aan een aantal condities voldaan is. Een andere verzameling condities duidt het einde van de activiteit aan.

In Figuur 2.1 wordt het verschil tussen event, activiteit en proces geïllustreerd aan de hand van een voorbeeld. Als voorbeeld werd een warenhuis gekozen. Dit systeem bestaat onder andere uit klanten en een kassa. Een klant start z'n bezoek aan het warenhuis door de produkten uit te kiezen die hij nodig heeft. Deze fase zal verder door *winkelen* aangeduid worden. Na het winkelen begeeft de klant zich naar de kassa en betaalt. Aan de kassa kan eventueel een wachtlijn onstaan. In dit geval zal een klant eerst aan de rij aansluiten, z'n beurt afwachten, dan de rij verlaten en betalen. Deze twee laatste fases zullen respectievelijk *wachten* en *afrekenen* genoemd worden. De entiteiten waaruit het warenhuis-systeem

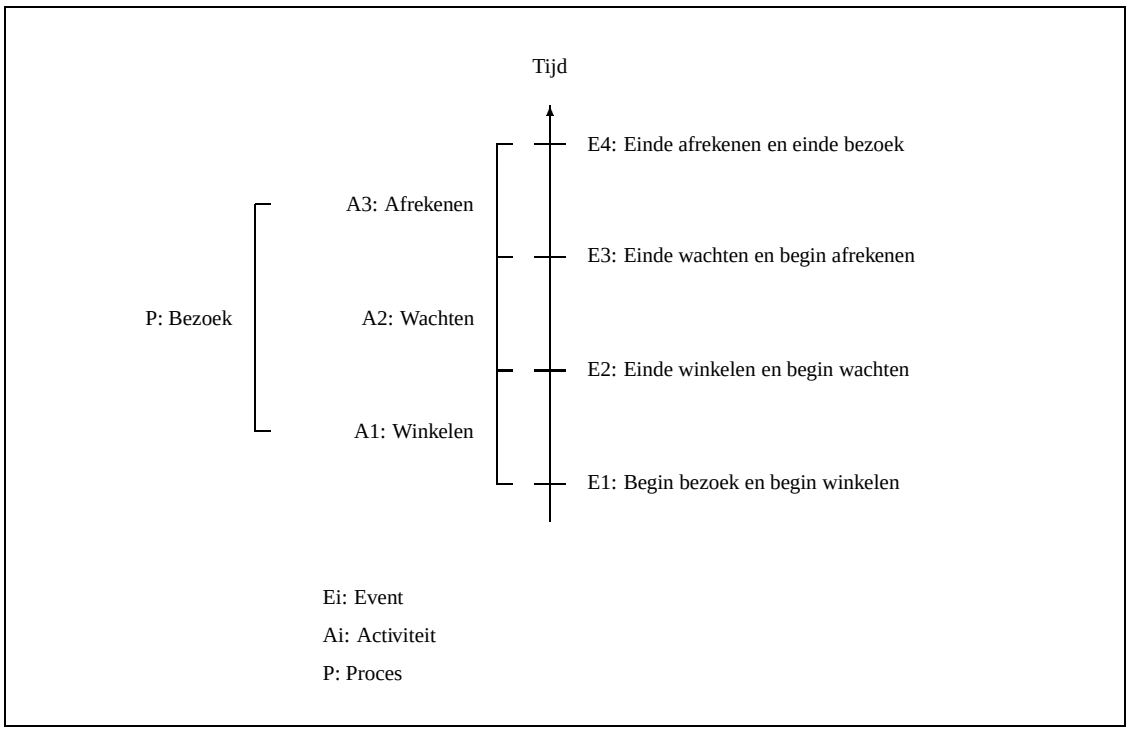

Figure 2.1: Event, activeit en proces

bestaat zijn de klanten en de kassa. De attributen van een klant zijn bijvoorbeeld het aantal produkten die hij koopt en daaruit volgend, de totale duur van het winkelen. Een kassa heeft als attribuut de tijd die het afhandelen van een klant in beslag neemt.

Event, activiteit en proces zijn meer dan zomaar drie begrippen. Elk geeft aanleiding tot een filosofie om discrete-event te benaderen. *Event-scheduling, activity-scanning* en *process-interaction* zijn de drie resulterende formalismes. Deze formalismes worden *world views* genoemd omdat ze staan voor een wijze om tegen de discrete-event wereld aan te kijken.

2.3 World views

2.3.1 Event-scheduling

De event-scheduling approach wordt ook kortweg *discrete-event approach* genoemd, alhoewel dit verwarring in de hand werkt met de algemenere context waarin het begrip discrete-event wordt gebruikt. De event-scheduling approach is een eerste formalisme of world view dat kan gehanteerd worden bij de constructie van een discrete-event simulatie-model. Deze aanpak veronderstelt dat bekend is welke events kunnen plaatsvinden en dat een beschrijving voorhanden is van de verschillende operaties die uitgevoerd moeten worden wanneer een bepaald event zich manifesteert. Deze operaties worden gebundeld binnen een routine. De routine zal worden aangeroepen wanneer het event plaatsvindt. Referenties naar events worden expliciet en in chronologische volgorde in een lijst geplaatst. Het in de lijst opnemen van de referenties wordt *scheduling* genoemd, terwijl de lijst zelf als *sequencing set*, *event list*, *noticeboard* of *scheduling list* aangeduid wordt. Tijdens het simulatie-proces zal de lijst doorlopen worden. Telkens een vertegenwoordiger van een event ontmoet wordt, wordt de bijbehorende routine aangeroepen. Over het

algemeen wordt een gespecialiseerd stuk software ingezet voor de afhandeling van alle scheduling- en event list management-taken: het *run time control system*, *scheduler*, *simulation executive* of *processor*. De referenties naar events opgeslagen in de event list, worden *event notices* genoemd. Een dergelijk event notice moet minimaal bestaan uit een referentie naar de aan te roepen event routine en naar het tijdstip waarop het event plaatsvindt. De processor houdt steeds het tijdstip bij tot hetwelke de simulatie gevorderd is. Dit tijdstip is de *simulatie-tijd* en wordt bijgehouden door middel van de *simulatie-klok*. Event-scheduling simulatie is een vorm van *imperative sequencing*, aangezien de scheduling enkel bepaald wordt door de tijd en niet door andere condities.

In Figuur 2.2 wordt het event-scheduling mechanisme geïllustreerd voor het warenhuis-systeem. Volgende events kunnen optreden:

- Aankomst in de winkel en aanvang van het winkelen.
- Beëindiging van het winkelen.
- Aanvang van het wachten.
- Beëindiging van het wachten.
- Aanvang van het afrekenen.
- Beëindiging van het afrekenen en verlaten van de winkel.

Naast de klanten en de kassa kan ook de rij als entiteit van het systeem beschouwd worden. De figuur vertolkt het scheduling-proces in een mogelijke modellering van het warenhuis-systeem. Elk blok symboliseert een event en de eraan gekoppelde event routine. De pijlen vertrekkend vanuit een event wijzen op de scheduling van andere events door de event routine. De taak van de event routines is tweeërlei en behelst naast het schedulen van events ook het uitvoeren van operaties op de entiteiten. De entiteiten worden gesymboliseerd door blokken in stippellijn. Een aantal parameters kenmerken het systeem. De tijd tussen opeenvolgende aankomsten van klanten in de winkel wordt door *aankomsttijd* aangeduid, de tijd die het winkelen in beslag neemt wordt *winkeltijd* genoemd, de duur van het wachten *wachttijd* en de duur van het afrekenen *afrekentijd*. In de praktijk zullen deze parameters niet constant zijn maar onderhevig aan een aantal factoren. Het scheduling-proces kan als volgt beschreven worden:

- Initieel wordt de aankomst van een klant in de winkel gescheduled. Bij de afhandeling van dit event wordt de aankomst van de volgende klant gescheduled. Het tijdstip waarop dit event zal plaatsvinden is het huidige tijdstip vermeerderd met de aankomsttijd. Naast de aankomst van de volgende klant wordt ook het einde van het winkelen van de huidige klant gescheduled. Dit event zal plaatsvinden op het huidige tijdstip vermeerderd met de winkeltijd.
- Beëindigen van het winkelen geeft aanleiding tot de scheduling van het aanvangen van het wachten.
- Aansluiten aan de rij geeft aanleiding tot het afsluiten van het wachten na de wachttijd. De wachttijd is niet constant maar is afhankelijk van het tempo waaraan de klanten aan de kassa worden bediend.
- Als het wachten beëindigd is, kan het afrekenen beginnen. Het einde van het afrekenen wordt gescheduled.
- Bij het afsluiten van het afrekenen wordt de klant uit het systeem verwijderd en wordt de volgende klant uit de rij toegelaten aan de kassa. Daartoe wordt een beëindiging van het wachten gescheduled.

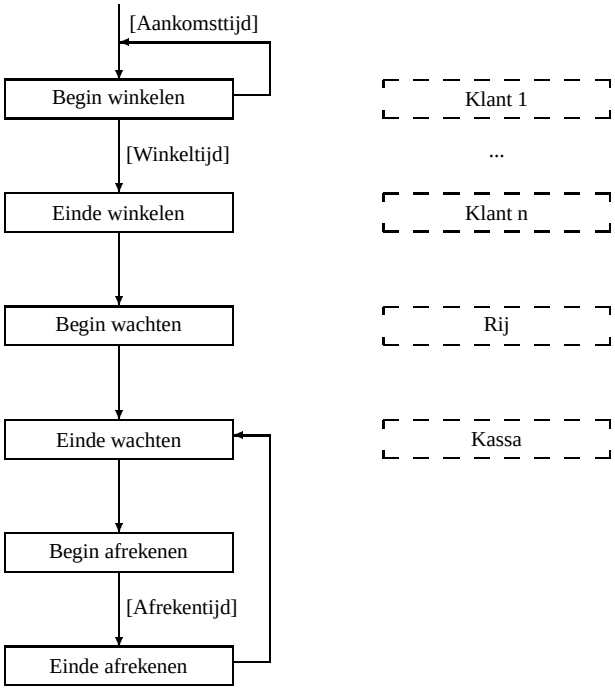

Figure 2.2: Event-scheduling

Het scheduling-proces kan enigzins vereenvoudigd worden door events die steeds gekoppeld optreden, samen te bundelen. In het voorbeeld van het warenhuis kan dit gebeuren door het einde van het winkelen en het begin van het wachten samen te nemen, evenals het einde van het wachten en het begin van het afrekenen, aangezien deze beide paren events steeds samen optreden. De zes events uit Figuur 2.2 zijn nu gereduceerd tot vier events, zoals geïllustreerd in Figuur 2.3.

2.3.2 Activity-scanning

De activity-scanning approach legt de nadruk op de activiteiten in het systeem. Het model zal bestaan uit een lijst van routines die elk een activiteit beschrijven. Een verzameling condities specifieert wanneer een bepaalde routine moet opgestart worden, terwijl een andere verzameling condities het einde van de routine aangeeft. De levensloop van elke activiteit zal bij deze vorm van simulatie door een aparte tekstuele module beschreven worden.

De processor zal herhaaldelijk eenzelfde cyclus doorlopen. Deze cyclus bestaat uit drie fasen:

- In de eerste fase worden herhaaldelijk alle condities getest die het initiëren van een activiteit bepalen. De activiteiten die kunnen opgestart worden, worden ook effectief opgestart. Als na het één of meerdere malen testen van alle condities blijkt dat geen enkele activiteit meer kan opgestart worden, wordt overgegaan naar de tweede fase.

- In de tweede fase wordt de simulatie-klok ingesteld op het tijdstip van beëindiging van de eerstvolgende activiteit.

- In de derde en laatste fase worden alle condities getest die het beëindigen van de activiteiten

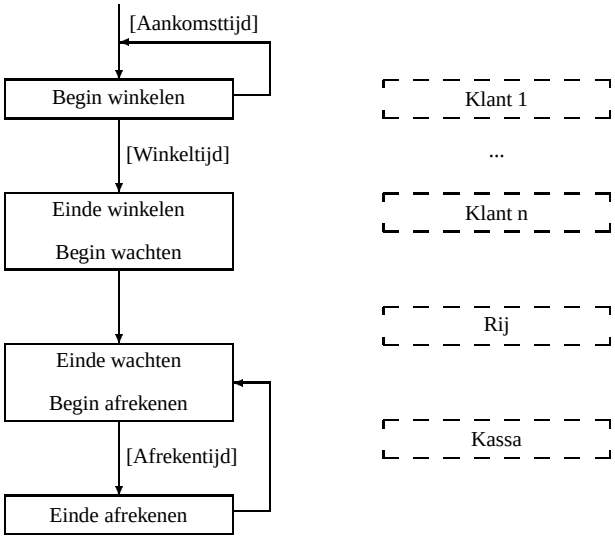

Figure 2.3: Vereenvoudigde event-scheduling

markeren. De activiteiten die aflopen worden beëindigd waarna terug wordt overgegeaan naar de eerste fase.

Gezien het testen van condities het belangrijkste aspect is van de activity-scanning approach, wordt deze methode als *conditional sequencing* gecatalogeerd.

In Figuur 2.4 wordt de activity-scanning techniek toegespitst op het warenhuis-systeem. Er zijn drie activiteiten binnen het systeem: het winkelen, het wachten en het afrekenen. Het winkelen begint als een klant de winkel betreedt en wordt afgesloten als de winkeltijd verstreken is, waarna het wachten kan opgestart worden. Dit wachten wordt afgesloten als de kassa vrij is. Afsluiten van het wachten leidt onmiddellijk tot het opstarten van het afrekenen. Uiteindelijk wordt het afrekenen besloten na verstrijken van de afrekentijd.

2.3.3 Process-interaction

De process-interaction aanpak richt zich op de stroom van entiteiten doorheen het systeem. Deze strategie beschouwt het systeem als een web van concurrente, interagerende processen. De mobiele entiteiten die eenzelfde proces ondergaan, worden verdeeld in klassen. Bij elke entiteitsklasse, hoort een procesklasse. De procesklasse beschrijft de levensloop van de entiteiten deel uitmakend van de geassocieerde entiteitsklasse. Een proces kan op meerdere plaatsen interageren met de omgeving. Bij process-interaction simulatie worden de processen beschreven door afzonderlijke tekstuele modules.

Ook bij een process-interaction aanpak moet de processor een lijst bijhouden. De items die deel uitmaken van deze lijst zullen bestaan uit een verwijzing naar een process, de tijd waarop het proces moet geactiveerd worden en de laatste toestand van het proces. Een proces zal immers niet zonder onderbreking van start tot einde doorlopen. Er kunnen allerhande situaties optreden waardoor het proces wordt geblokkeerd. De processor zal dan zijn aandacht verleggen naar andere processen totdat de oorzaken van de blokkering van het eerste proces zijn vervallen. Process-interaction simulatie is evenals event-scheduling simulatie een vorm van imperative sequencing.

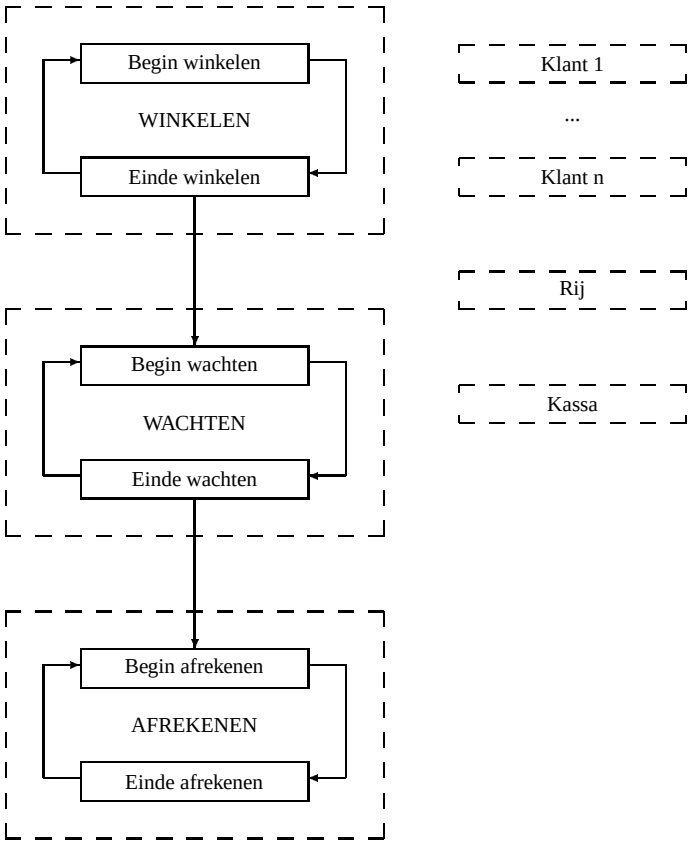

Figure 2.4: Activity-scanning

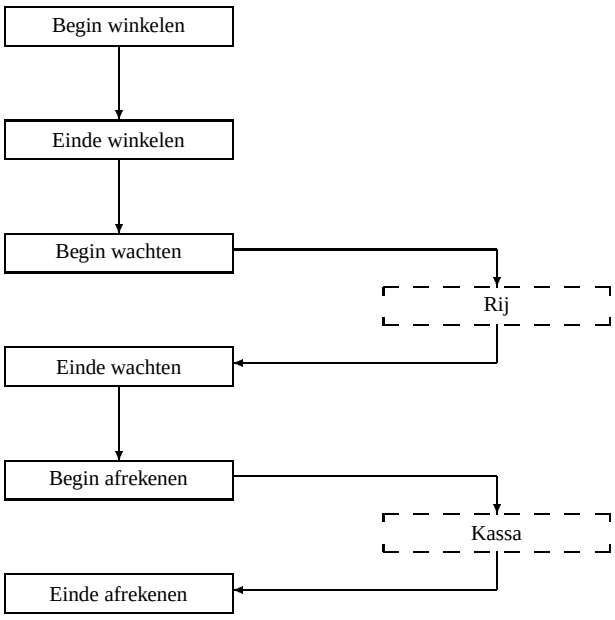

Figure 2.5: Process-interaction

In Figuur 2.5 wordt opnieuw het warenhuis-systeem beschouwd. Het systeem bestaat slechts uit één proces. In tegenstelling tot de vorige figuren wordt hier niet het scheduling-proces verduidelijkt, maar de entiteitsstroom. Met entiteitsstroom wordt in dit geval de stroom van de klanten bedoeld, aangezien de rij en de kassa niet mobiel zijn. In de figuur zijn de klanten niet expliciet weergegeven, maar zijn de rij en de kassa wel expliciet in de klantenstroom opgenomen.

2.3.4 Vergelijking

Twee criteria zijn van primordiaal belang bij het bepalen van de world view die in een bepaalde applicatie aangewezen zal zijn:

- De performantie van de uiteindelijke simulator. De uitvoeringssnelheid zal kenmerkend zijn voor deze performantie.
- Het modelleringsgemak, met andere woorden het gemak van vertaling van de systeemspecificaties naar modelspecificaties.

Het is intuïtief aan te voelen dat deze beide eigenschappen moeilijk te verzoenen zullen zijn. Een world view die het eerste criterium bevoordeeld, zal dit niet kunnen doen zonder afbreuk te doen aan het tweede criterium, en omgekeerd.

Toepassing van de event-scheduling approach zal resulteren in een performante simulator. Het op een natuurlijke wijze noteren van de systeemeigenschappen binnen het formalisme is echter sterk afhankelijk van de mate van interactie tussen de verschillende entiteiten binnen het systeem. Als er nauwelijks interactie is, zal de event-scheduling techniek attractief zijn, in het andere geval beslist niet.

De process-interaction approach situeert zich aan de andere zijde van het spectrum. Zelfs bij een grote mate van interactie tussen de enteiten, zal het model op een natuurlijke wijze tot stand kunnen

komen, dit echter ten nadele van de performantie. De process-interaction techniek wordt gezien de huidige stand van de technologie algemeen als de beste aanvaard.

De activity-scanning approach situeert zich tussen beide vorige technieken. De techniek is als compromis te halfslachtig en wordt weinig gebruikt.

Voorbeelden van aanwending van het event-scheduling concept zijn de simulatie-talen *SIMSCRIPT* en *GASP*. *ECSL* gebruikt activity-scanning terwijl *GPSS* en *SIMULA* voorbeelden zijn van gebruik van process-interaction.

2.4 Discrete-event simulatie in de praktijk

Alhoewel discrete-event simulatie kan ingezet worden binnen een waaier van mogelijkheden, blijkt het bestuderen van wachtlijnproblemen gebruik makend van discrete-event methodes, populair te zijn. Het doel van dergelijke studies is het nagaan van de effecten van gelimiteerde *resources* en *routing strategieën* op een entiteitsstroom. In principe kan de wachtlijntheorie ingezet worden voor de oplossing van dergelijke problemen. Deze theorie loopt echter spaak bij problemen met een bepaalde graad van complexiteit.

Dikwijls worden stochastische modellen gebruikt. Deze modellen brengen uitkomst in die gevallen waarin de exacte relaties tussen grootheden niet of slechts bij benadering gekend zijn. Als wel distributies gekend zijn, kan gebruik gemaakt worden van *distributie-sampling.*

Elke discrete-event simulatie blijkt volgende componenten te bevatten:

- Een methode voor het specifiëren van de structuur van het model.
- Een simulatie-klok.
- Methodes voor het introduceren van willekeur binnen het model met het oog op het modelleren van stochastische verschijnselen.
- Faciliteiten voor het vergaren, opslaan en verwerken van gegevens vermits het uiteindelijk de bedoeling is om informatie uit de simulatie te extraheren.
- Een processor die instaat voor scheduling en het beheer van event lists.

Chapter 3

GPSS

3.1 Historiek

GPSS [Schriber 1974, Neelamkavil 1987, Gordon 1978, Siemens 1979] is een programmeertaal, vertaler en bijbehorende run-time omgeving voor het bouwen en simuleren van discrete-event simulatiemodellen. Het dynamische gedrag van systemen, evoluerend in de tijd wordt gereproduceerd door gebruik te maken van de process-interaction filosofie. De taal werd oorspronkelijk ontwikkeld door Gordon in opdracht van IBM in 1962. GPSS werd geconcipieerd met het oog op gebruik door personen zonder specialisatie in de manipulatie van computers. Na een aantal versies van de taal werd in 1967 GPSS/360 geïntroduceerd. De naam werd veranderd van "General-Purpose System Simulator" naar "General-Purpose Simulation System". Later werd GPSS-V uitgebracht. Deze taal is een superset van GPSS/360 en heeft als belangrijkste additionele mogelijkheden de interface met FORTRAN en PL/I, en een vrijere codeervorm. Van de eerste versies van GPSS werden ook talen afgeleid door van IBM onafhankelijke onderzoekscentra, zoals NGPSS/6000, GPSS/NORDON, GPSS/UCC, GPSSTS, GPSS-V/6000 en GPSS-10. Recenter werd GPSS/H geïntroduceerd. Deze implementatie heeft als belangrijkste eigenschap dat programma's volledig gecompileerd worden, dit in tegenstelling met andere versies die na een precompilatiefase, gebruik maken van een interpreter. Het grote aantal verschillende implementaties heeft ervoor gezorgd dat GPSS steeds meer ingeburgerd is geraakt en op vrijwel elk belangrijk computersysteem beschikbaar is. Opmerkelijk is dat GPSS nog steeds populair is ondanks de vroege datum van conceptie.

3.2 Beschrijving

3.2.1 Inleiding

In deze sectie wordt een beknopte beschrijving gepresenteerd van GPSS/360, de versie van GPSS die het uitgangspunt van het HGPSS-systeem vormt. De beschrijving heeft tot doel een globaal overzicht van GPSS/360 te geven voor diegenen die al min of meer vertrouwd zijn met GPSS. Er werd op geen enkel gebied naar volledigheid gestreeft. Ter eerste kennismaking kan [Gordon 1978] of [Neelamkavil 1987] geraadpleegd worden. Een uitgebreide beschrijving van het systeem is te vinden in [Schriber 1974]. Telkens in hetgeen hierna volgt naar GPSS wordt gerefereerd, wordt hiermee impliciet GPSS/360 bedoeld.

Positie	Beschrijving
1	Een * in dit veld declareert de rest van de lijn als commentaar.
2-6	Verplicht of optioneel label, naargelang de aard van het statement.
8-18	Uit te voeren operatie.
19-71	Operanden, eventueel gevolgd door commentaar, gescheiden van de operanden door één of meerdere spaties.

Table 3.1: Velden ponskaart

3.2.2 Elementen van GPSS

GPSS laat zoals de andere discrete-event simulatie-talen toe om toestandsveranderingen van entiteiten in functie van de tijd te modelleren en te simuleren. Deze toestandsveranderingen komen tot stand door het ingrijpen van allerhande acties op de entiteiten. GPSS levert een aantal primitieve klassen voor het modelleren van entiteiten en acties. Een programma bestaat erin een aantal vertegenwoordigers van primitieve entiteitsklassen, acties in een bepaalde sequentie te laten ondergaan. De primitieve entiteiten worden binnen GPSS simpelweg *entiteiten* genoemd en de acties *blokken*. Entiteiten afkomstig van één van de klassen beschikken over een aantal attributen. De waarden van deze attributen kunnen door de modelbouwer gespecifieerd worden. De verzameling entiteitsklassen kan echter niet worden uitgebreid. Ook de primitieve acties zijn voorzien van een aantal parameters die op de gewenste waarde kunnen ingesteld worden. Uitbreiding van de verzameling primitieve acties is evenmin mogelijk. Zowel de primitieve entiteits- als actieklassen zijn weldoordacht gekozen, zodat een brede waaier van systemen op een eenvoudige en snelle manier te modelleren zijn.

Naast acties en entiteiten zijn er binnen een discrete-event systeem ook relaties tussen de entiteiten. Om deze relaties te kunnen modelleren is een vorm van communicatie tussen de entiteiten nodig. Deze communicatie behelst dan vooral het opvragen van de waarden van attributen van andere entiteiten. Binnen GPSS is een gestandardiseerde manier van gegevensuitwisseling voorzien, namelijk door het gebruik van zogenaamde *standard numerical attributes* of afgekort, *SNA's*.

Gezien de process-interaction approach wordt aangewend, kunnen processen concurrent verlopen. Ter ondersteuning van deze concurrentie is een geschikt scheduling-mechanisme vereist. GPSS levert dit mechanisme in de vorm van de processor. Deze processor handelt de eigenlijke simulatie af binnen het systeem.

3.2.3 Vorm van een programma

De vorm waarin een GPSS-programma moet neergeschreven worden is gezien de gevorderde leeftijd van het systeem, verouderd. De codeervorm is volledig geïnspireerd door het gebruik van *ponskaarten.* Elk GPSS-statement neemt exact één lijn in beslag. Een lijn is verdeeld in vier velden waarin gepaste informatie moet gespecifieerd worden (Tabel 3.1).

De GPSS-statements kunnen in drie categorieën verdeeld worden:

- declaraties van blokken,
- declaraties van entiteiten en
- commando's.

3.2.4 Entiteiten

Entiteiten afgeleid van eenzelfde klasse kunnen onderscheiden worden door een unieke *naam*. Deze naam kan een identificatienummer of een karaktersliert zijn. Indien mogelijk worden entiteiten automatisch gecreëerd als ze een eerste maal aangesproken worden. Indien een automatische creatie niet mogelijk is omdat bepaalde attributen een specifieke waarde moeten krijgen, moet de entiteit door middel van een speciaal statement gedeclareerd worden. Een dergelijk statement bestaat uit een verplicht *label*, een sleutelwoord en een aantal parameters. Het label bepaalt de naam van de entiteit en kan een identificatienummer of een karaktersliert van maximaal vijf karakters zijn. Als voor de hieronder besproken entiteitsklassen een declaratie-statement vereist is, wordt de vorm van dit statement in tabelvorm[1] aangegeven. In de tabellen wordt de betekenis gegeven van de statement-parameters aangeduid door de letters A tot G.

Binnen GPSS kunnen twee belangrijke groepen entiteitsklassen worden onderscheiden:

- *Transacties* zijn die entiteiten die zich doorheen het te modelleren systeem voortbewegen. Entiteiten van deze klasse zullen informatie transporteren en successieve toestandsveranderingen ondergaan. Het is de rol van de processor om alle transacties binnen het systeem voort te bewegen binnen het proces waarin ze zich bevinden. Aangezien GPSS geconcipieerd is om te werken op een seriële computer, kan de processor zich terzelfdertijd slechts over één transactie ontfermen. Deze transactie is de *actieve transactie.* Een transactie blijft actief totdat ze vrijwillig de processor vrijgeeft of ertoe genoodzaakt is. Een transactie bezit een aantal voor de modelbouwer belangrijke attributen:

  - De *parameters* zijn een aantal attributen in de vorm van een één-dimensionale matrix. De elementen van de matrix kunnen enkel numerieke waarden aannemen. De betekenis van deze parameters binnen het model wordt bepaald door de modelbouwer.
  - Het tijdstip waarop een transactie het systeem voor de eerste maal betreedt, wordt vastgelegd in een specifiek attribuut.
  - Elke transactie bezit een *prioriteit.* Deze prioriteit is van belang wanneer twee transacties op hetzelfde moment hun weg door het systeem willen verder zetten.

  Een transactie moet niet gedeclareerd worden.

- Naast de transacties zijn er een aantal andere, niet-mobiele entiteitsklassen. Entiteiten behorende tot deze klassen zijn statisch met betrekking tot hun locatie binnen het systeem.

  - Een eerste categorie niet-mobiele entiteitsklassen zijn deze die kunnen gebruikt worden voor het modelleren van resources met een gelimiteerde capaciteit:
    - *Facilities* kunnen aangewend worden om een resource met enkelvoudige capaciteit te modelleren. Naast benutting kan bij een facility ook verdringing optreden. In het laatste geval wordt de facility afhandig gemaakt van de transactie die ervan gebruik maakt en wordt ze toegekend aan een andere transactie die aan bepaalde voorwaarden voldoet. Een facility moet niet gedeclareerd worden.
    - *Storages* worden gebruikt om resources te modelleren waarvan de capaciteit kan gespecifieerd worden. Een verzameling gelijkwaardige resources met enkelvoudige capaciteit kan ook door een storage worden gemodelleerd. Storages moeten gedeclareerd worden met behulp van het STORAGE-statement (Tabel 3.2).

[1] Dergelijke tabellen zijn ook in [Schriber 1974] opgenomen.

Parameter	Significance
A	The capacity of the storage

Table 3.2: STORAGE-declaratie

Parameter	Significance
A	H or X depending on whether the matrix is to consists of halfword or fullword memory locations, respectively.
B	A constant indicating the number of rows in the matrix.
C	A constant indicating the number of columns in the matrix.

Table 3.3: MATRIX-declaratie

– Als hulpmiddelen bij het behandelen van wachtlijnproblemen zijn een aantal entiteitsklassen voorhanden. Entiteiten van deze klassen hebben allen de vorm van een keten. De schakels van de keten zijn transacties. Ketens kunnen op de volgende manier gemodelleerd worden:
  * Een *queue* is een wachtlijn die door de GPSS-processor wordt beheerd. Binnen het model zal deze entiteit dus niet expliciet moeten worden gemanipuleerd. De gebruikte *queuing discipline* is steeds FIFO (First In First Out). Een queue moet niet gedeclareerd worden.
  * *User chains* vertonen gelijkenissen met queues. Transacties moeten echter expliciet in de keten geplaatst en eruit verwijderd worden. In tegenstelling met queues kunnen andere queuing disciplines dan FIFO gebruikt worden. User chains worden niet gedeclareerd.
  * *Matching chains* tenslotte hebben een eerder gespecialiseerd nut. Ze worden niet gebruikt voor het bestuderen van de wachttijd maar voor het groeperen van een aantal transacties die aan bepaalde voorwaarden voldoen. Matching chains worden eveneens niet gedeclareerd.
– Doorheen het verloop van een simulatie zullen typisch bepaalde resultaten gegenereerd worden die moeten vastgehouden worden voor latere verwerking. Een aantal entiteitsklassen voor opslag van gegevens staan ter beschikking van de gebruiker:
  * Een *logic switch* is een entiteit die een booleaanse waarde kan opslaan en wordt niet gedeclareerd.
  * Een *savevalue* daarentegen laat toe om een numeriek resultaat vast te houden. Savevalues worden evenmin gedeclareerd.
  * *Matrix savevalues* zijn twee-dimensionale matrices van enkelvoudige savevalues waarbij elk element afzonderlijk kan aangesproken worden. Matrix savevalues moeten door het MATRIX-statement (Tabel 3.3) gedeclareerd worden.
– Hulpmiddelen voor het uitvoeren van berekeningen binnen een model staan in de vorm van volgende klassen ter beschikking:
  * Een *variable* zorgt niet zoals de naam het laat vermoeden voor de opslag van een waarde, maar laat toe om een bewerking te speficiëren om deze één of meerdere malen uit te voeren. Bij het aanspreken van een variable zal de bewerking worden uitgevoerd waarna het resultaat beschikbaar wordt voor verdere verwerking. Een variable werkt in op numerieke data en moet gedeclareerd worden door het VARIABLE-statement (Tabel 3.4).

Parameter	Significance
A	The arithmetic expression which defines the arithmetic variable.

Table 3.4: VARIABLE-declaratie

Parameter	Significance
A	The fullword arithmetic expression which defines the fullword arithmetic variable.

Table 3.5: FVARIABLE-declaratie

  - *Fullword variables* zijn identiek aan variables maar leveren een fullword resultaat. Het FVARIABLE-statement (Tabel 3.5) wordt gebruikt ter declaratie.
  - *Boolean variables* zijn te vergelijken met variables maar hebben booleaanse waarden als argumenten en leveren ook een booleaans resultaat. Declaratie gebeurt met het BVARIABLE-statement (Tabel 3.6).
  - *Functies* kunnen beschouwd worden als tabellen bestaande uit punten gelegen in een twee-dimensionaal vlak. Deze verzameling punten kan op een aantal manieren geïnterpreteerd worden. Een veelgebruikte interpretatie is deze waarbij de ligging van de punten en hun volgorde een bepaalde mathematische functie benaderen. Het FUNCTION-statement (Tabel 3.7) declareert een functie.

- *Random number generators* kunnen gebruikt worden bij het construeren van stochastische modellen en moeten niet gedeclareerd worden.
- *Tables* tenslotte zijn data-collectoren. Alhoewel het GPSS-systeem automatisch informatie vergaart, kan bepaalde relevante informatie buiten de automatische vergaring vallen. In dit geval kunnen tables ingeschakeld worden. Door gebruik te maken van het TABLE-statement (Tabel 3.8) kan een table gedeclareerd worden. Een table kan ook verbonden worden met een queue. Informatie automatisch vergaard door de queue wordt dan doorgespeeld naar de table om aldaar opgeslagen te worden. Een dergelijk verbinding kan gedeclareerd worden door het QTABLE-statement (Tabel 3.9).

3.2.5 Processor

De processor kan beschouwd worden als de orchestrator van het simulatie-gebeuren. De voornaamste taken van de processor zijn:

- Voortbewegen van transacties doorheen het model door het schedulen van events.
- Bijhouden van de simulatie-tijd.
- Nagaan wanneer de simulatie mag beëindigd worden.

Parameter	Significance
A	The boolean expression which defines the boolean variable.

Table 3.6: BVARIABLE-declaratie

Parameter	Significance
A	The argument of the function.
B	Dn, Cn, En, Ln, Mn where n is the number of different function points.

Table 3.7: FUNCTION-declaratie

Parameter	Significance
A	The name of the random variable whose values are to be entered in the table. In particular, the A-operand will be the name of some standard numerical attribute.
B	The first boundary point.
C	The width of each intermediate table interval.
D	The total number of intervals in the table, including the leftmost and rightmost.
E	Optional time interval for RT tables.

Table 3.8: TABLE-declaratie

Bij het schedulen van events worden event notices gebruikt. Deze worden bijgehouden door gebruik te maken van niet één, maar twee event lists:

- De eerste lijst wordt de *current event chain* genoemd. De lijst wordt gebruikt voor het bijhouden van events die plaatsvinden op het tijdstip aangeduid door de actuele waarde van de simulatie-klok. Elk event staat in rechtstreeks verband met één welbepaalde transactie. De events op de current event chain zijn geordend naargelang de prioriteit van de gerefereerde transacties. De events waarvan de prioriteit van de transacties de grootste is, bevinden zich vooraan in de keten.

- De *future event chain* bevat events die gescheduled zijn voor tijdstippen in de toekomst. De lijst is geordend op tijd. Events het verst in de toekomst gescheduled bevinden zich achteraan.

De processor vericht zijn taak door de current event chain herhaaldelijk af te lopen en alle events - indien mogelijk - uit te voeren. Uitvoering van een event kan tot gevolg hebben dat nieuwe events op de current event chain worden gescheduled. Als de current event chain doorlopen werd zonder dat ook maar één enkel event tot uitvoering werd gebracht, wordt de simulatie-tijd opgeschoven naar het tijdstip aangegeven door het eerste event op de future event chain. Alle events op de future event chain gescheduled op dit tijdstip worden overgeheveld van de future event chain naar de current event chain en het hele scenario wordt hervat.

De processor houdt door middel van twee klokken de simulatie-tijd bij. De *absolute klok* is een klok die niet kan teruggedraaid worden en duidt de simulatie-tijd aan sinds het begin van de eerste simulatie-taak. Een volledige simulatie-sessie kan bestaan uit meerdere simulatie-taken. De *relatieve klok* houdt de simulatie-tijd bij sinds de aanvang van de laatste taak binnen de simulatie-sessie. Deze klok wordt bij de aanvang van elke nieuwe taak teruggedraaid tot de initiële positie.

Parameter	Significance
A	Name of queue.
B	The first boundary point.
C	The width of each intermediate table interval.
D	The total number of intervals in the table, including the leftmost and rightmost.

Table 3.9: QTABLE-declaratie

Een simulatie-taak kan automatisch ten einde lopen of kan expliciet gestopt worden. De taak loopt automatisch ten einde als er geen events meer zijn om verwerkt te worden. In het andere geval zorgt de *beëindigingsteller* ervoor dat de taak wordt afgebroken. Deze beëindigingsteller wordt bij de aanvang van de taak op een door de gebruiker gespecifieerde waarde ingesteld en kan vanaf dan enkel maar gedecrementeerd worden. Deze decrementering kan gebeuren telkens een transactie uit het model wordt verwijderd. De simulatie-taak wordt afgesloten als de beëindigingsteller op nul komt te staan.

3.2.6 Blokken

Acties worden uitgevoerd op entiteiten door geparameteriseerde blokken, afgeleid van bepaalde klassen. Deze blokken moeten aan elkaar gekoppeld worden in de vorm van een netwerk overeenkomstig de stroom van transacties doorheen het systeem. Elk blok zal dan een lokaal uit te voeren actie modelleren. De actie geassocieerd met een blok zal tot uitvoering worden gebracht als een transactie het blok betreedt. Een actie kan inwerken op entiteiten of de processor, maar kan ook de transactie-stroom manipuleren. Een combinatie van deze aspecten is eveneens mogelijk. De meeste blokken zijn *instantaan.* Hiermee wordt bedoeld dat de uitvoering van de actie met het blok geassocieerd, geen simulatie-tijd verbruikt. De meeste blokken zijn uitgerust met een aantal parameters. Niet elk blok telt evenveel parameters. Voor sommige parameters bestaat een verplichting tot het specifiëren van een waarde, terwijl andere parameters facultatief met een waarde kunnen toebedeeld worden. Indien voor deze laatste categorie parameters, geen waarde wordt gespecifieerd, zal een waarde bij verstek worden aangenomen.

Een blok wordt gedeclareerd door een statement beginnend met een optioneel label, gevolgd door een sleutelwoord en een parameter-lijst. De sleutelwoorden komen overeen met de namen van de blokklassen. Het label mag niet numeriek zijn.

In volgend overzicht worden de GPSS-blokken kort besproken. Tevens werd voor elk blok een tabel[2] opgenomen die de vorm van het corresponderende blokdeclaratie-statement verduidelijkt. Elke tabel geeft de betekenis van de parameters en hun eventuele waarde bij verstek. Met een X worden zogenaamde hulpoperatoren aangeduid. Deze operatoren zijn geen echte parameters maar worden in een statement wel ná het sleutelwoord opgenomen. Tussen de eigenlijke parameters en de hulpoperator bevindt zich geen comma.

GENERATE en TERMINATE

Deze blokken markeren het begin en einde van de levensloop van een transactie. In een GENERATE-blok (Tabel 3.10) worden transacties vanuit het niets gegenereerd. Dit blok is het enige zonder ingang. Een aantal attributen van de gegenereerde transacties kunnen worden ingesteld op een gewenste waarde.

Het TERMINATE-blok (Tabel 3.11) verwijdert transacties uit het systeem. Bij elke verwijdering kan de beëindigingsteller met een gespecifieerde waarde worden gedecrementeerd.

ADVANCE

Dit blok (Tabel 3.12) is het enige niet-instantane van alle GPSS-blokken. De uitvoering van de actie geassocieerd met het blok verbruikt met andere woorden simulatie-tijd. Een transactie die het blok betreedt wordt onvoorwaardelijk gedurende een gespecifieerde tijd op inactief geplaatst.

[2] Dergelijke tabellen zijn ook in [Schriber 1974] opgenomen.

Parameter	Significance	Default value
A	Average interarrival time.	Zero
B	Half-width of range over which interarrival time is uniformely distributed.	Zero
C	Offset interval	No offset in effect
D	Limit count	Infinity
E	Priority level	Zero
F	Number of parameters	?
G	Type of parameters	Fullword

Table 3.10: GENERATE-blok

Parameter	Significance	Default value
A	Termination counter decrement.	Zero

Table 3.11: TERMINATE-blok

SEIZE en RELEASE

Een facility kan door een transactie worden bezet door de facility vast te grijpen via een SEIZE-blok (Tabel 3.13). Als de bewuste facility reeds bezet is, zal de toegang tot het SEIZE-blok aan geïnteresseerde transacties ontzegd worden.

Een facility bezet door een transactie kan door die transactie terug vrijgegeven worden via het RELEASE-blok (Tabel 3.14). Eén van de eventuele transacties wachtend vóór een SEIZE-blok met de bedoeling de facility vast te grijpen zal tot het SEIZE-blok worden toegelaten.

ENTER en LEAVE

Analoog aan de combinatie SEIZE-RELEASE is de combinatie ENTER-LEAVE. Het ENTER-blok (Tabel 3.15) laat transacties toe een aantal eenheden van een storage op te eisen. Of het gevraagde aantal eenheden kan toegekend worden, hangt af van de nog beschikbare capaciteit van de storage.

Een transactie die een aantal eenheden van een storage in bezit heeft, kan deze eenheden terug vrijgeven via een LEAVE-blok (Table 3.16). Als na het vrijgeven van de eenheden, een andere transactie kan voorzien worden van het aantal eenheden van de storage waar de transactie om vraagt, zal de transactie tot het ENTER-blok worden toegelaten vóór hetwelk ze werd opgehouden.

QUEUE en DEPART

Als omwille van een blokkerende conditie op een bepaalde plaats in het model, transacties belet worden hun weg voort te zetten totdat de blokkerende conditie verdwenen is, zal er op die plaats een wachtlijn onstaan. Gegevens omtrent deze wachtlijn kunnen enkel verzameld worden als de locaties waar de

Parameter	Significance	Default value
A	Average service time.	Zero
B	Half-width of range over which holding time is uniformely distributed.	Zero

Table 3.12: ADVANCE-blok

Parameter	Significance	Default value
A	The name (numeric or symbolic) of the facility to be seized.	-

Table 3.13: SEIZE-blok

Parameter	Significance	Default value
A	The name (numeric or symbolic) of the facility to be released.	-

Table 3.14: RELEASE-blok

Parameter	Significance	Default value
A	The name (numeric or symbolic) of the storage to be captured.	-
B	The number of servers to be captured.	1

Table 3.15: ENTER-blok

Parameter	Significance	Default value
A	The name (numeric or symbolic) of the storage to be released.	-
B	The number of servers to be released.	1

Table 3.16: LEAVE-blok

Parameter	Significance	Default value
A	The name (numeric or symbolic) of the queue to be joined.	-
B	Number of units by which the recorded content of the queue is to be modified.	1

Table 3.17: QUEUE-blok

Parameter	Significance	Default value
A	The name (numeric or symbolic) of the queue to be departed.	-
B	Number of units by which the recorded content of the queue is to be modified.	1

Table 3.18: DEPART-blok

wachtlijn aangevat en verlaten wordt, expliciet worden aangeduid. Het QUEUE-blok (Tabel 3.17) duidt het begin van de wachtlijn aan terwijl het DEPART-blok (Tabel 3.18) het einde ervan markeert.

ASSIGN, MARK en PRIORITY

Aangezien simulatie als doel heeft om de verandering van de attributen van entiteiten te onderzoeken en transacties een belangrijke categorie entiteiten zijn, moeten er een aantal blokken zijn die rechtstreeks inwerken op de attributen van transacties.

Het ASSIGN-blok (Tabel 3.19) laat toe om waarden toe te kennen aan de parameters van een transactie.

Het PRIORITY-blok (Tabel 3.20) verandert de prioriteit van een transactie.

Het MARK-blok (Tabel 3.21) laat toe om transacties die het blok betreden te markeren met de huidige waarde van de absolute klok. Ofwel wordt de waarde van de absolute klok gekopieerd in één van de parameters van de transactie, ofwel wordt het transactie-attribuut dat het tijdstip van creatie aangeeft door de waarde van de absolute klok overschreven.

LOGIC, SAVEVALUE, MSAVEVALUE

Gebruik van het LOGIC-blok (Tabel 3.22) laat toe om de waarde van logic switches in te stellen en te veranderen.

Voor het aanpassen van de waarden van savevalues, is het gebruik van het SAVEVALUE-blok (Tabel 3.23) noodzakelijk. Het MSAVEVALUE-blok (Tabel 3.24) heeft een analoge functie maar werkt in op elementen van een matrix savevalue.

Parameter	Significance	Default value
A	Number of the parameter to be modified.	-
B	Data to be used for the modification.	-

Table 3.19: ASSIGN-blok

Parameter	Significance	Default value
A	Value to be assigned as the priority level of transactions which enter the PRIORITY block.	-
B	BUFFER: when immediate rescan of the current events chain is required.	No immediate rescan

Table 3.20: PRIORITY-blok

Parameter	Significance	Default result
A	Number of the parameter into which the absolute clock's value is to be copied.	Transaction's time of model entry is overwritten by absolute clock's value.

Table 3.21: MARK-blok

Parameter	Significance	Default value
A	The name (numeric or symbolic) of a logic switch.	-
X	The auxiliary operator X indicates what is to be done to the indicated logic switch; the forms X can assume are shown below: R: Reset S: Set I: Invert	-

Table 3.22: LOGIC-blok

Parameter	Significance	Default value
A	Number or symbolic name of the savevalue to be modified.	-
B	Data to be used in the modification process.	-
C	Specifies whether the savevalue involved is a halfword or fullword type; the character H designates a halfword type; defaulting on the C-operand implies that a fullword savevalue is being referenced.	A fullword savevalue is implied.

Table 3.23: SAVEVALUE-blok

Parameter	Significance	Default value
A	Name (numeric or symbolic) of the matrix in which an element is to be modified.	-
B	Row subscript.	-
C	Column subscript.	-
D	Data to be used in the modification process.	-
E	The character H indicates that the matrix involved is of the halfword type; if a fullword matrix is intended, the E-operand is left blank.	A fullword matrix is implied.

Table 3.24: MSAVEVALUE-blok

Parameter	Significance	Default value
A	Name (numeric or symbolic) of a logic switch.	-
X	Auxiliary operator, termed a logical mnemonic, indicating the switch setting which is required for the test to be true; the two logical mnemonics for the logic switches are shown below: LS: Test for set condition LR: Test for reset condition	-
B	Optional operand; block location to which the testing transaction moves if the logic switch is not in the condition required for the test to be true.	Test is conducted in refusal mode when no B-operand is provided.

Table 3.25: Eerste vorm GATE-blok

Parameter	Significance	Default value
A	Location of ASSEMBLE, GATHER, or MATCH block.	-
X	Auxiliary operator, termed a logical mnemonic, indicating the matching condition which is required for the test to be true; the two logical mnemonics for the matching conditions are shown below: M: Test for another assembly set member in matching condition in the A-block. NM: Test for no other assembly set member in matching condition in the A-block.	-
B	Optional operand; block location to which the testing transaction moves if the block is not in the condition required for the test to be true.	Test is conducted in refusal mode when no B-operand is provided.

Table 3.26: Tweede vorm GATE-blok

BUFFER

Het BUFFER-blok, waarvoor geen enkele parameter kan gespecifieerd worden, heeft slechts een beperkt nut. Bij het betreden van het blok door een transactie zal een onmiddellijke rescan van de current event chain worden geïnitieerd.

GATE

GATE-blokken laten toe om de toegang tot het sequentiële blok af te schermen als niet aan een bepaalde conditie wordt voldaan. Tegengehouden transacties worden dan ofwel naar een derde blok gevoerd, ofwel wordt de test herhaald totdat de transactie zijn weg kan verder zetten naar het sequentiële blok. De conditie kan bepaald zijn door

- de stand van logic switches (Tabel 3.25),
- de aanwezigheid van een matching condition bij een MATCH-, GATHER- of ASSEMBLE-blok (Tabel 3.26),
- de toestand van een facility (Tabel 3.27) of
- de toestand van een storage (Tabel 3.28).

Parameter	Significance	Default value
A	Name (numeric or symbolic) of a facility.	-
X	Auxiliary operator, termed a logical mnemonic, indicating the facility status which is required for the test to be true; the logical mnemonics are shown below: U: Test for facility in use NU: Test for facility not in use I: Test for facility interrupted NI: Test for facility not interrupted	-
B	Optional operand; block location to which the testing transaction moves if the facility is not in the state required for the test to be true.	Test is conducted in refusal mode when no B-operand is provided.

Table 3.27: Derde vorm GATE-blok

Parameter	Significance	Default value
A	Name (numeric or symbolic) of a storage.	-
X	Auxiliary operator, termed a logical mnemonic, indicating the storage status which is required for the test to be true; the logical mnemonics are shown below: SF: Test for storage full SNF: Test for storage not full SE: Test for storage empty SNE: Test for storage not empty	-
B	Optional operand; block location to which the testing transaction moves if the storage is not in the state required for the test to be true.	Test is conducted in refusal mode when no B-operand is provided.

Table 3.28: Vierde vorm GATE-blok

Parameter	Significance	Default value or result
A	Name of the first standard numerical attribute.	-
B	Name of the second standard numerical attribute.	-
X	The auxiliary operator X represents the relational operator to be used in the test; the forms X can assume are shown below: G: Is A greater than B? GE: Is A greater than or equal to B? E: Is A equal to B? NE: Is A not equal to B? LE: Is A less than or equal to B? L: Is A less than B?	-
C	Optional operand; block location to which the testing transaction moves if the answer to the question implied by the relational operator is "no".	The test is conducted in refusal mode when no C-operand is provided.

Table 3.29: TEST-blok

Parameter	Significance	Default value
A	Number of a parameter.	-
B	Symbolic name of non-sequential block location	-

Table 3.30: LOOP-blok

TEST

Dit blok (Tabel 3.29) laat toe om uit twee paden, een pad via het sequentiële blok en een ander, één te kiezen. Langs het gekozen pad zal de transactie haar weg verder zetten. De keuze wordt bepaald door het resultaat van een relationele operatie op twee numerieke argumenten.

LOOP

Soms is het nuttig om een transactie een bepaald traject een aantal malen te laten afleggen. De test op het beëindigen van de iteratie en indien vereist, het terugvoeren naar het begin van de iteratielus kan worden bewerkstelligd door gebruik te maken van het LOOP-blok (Tabel 3.30). Dit blok decrementeert telkens de waarde van een bepaalde parameter van de transactie die het blok betreedt met één, vergelijkt deze waarde met nul, en voert de transactie eventueel terug naar een aangegeven locatie.

PRINT

Telkens een transactie het PRINT-blok (Tabel 3.31) betreedt, zal bepaalde informatie worden afgedrukt. De aard van deze informatie kan door enkele parameters worden gespecifieerd. Globaal gezien is er keuze uit informatie over entiteiten en informatie bijgehouden door de processor.

TRANSFER

Zoals het TEST-blok kan het TRANSFER-blok gebruikt worden om uit twee paden één te kiezen waarlangs de transactie haar weg zal verder zetten. De keuze uit de paden kan op drie wijzen gebeuren.

- Een *conditioneel* TRANSFER-blok (Tabel 3.32) controleert het begin van de gespecifieerde paden op een blokkerende conditie en stuurt de transactie het pad op waar geen blokkerende conditie aan-

Parameter	Significance	Default value or result
C	A mnemonic indicating the entity class of interest.	Fullword savevalues.
A and B	The smallest and largest numbers, respectively, of entity members for which information is to be output.	Information will be printed for all.
D	paging indicator.	Printing occurs at top of new page.

Table 3.31: PRINT-blok

Parameter	Significance	Default value
A	Literally the word BOTH.	-
B	A block location.	Sequential block.
C	Another block location.	-

Table 3.32: Conditioneel TRANSFER-blok

wezig is. Als op beide paden geen blokkering optreedt wordt één van de paden arbitrair gekozen. Als op beide paden blokkering optreedt wordt de test herhaald tot één van de blokkeringen verdwijnt.

- Een *statistisch* TRANSFER-blok (Tabel 3.33) stuurt een opgegeven percentage transacties het ene pad op en de rest langs het andere pad.
- Een *onconditioneel* TRANSFER-blok (Tabel 3.34) maakt geen keuze uit paden en stuurt een transactie steeds een welbepaald pad op.

TABULATE

Waarden kunnen ingevoerd worden in een data-collectie entiteit, een table, door gebruik te maken van het TABULATE-blok (Tabel 3.35).

SELECT

Het kan nuttig zijn om een aantal waarden aan een test te onderwerpen en de eerste die aan een bepaalde conditie voldoet te selecteren voor verdere verwerking. Het SELECT-blok is bruikbaar in dergelijke gevallen. Dit blok heeft drie vormen van voorkomen:

- Een *logisch* SELECT-blok (Tabel 3.36) kiest een entiteit die aan een bepaalde voorwaarde voldoet uit een aantal entiteiten van hetzelfde type.
- Een *minimum* of *maximum* SELECT-blok (Tabel 3.37) kiest uit een aantal waarden respectievelijk het kleinste of het grootste.

Parameter	Significance	Default value
A	Fraction of the time that entering transactions will transfer to the C-block.	-
B	A block location.	Sequential block.
C	Another block location.	-

Table 3.33: Statistisch TRANSFER-blok

Parameter	Significance	Default value
A	Not used.	-
B	Block to which transactions move next.	Sequential block.

Table 3.34: Onconditioneel TRANSFER-blok

Parameter	Significance	Default value
A	Name (numeric or symbolic) of the table into which a value is to be entered.	-
B	The number of times the value is to be entered in the table each time a transaction moves into the TABULATE block.	1

Table 3.35: TABULATE-blok

- Een *relationeel* SELECT-blok (Tabel 3.38) vergelijkt een aantal waarden met een andere waarde en selecteert de eerste waarde waarvoor een relationele conditie opgaat.

PREEMPT en RETURN

Een niet-bezette facility wordt normaal in gebruik genomen door een transactie als deze een SEIZE-blok binnendringt. Door het RELEASE-blok kan de facility weer vrijgegeven worden. Het PREEMPT-blok (Tabel 3.39) laat toe om een facility toe te wijzen aan een transactie zelfs als de facility reeds bezet is door een andere transactie. Deze *preempted* transactie verliest tijdelijk het bezit van de facility om ze later eventueel terug in handen te krijgen als de *preempting* transactie de facility terug vrijgeeft via een RETURN-blok (Tabel 3.40).

SPLIT en ASSEMBLE

Nieuwe transacties kunnen het model worden ingebracht door het GENERATE-blok. Er bestaat echter via het SPLIT-blok (Tabel 3.41) een andere wijze om dit te bewerkstelligen. In dit blok worden uitgaande van een binnentredende transactie een gespecifieerd aantal duplicaten gemaakt die verder als zelfstandige transacties een eigen levensloop zullen hebben. De afgeleide transacties blijven wel verbonden met hun voorouders door een intern attribuut: het *assembly set*-nummer. Een ASSEMBLE-blok (Tabel 3.42) laat het duale toe: een aantal tot eenzelfde assembly set behorende transacties worden gecombineerd tot één transactie.

GATHER

Een aantal transacties behorende tot een bepaalde assembly set accumuleren in een blok totdat het aantal geaccumuleerde transacties een bepaalde waarde bereikt heeft, waarna de transacties hun weg kunnen vervolgen, wordt verwezenlijkt door gebruik te maken van het GATHER-blok (Tabel 3.43).

MATCH

Als een transactie een MATCH-blok (Figuur 3.44) betreedt wordt nagegaan of in een ander, door een parameter gespecifieerd blok, een *matching*-operatie plaatsvindt. Als dit het geval is, kan de transactie haar

Parameter	Significance	Default value
A	Number of the parameter into which is to be copied the number of an entity member currently satisfying the stated condition.	-
B and C	The smallest and largest numbers, respectively, in the set of entity members to be scanned.	-
X	The logical mnemonic indicating the condition which must be satisfied; the available logical mnemonics are shown below: LS: Logic switch set LR: Logic switch reset U: Facility in use NU: Facility not in use I: Facility preempted NI: Facility not preempted SF: Storage full SNF: Storage not full SE: Storage empty SNE: Storage not empty	-
F	Optional operand; block location to which the selecting transaction moves if no entity member currently satisfies the required condition.	Selecting transaction moves to sequential block unconditionally.

Table 3.36: Logisch SELECT-blok

Parameter	Significance	Default value
E	The family name of the standard numerical attribute whose value is being investigated.	-
B and C	The inclusive lower and upper limits, respectively, of the range of members of the specific family members involved.	-
A	The number of a parameter into which is to be put the number of the family member whose attribute is the minimum or maximum.	-
X	The axiliary operator X is to be MIN or MAX, depending on whether the entity with the minimum or maximum attribute value is sought, respectively.	-

Table 3.37: Minimum of maximum SELECT-blok

Parameter	Significance	Default value
E	The family name of the standard numerical attribute being investigated.	-
B and C	The smallest and largest numbers, respectively, in the set of entity members subject to the scan.	-
D	The data against which the E-operand SNA is to be compared.	-
X	X is an auxiliary operator. It represents the relational operator which specifies the way in which the E-operand SNA is to be compared to the D-operand data. In practice, X takes one of the forms shown below: G: Is E greather than D? GE: Is E greater than or equal to D? E: Does E equal D? NE: Does E not equal D? LE: Is E less than or equal to D? L: Is E less than D?	-
A	Number of the parameter into which the number of an entity member currently satisfying the stated condition is to copied.	-
F	Optional operand; block to which the selecting transaction moves if no entity member currently satifies the indicated condition.	Selecting transaction moves to sequential block unconditionally.

Table 3.38: Relationeel SELECT-blok

Parameter	Significance	Default value
A	Name (numeric or symbolic) of the facility to be preempted.	-
B	Optional operand used to indicate the conditions under which preemption is to be permitted; there are two alternatives: Not used: A preempt occurs only if the current user is not himself a preempter. PR: A preempt occurs only if the would-be preempter has a higher priority level than the current user.	
C	The C-operand supplies the name of the block location to which the interrupted transaction will be sent.	
D	The D-operand's value is interpreted as the number of one of the interrupted transaction's parameters. The processor places a copy of the transaction's remaining facility holding time into this parameter.	
E	The E-operand indicates whether or not the interrupted transaction is to remain in contention for automatic reinstatement as the capturer of the facility. When E is the two-character sequence RE (for remove), the transaction is removed from contention. When the default is taken on the E-operand, the transaction remains in contention.	

Table 3.39: PREEMPT-blok

Parameter	Significance	Default value
A	Name (numeric or symbolic) of the facility to be disengaged.	-

Table 3.40: RETURN-blok

Parameter	Significance	Default value or result
A	The number of additional transactions to be brought into the model.	-
B	Name of the block location to which these additional transactions are to be sent.	-
C	Number of the serialization parameter.	No serialization occurs.
D	Number of parameters each offspring is to have.	Each offspring has the same number of parameters as the parent.

Table 3.41: SPLIT-blok

Parameter	Significance	Default value
A	Assembly count; indicates how many assembly set members are to be "combined" into a single transaction.	-

Table 3.42: ASSEMBLE-blok

Parameter	Significance	Default value
A	Gather count; indicates how many assembly set members are to be accumulated at the block.	-

Table 3.43: GATHER-blok

Parameter	Significance	Default value
A	Supplies the name of the location which the conjugate block occupies.	-

Table 3.44: MATCH-blok

weg verder zetten. In het andere geval blijft de transactie in het blok en wordt het blok in de matching-toestand geplaatst totdat het geconjugeerde blok ook in de matching-toestand komt. Een combinatie van MATCH-blokken kan dus gebruikt worden om transacties te synchroniseren.

3.2.7 Standard numerical attributes

De manier om de waarden van attributen en andere gegevens op te vragen en de opvraging te formuleren in GPSS is via standard numerical attributes. Door een mnemonic wordt aangegeven welke de klasse van de entiteit en het specifieke attribuut is waarin men geïnteresseerd is. Deze mnemonic wordt gevolgd door de naam van de entiteit. De naam kan een identificatienummer of een karaktersliert zijn. Niet voor alle standard numerical attributes is een naam vereist. Een bijkomende mogelijkheid is het gebruik van indirectie. Hierbij moet verplicht een identificatienummer gebruikt worden, voorafgegaan door een *. Het identificatienummer slaat op één van de parameters van de actieve transactie. De waarde van de parameter zal als identificatienummer van de entiteit gebruikt worden. Er is slechts één indirectieniveau toegelaten.

3.2.8 Grafische voorstelling

De specificatie van een model te gebruiken voor simulatie moet aan het GPSS-systeem worden aangeboden in de vorm van een tekstuele beschrijving. Het is echter gebruikelijk om een model ook door middel van een grafische beschrijving voor te stellen. In deze beschrijving wordt de transactiestroom doorheen het model op een overzichtlijke wijze gepresenteerd. Daartoe wordt het netwerk getekend opgebouwd uit de gebruikte blokken en hun interconnecties. Elk blok wordt voorgesteld door een gestandaardiseerd symbool aangevuld met de actuele waarde van de parameters geassocieerd met het blok. In Figuren 3.1 en 3.2 werden de symbolen gebruikt om de GPSS-blokken voor te stellen, opgenomen. De letters A tot G slaan op de blok-parameters.

3.2.9 Commando's

Alle tot nu toe behandelde statements stonden in verband met de declaratie van entiteiten of blokken. GPSS beschikt echter ook over een aantal commando-statements. De belangrijkste commando's zijn:

- CLEAR: Herinitialiseert het model om een nieuwe simulatie-taak te kunnen beginnen binnen de huidige sessie.

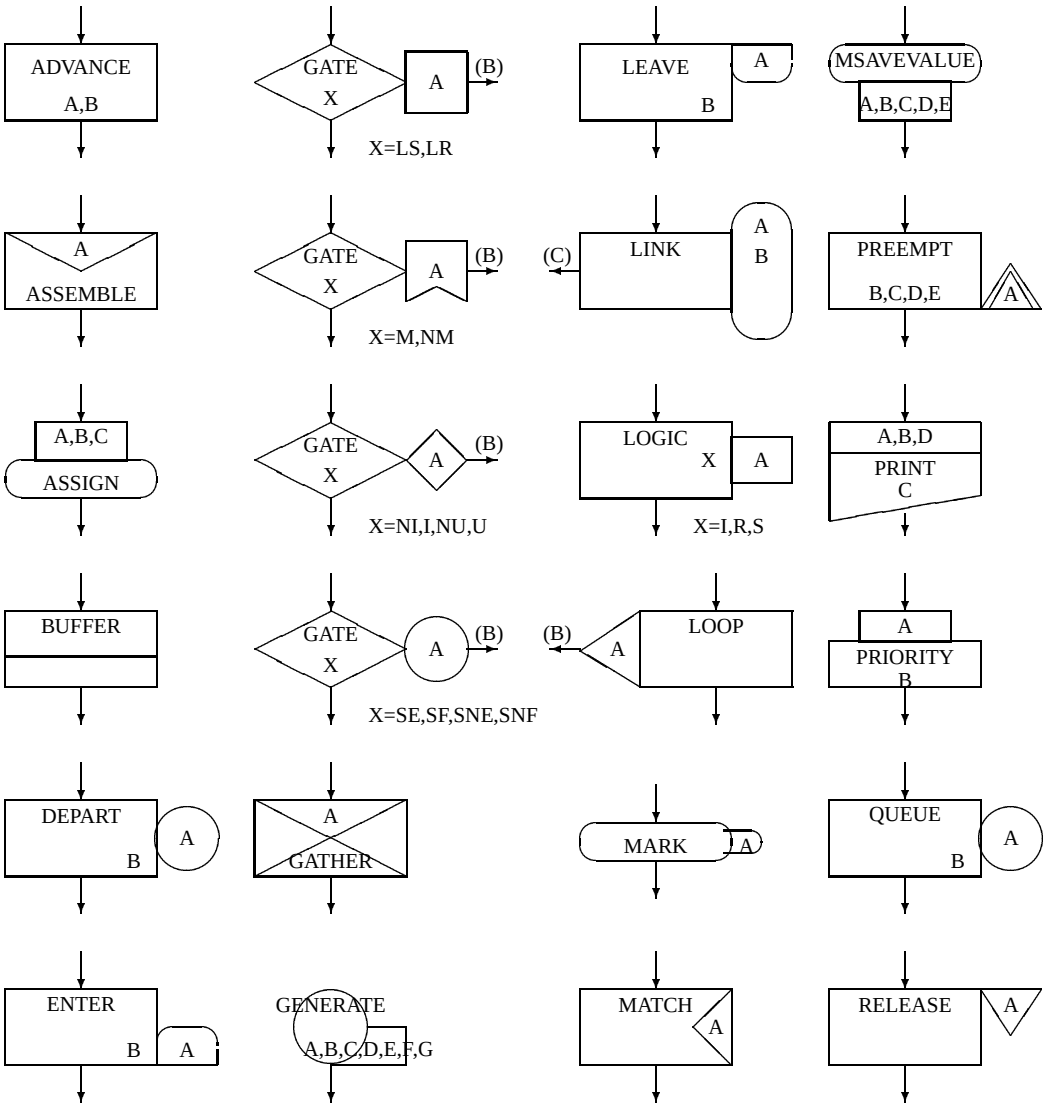

Figure 3.1: Grafische voorstelling van de GPSS-blokken

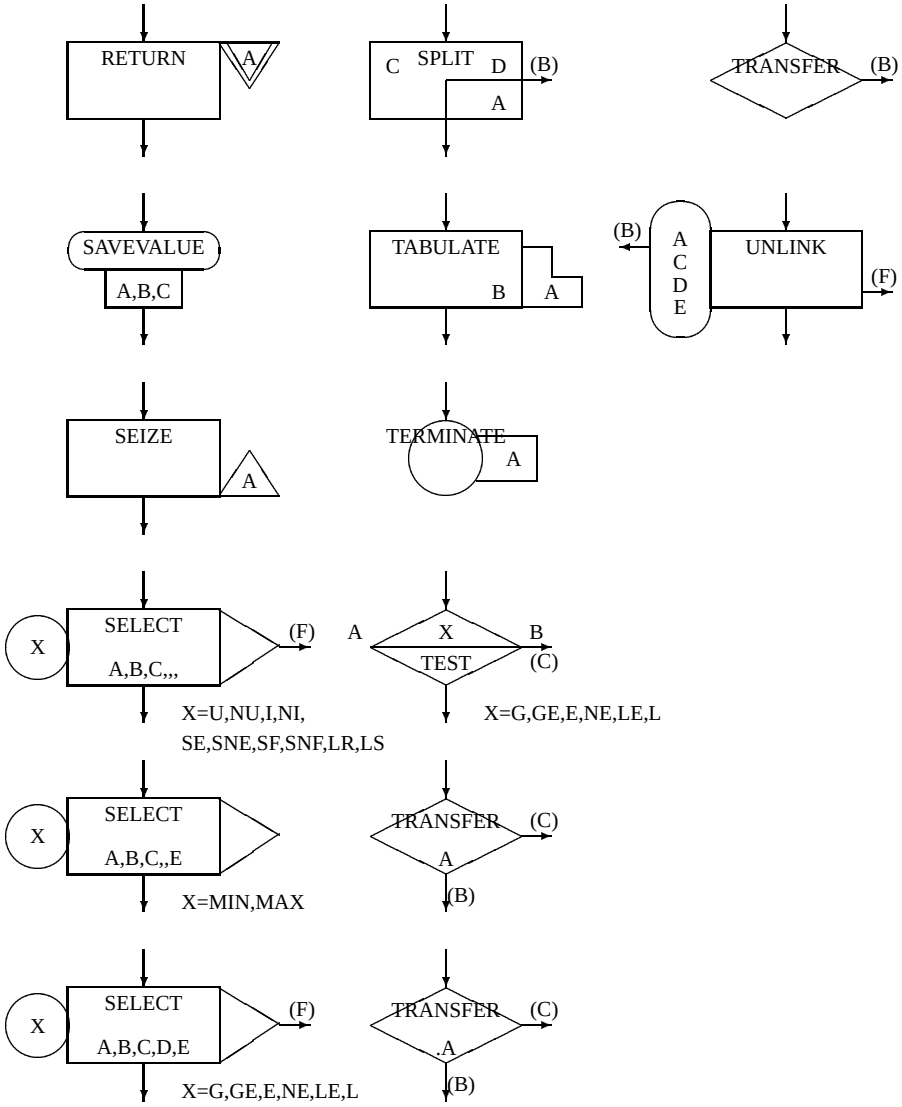

Figure 3.2: Grafische voorstelling van de GPSS-blokken (vervolg)

```
*
* Blokdeclaraties
*
       GENERATE   10,7       Binnentreden winkel
       ADVANCE    12,5       Winkelen
       QUEUE      WACHT      Aansluiten bij wachtlijn
       SEIZE      KASSA      Verkrijgen aandacht kassier
       DEPART     WACHT      Verlaten van wachtlijn
       ADVANCE    3,1        Afrekenen
       RELEASE    KASSA      Opgeven aandacht kassier
       TERMINATE  1          Verlaten winkel
*
* Commando's
*
       START      500        Simulatie van 500 klanten
       END                   Beeindigen simulatie
```

Figure 3.3: GPSS-programma voor simulatie warenhuis-systeem

- END: Beëindigt een simulatiesessie.
- INITIAL: Voorziet logic switches, savevalues of matrix savevalues van een intiële waarde bij de aanvang van een simulatie-taak.
- RESET: Herinitialiseert het model in een beperkte mate. Het commando wordt gebruikt om de reeds vergaarde statistieken uit te vegen terwijl het model toch in de huidige toestand blijft.
- START: Initialiseert de beëindigingsteller met een opgegeven waarde en start de simulatie.

3.2.10 Voorbeeld

In Figuur 3.3 is een eenvoudig GPSS-programma als voorbeeld opgenomen. Het doel van het programma is het simuleren van een warenhuis-systeem zoals geschetst in Hoofdstuk 2. In het voorbeeld wordt het model opgebouwd enkel door het gebruik van blokdeclaratie-statements. Alle entiteiten worden automatisch gecreëerd. De simulatie wordt gestart en beëindigd door twee commando's.

Figuur 3.4 is eveneens een programma voor het simuleren van een warenhuis-systeem. In plaats van één kassa beschikt het warenhuis nu over drie kassa's. De drie kassa's worden gemodelleerd door een storage. Deze entiteit moet worden gedeclareerd door een entiteitsdeclaratie-statement.

3.2.11 Evaluatie

GPSS heeft naast een aantal nadelen ook een aantal onmiskenbare voordelen. De nadelen situeren zich vooral op het niveau van de implementatie en de syntax terwijl de voordelen zich vooral binnen de gebruikte concepten manifesteren. Binnen HGPSS en de implementatie ervan door de HGPSS++-kernel is zoveel mogelijk gepoogd de nadelen van GPSS weg te werken en de voordelen te laten primeren.

De belangrijkste voordelen van GPSS zijn:

- De aanwending van de process-interaction approach.
- De algemene verspreiding van de taal en de grote hoeveelheid reeds in de taal ontwikkelde software.

```
*
* Entiteitsdeclaraties
*
KASSA  STORAGE    3          Declaratie van 3 kassa's
*
* Blokdeclaraties
*
       GENERATE   2,1        Binnentreden winkel
       ADVANCE    12,5       Winkelen
       QUEUE      WACHT      Aansluiten bij wachtlijn
       ENTER      KASSA      Verkrijgen van een kassier
       DEPART     WACHT      Verlaten van wachtlijn
       ADVANCE    3,2        Afrekenen
       LEAVE      KASSA      Opgeven van de kassier
       TERMINATE  1          Verlaten winkel
*
* Commando's
*
       START      500        Simulatie van 500 klanten
       END                   Beeindigen simulatie
```

Figure 3.4: Uitgebreid GPSS-programma voor simulatie warenhuis-systeem

- Het high-level karakter van de taal. Krachtige constructies laten toe om een systeem op een eenvoudige en snelle manier te modelleren. GPSS is een volledige programmeertaal [Boehm & Jacopini 1966]: zowel sequentie, iteratie als selectie zijn aanwezig.
- Het programmeergemak: de processor handelt vele taken af die anders expliciet zouden moeten geprogrammeerd worden.
- De automatische data-collectie en generatie van uitvoer.

De nadelen kunnen als volgt worden gecatalogeerd:

- Er is geen conceptuele ondersteuning van hiërarchisch modelleren en geen duidelijke tekstuele scheiding tussen de specificatie van processen.
- Er is geen mogelijkheid tot koppeling met externe programmatuur.
- De taal beschikt niet over een vrije codeervorm en is gericht op het gebruik van ponskaarten.
- De taal bevat een aantal inconsistenties die het memoreren van bepaalde constructies bemoeilijken. Ook het veelvuldig gebruik van mnemonics leidt tot eenzelfde moeilijkheid.
- De processor maakt gebruik van een klok die enkel gehele waarden kan aannemen. Dit maakt het voor de modelbouwer in vele gevallen noodzakelijk om de te simuleren tijd te converteren naar simulatie-tijd via een mapping.
- Alle geheugen wordt statisch gealloceerd.
- Er is binnen een programma geen verplichte scheiding tussen *environmental* en *model frame* [Zeigler 1976].
- De gebruiker heeft geen greep op de wijze waarop uitvoer wordt gegenereerd en de manier waarop de processor werkt.

Chapter 4

Object-oriëntatie

4.1 Inleiding

In Hoofdstuk 1 werd aangegeven dat één van de doelstellingen van het HGPSS-project, het construeren van een object-georiënteerde simulatie-omgeving is. Hiermee werd bedoeld dat niet enkel de modellen van de te simuleren systemen op een object-georiënteerde wijze moeten kunnen gespecifieerd worden, maar dat ook de ondersteunende kernel met behulp van een object-georiënteerde programmeertaal moet worden geïmplementeerd. In dit hoofdstuk wordt nader ingegaan op de object-georiënteerde filosofie en de redenen waarom deze in de context van discrete event simulatie interessant is.

4.2 De object-georiënteerde filosofie

De object-georiënteerde filosofie laat toe om software op een natuurlijke wijze tot stand te laten komen aangezien ze nauw aanleunt bij de visie die wijzelf hebben op de wereld rondom ons. Binnen de filosofie wordt de wereld beschouwd als een verzameling *objecten* die met elkaar interageren. Elk object bezit een *toestand* en een bepaalde *functionaliteit.* De communicatie tussen de objecten gebeurt via het zenden van *berichten.* Niet alle interne details van een object zijn naar buiten toe zichtbaar. Het zijn enkel de voor de buitenwereld relevante eigenschappen van een object die op het voorplan treden, de rest wordt *verscholen.* De objecten verhouden zich tegenover elkaar als slaven en meesters. De meester-objecten zullen door het zenden van berichten naar slaaf-objecten deze objecten opdragen bepaalde acties te ondernemen.

Sterk verbonden met het begrip object is het begrip *klasse.* De relatie tussen object en klasse is analoog aan deze tussen variabele en type in een klassieke programmeertaal. Objecten zijn *instanties* van klassen. Het zijn de klassen die binnen een programma beschreven worden, de objecten worden pas gecreëerd als het programma wordt uitgevoerd. Voor elke klasse worden de variabelen gespecifieerd die gebruikt moeten worden om de toestand van de instanties van de klasse weer te geven. Deze variabelen worden de *data* of de *lidvariabelen* van een object genoemd. Naast de data worden bij een klasse-specificatie ook de routines beschreven die het object een bepaalde functionaliteit zullen verlenen. Deze routines worden door *methodes* of *lidfuncties* aangeduid. De methodes kunnen in twee categorieën opgedeeld worden.

- De *accessormethodes* zijn deze die toegang verlenen tot de data van een object. Deze toegang kan zich beperken tot het lezen, maar kan ook het schrijven tot gevolg hebben.
- De *transformatiemethodes* werken in op de toestand van het object.

Een belangrijk aspect van klassen is de mogelijkheid tot *overerving.* Een klasse kan immers *afgeleid* worden van een andere klasse, de *basisklasse,* waarbij de eigenschappen van de basisklasse in principe worden overgenomen en aangevuld met additionele eigenschappen. Als niet van één maar van meerdere klasse wordt geërfd, wordt dit door *meervoudige overerving* aangeduid.

Een ander aspect van object-oriëntatie is *polymorfisme* en *dynamische binding.* Polymorfisme is het mechanisme waarbij eenzelfde methode op objecten behorende tot verschillende klassen van toepassing is. De bewerking kan naargelang de klasse waartoe het object behoort een verschillende uitwerking hebben. Bij de implementatie van polymorfisme is dynamische binding nodig aangezien niet a priori kan uitgemaakt worden welke de klasse zal zijn van de objecten die aan de bewerking worden onderworpen.

In [Meyer 1988] wordt gesteld dat een systeem vooralleer als volledig object-georiënteerd bestempeld te kunnen worden, minstens aan een aantal voorwaarden moet voldoen. Als het systeem aan een selectie van de voorwaarden voldoet, vertoont het slechts een bepaalde mate van object-oriëntatie. De voorwaarden zijn de volgende:

1. Het systeem moet een *object-gebaseerde modulaire structuur* bezitten. Hiermee wordt bedoeld dat de beschreven systemen gemodulariseerd moeten worden volgens hun gegevensstructuren. Bij elkaar horende gegevens worden gegroepeerd in een object. De communicatie tussen de objecten modelleert de stroom van gegevens door het systeem. Tussen de objecten onderling heerst er een zwakke koppeling, de gegevens binnen een object zijn echter sterk gekoppeld.

2. Objecten moeten beschreven worden als de implementaties van *abstracte datatypes.* In deze context zijn ook de begrippen *information hiding* en *data encapsulation* van belang.

3. Er dient een automatisch geheugenbeheer te zijn waarbij ongebruikte objecten uit het geheugen worden verwijderd zonder dat de gebruiker daar expliciet om vraagt.

4. Elk niet-eenvoudig type is een module en elke module van hoog niveau is een type, *klasse* genoemd.

5. Een klasse moet kunnen worden gedefinieerd als een uitbreiding of een restrictie van een andere bestaande klasse. Dit is het aspect *overerving.*

6. Programma-instructies moeten de mogelijkheid hebben om naar objecten van meer dan één klasse te refereren. De bewerkingen mogen daarbij verschillende realisaties in verschillende klassen hebben. Er moet met andere woorden een mogelijkheid tot *polymorfisme* aanwezig zijn, geïmplementeerd door *dynamische binding.*

7. *Meervoudige overerving* moet toegelaten zijn. Hierbij erft een klasse van meer dan één klasse en mogelijks meermaals van dezelfde klasse.

Het gebruik van object-oriëntatie bij de constructie van software biedt een aantal onmiskenbare voordelen.

- Bij object-georiënteerd programmeren vervagen de grenzen tussen ontwerp en implementatie. Bij klassieke programmeertalen is de sterke scheiding tussen de verschillende ontwerpfasen juist één van de grote problemen. In een object-georiënteerde ontwikkelingscyclus kunnen de resultaten van de éne fase rechtstreeks in de andere fasen gebruikt worden.

- De herbruikbaarheid van ontwikkelde software is heel wat groter dan bij klassieke programmeertalen. Een nieuw systeem hoeft veelal niet meer tot in de kleinste details vanaf niets opgebouwd worden. Er kan met behulp van een bottom-up strategie gebruik gemaakt worden van reeds bestaande modules die op een geschikte manier met elkaar in relatie worden gebracht.

- Het ingevoerde hogere abstractie-niveau laat een eenvoudiger management toe van grote projecten.
- Het aanpassen van een systeem aan veranderde noden wordt sterk vereenvoudigd.

4.3 Object-georiënteerd modelleren

4.3.1 Algemeen

Een systeem binnen de discrete event-wereld bestaat per definitie uit een collectie interagerende entiteiten. Het is dus meteen duidelijk dat bij de constructie van een model voor een dergelijk systeem efficiënt zal kunnen gebruik gemaakt worden van een object-georiënteerde aanpak. De verschillende soorten entiteiten binnen het systeem kunnen beschreven worden door klassen terwijl de entiteiten zelf door de instanties van de klassen, de objecten, zullen worden voorgesteld.

4.3.2 Hiërarchisch decomponeren

Het modelleren binnen GPSS is verschillend van het beschrijven van een aantal interagerende entititeiten. De in een model te gebruiken entiteitensklassen zijn immers in een basisvorm reeds voorhanden. Door een speciale soort entiteiten, blokken, wordt de manier beschreven waarin de instanties van de entiteitsklassen interageren. Het eenvoudig combineren van entiteiten en blokken heeft weinig te maken met object-oriëntatie. Een GPSS-model kan echter vanop een hoger abstractieniveau bekeken worden. Hierbij wordt niet de samenhang geïnspecteerd van de afzonderlijke entiteiten en blokken, maar van verzamelingen entiteiten en blokken die logisch bij elkaar horen. GPSS laat echter niet toe om een model vanop een dergelijk hoog niveau te beschrijven. Gezien de complexiteit die GPSS-modellen kunnen aannemen is het echter wel wenselijk om op een hoger abstractieniveau te kunnen redeneren. Deze nood is vergelijkbaar met de nood die vanuit klassieke general-purpose programmeertalen naar voor komt als de te ontwikkelen systemen een te grote omvang beginnen aan te nemen. Een object-georiënteerde techniek zal dan ook aangewezen zijn om een oplossing te bieden aan te complexe GPSS-modellen. Deze techniek vertaalt zich in het opdelen van een model in submodellen. Elk submodel is opgebouwd uit een combinatie van entiteiten en blokken die lokaal zijn ten opzichte van het submodel. Een submodel fungeert als object terwijl de entiteiten en blokken die deel uitmaken van een submodel als de data en methodes van het object kunnen beschouwd worden. Tussen de submodellen moet een vorm van communicatie mogelijk zijn. Aangezien de transacties de entiteiten zijn die informatie transporteren doorheen het systeem, moet er om communicatie tussen de submodellen mogelijk te maken een manier zijn om transacties vanuit een submodel over te brengen naar een ander model. Als de opdeling van een model in submodellen slechts tot één niveau beperkt blijft, wordt hiermee slechts een kleine contributie geleverd tot het beter beheersen van de complexiteit van een model. Het is wenselijk om de submodellen op een analoge wijze te kunnen opdelen als het hoofdmodel. Als deze procedure onbeperkt kan worden voortgezet totdat de complexiteit van de basiscomponenten aanvaardbaar is, kan van *hiërarchisch decomponeren* worden gesproken. Hierbij vervaagt het onderscheid tussen model en submodel aangezien een submodel op haar beurt uit een aantal submodellen kan bestaan die het omvattende submodel als model zullen ervaren. De analogie tussen object-oriëntatie in het algemeen en toegespitst op discrete event simulatie binnen GPSS, is in Tabel 4.1 samengevat.

De voordelen van hiërachisch decomponeneren bij het modelleren van een systeem uit de reële wereld kunnen als volgt worden samengevat [Pooley 1991]:

- Hiërarchisch decomponeren laat een natuurlijke wijze van werken toe. Elke component binnen de hiërarchie kan immers zodanig worden gekozen dat deze overeenstemt met een fysisch te onder-

ALGEMEEN	SIMULATIE
object	model
klasse	modelklasse
data	entiteiten
methodes	blokken

Table 4.1: Analogie tussen object-oriëntatie en hiërarchisch decomponeren

scheiden deel van het systeem. In dit opzicht leidt het decomponeren tot een betere modellering van de reële wereld aangezien er naast een *functionele* nu ook een *structurele* analogie kan worden verwezenlijkt.

- Het decomponeren voert een hogere graad van *abstractie* in waardoor het eenvoudiger wordt om een probleem te vatten en te beredeneren.
- De afzonderlijke componenten laten toe om informatie te *verstoppen* ten aanzien van hogere abstractie-niveaus. De informatie wordt opgenomen op de plaats waar zij relevant is en vertroebelt het globale beeld niet meer.
- De componenten uit de hiërarchie zijn *herbruikbaar*. Als een voldoend aantal basiscomponenten ter beschikking staat, kan gebruik gemaakt worden van een bottom-up techniek bij de synthese van een model.
- Het in meer detail beschrijven van een model naargelang er meer gegevens voorhanden zijn of *stepwise refinement*, wordt mogelijk.
- Het *onderhoud* van het model wordt eenvoudiger. Aanpassingen en uitbreidingen kunnen op de plaats waar ze vereist zijn worden doorgevoerd zonder de rest van het model te beïnvloeden. Fouten kunnen beter gelokaliseerd en verholpen worden.

Deze voordelen van hiërarchisch decomponeren zijn niet enkel van toepassing bij de constructie van een model met het oog op simulatie maar ook bij hiërarchisch ontwerpen in het algemeen. Bij de constructie van een model voor simulatie als in GPSS manifesteren zich enkele supplementaire voordelen van hiërarchisch modelleren.

- Vooralleer een model kan gesimuleerd worden moet er eerst een vertaling gebeuren van de high-level modelbeschrijving naar een voor de machine waarop de simulatie moet uitgevoerd worden, interpreteerbaar low-level equivalent. Deze compilatie, die traditioneel vrij tijdsintensief is, kan worden ingekort door gebruik te maken van voorgecompileerde modules. Deze modules zullen vertaalde beschrijvingen van componenten van het te simuleren model zijn. Door bij de synthese van een model enkel gebruik te maken van in een bibliotheek opgeslagen voorgecompileerde componenten kan heel wat tijd bespaard worden.
- Het gebruik maken van modules uit een bibliotheek laat ook toe om aan met het simulatie-systeem minder vertrouwde gebruikers, kant en klare bibliotheekmodules te leveren. Van deze modules wordt enkel de wijze waarop ze kunnen gebruikt worden bekend gemaakt, en niet de interne implementatie-details. Een dergelijke werkwijze laat ook toe om veiligheid in te bouwen in het systeem.

De wijze waarop GPSS uitgebreid werd om het hiërarchisch decomponeren mogelijk te maken, wordt in Hoofdstuk 5 besproken.

4.4 Object-georiënteerde implementatie van de kernel

4.4.1 Inleiding

Bij het modelleren in HGPSS wordt gebruik gemaakt van entiteiten en blokken. De wijze waarop de entiteiten en blokken interageren wordt georchestreerd door de HGPSS-processor. Op het niveau van de implementatie kunnen entiteiten, blokken en processor als objecten beschouwd worden[1]. Bij het ontwikkelen van een kernel waarin de implementaties van entiteiten, blokken en processor vervat zitten kan dus ook gebruik gemaakt worden van de object-georiënteerde filosofie. Als implementatietaal werd C++ gekozen. De HGPSS++-kernel wordt in Hoofdstuk 6 besproken.

4.4.2 C++

Historiek

C++ is een uitgebreide versie van C. C is zowel flexibel als krachtig en werd gebruikt bij de ontwikkeling van de meeste belangrijke software-produkten van de laatste 15 jaar. Wanneer een project echter een bepaalde omvang overschrijdt, wordt de limiet van C bereikt. Afhankelijk van de aard van het project, ligt deze limiet bij 25 000 tot 100 000 lijnen. Dergelijke projecten zijn moeilijk te managen omdat het vrijwel uitgesloten is om het hele programma in zijn totaliteit te vatten. Bjarne Stroustrup ervoer dit probleem toen hij in 1980 werkte bij Bell Laboratories te Murray Hill, New Jersey. Door het bijvoegen van enkele extensies bij C trachtte Stroustrup het probleem op te lossen, en met succes. Aanvankelijk werd de uitgebreide C, *C with classes* gedoopt. In 1983 werd de naam veranderd in C++.

De meeste door Stroustrup uitgedachte uitbreidingen supporteren het object-georiënteerd programmeren. Stroustrup stelt dat enkele van de object-georiënteerde eigenschappen van C++ geïnspireerd werden door een andere object-georiënteerde taal, namelijk *Simula67*. Door de combinatie van de kracht van C en de object-georiënteerde filosofie kwam een zeer interessante programmeertaal tot stand.

De syntax en mogelijkheden van C++ zullen niet besproken worden. Hiervoor wordt verwezen naar [Stroustrup 1987] of [Schildt 1990].

Voordelen

C++ bezit voor de ontwikkeling van een bibliotheek van, in casu, discrete-event simulatie-routines, een aantal onmiskenbare voordelen.

- Ten eerste is C++ in hoge mate portabel. De taal is immers sterk gestandardiseerd en beschikbaar op kwasi alle belangrijke computer-systemen gaande van personal computers tot mainframes.
- Alhoewel C++ gecompileerd wordt, heeft de taal toch de flexibiliteit van een geïnterpreteerde taal. In het bijzonder moet de fexibiliteit nauwelijks onderdoen voor deze bereikt in dedicated simulation language interpreters.
- Uiteindelijk wordt C++ ondersteund door een hele pleiade van development tools, debuggers en class libraries, wat ook als een belangrijk voordeel kan worden beschouwd.

[1] Op het niveau van het modelleren zijn dit de submodellen.

4.4.3 Bestaande C++-gebaseerde discrete-event simulatie-tools

Inleiding

Het ontwikkelen van een C++-bibliotheek voor discrete-event simulatie is geen nieuw gegeven. De commerciële pakketten *SIM++* en *Meijin++* bijvoorbeeld, zijn recent ontwikkelde C++-bibliotheken met routines die kunnen gebruikt worden bij de constructie van een proces-georiënteerde discrete event-simulator. Zij vertonen bijgevolg gelijkenissen met de ontwikkelde HGPSS++-kernel vermits hetzelfde doel wordt nagestreeft. Er is echter wel een verschil aangaande het abstractieniveau van de beschikbare primitieven. HGPSS++ is een specifiek op HGPSS gerichte bibliotheek en situeert zich tengevolge daarvan op een hoog niveau terwijl de andere pakketten algemener zijn en zich op een relatief laag niveau bevinden.

SIM++

SIM++ [Sim++ 1989] is een in Calgary ontwikkelde proces-georiënteerde discrete-event simulatie-taal die in C++ is ingebed. De taal kan als een midden-niveau simulatie-taal beschouwd worden aangezien ze in feite uit niets anders bestaat dan een bibliotheek van basisroutines die kunnen gebruikt worden bij de constructie van een simulator. Bij de eigenlijke simulatie wordt gebruik gemaakt van een ingebouwde scheduler. Het grote voordeel van het gebruik van SIM++ is dat eenzelfde simulatie-programma kan gebruik maken van verschillende run-time executives, naargelang de beschikbare hardware. Concreet staan drie run-time executives ter beschikking.

- De *optimized sequential* executive laat toe om een simulatie-programma uit te voeren op een sequentiële computer.
- De *distributed* executive distribueert de uitvoering van een programma over verschillende processoren. Hierbij wordt gebruik gemaakt van het TimeWarp-principe [Fujimoto 1990].
- De gedistribueerde uitvoering van een programma kan op een sequentiële computer worden nagebootst door gebruik te maken van de *emulated distributed* executive.

Modellen worden binnen SIM++ opgebouwd door gebruik te maken van entiteiten. Deze entiteiten beschikken over een eigen workspace. Communicatie tussen entiteiten gebeurt door het schedulen van events. De concreet te gebruiken events en entiteiten moeten door de modelbouwer zelf van een basisklasse afgeleid worden zodat niet van een high-level simulatie-taal kan gesproken worden. De door SIM++ geleverde hulpmiddelen zijn:

- Een aantal datatypes en klassen. Van deze klassen moeten onder andere de in de simulatie te gebruiken entiteiten en events worden afgeleid.
- Routines voor de dynamische creatie en destructie van entiteiten en events.
- Primitieven voor de controle van het scheduling mechanisme.
- Primitieven voor de selectie van de uit te voeren events.
- Hulpmiddelen voor invoer en uitvoer van gegevens.
- Hulpmiddelen voor de generatie van random numbers en manipulatie van distributies.
- Gereedschappen voor het verzamelen van geproduceerde gegevens in de vorm van data-collectoren.

- Faciliteiten voor het manipuleren van door de gebruiker beheerde lijsten.
- Hulpmiddelen voor het afhandelen van geconstateerde fouten.

Meijin++

De Meijin++ *C++ Modeling and Simulation Class Library* is een low-cost bibliotheek van kant en klare wetenschappelijke routines. Deze routines situeren zich op het domein van de discrete-event simulatie en de numerieke analyse. Zoals SIM++ levert Meijin++ hulpmiddelen voor het modelleren van entiteiten, het vergaren van informatie, de behandeling van lijsten en het opvangen van fouten. Een model wordt opgebouwd door de combinatie en interactie van een aantal concurrente taken. Hierdoor wordt modulariteit ingevoerd. De primitieven voor discrete-event simulatie aangeboden door Meijin++ situeren zich op een hoger niveau dan deze die SIM++ ter beschikking stelt.

Chapter 5

HGPSS

5.1 Inleiding

Aan drie van de vier in Hoofdstuk 1 geschetste problemen die kunnen optreden bij het gebruik van simulatie als probleemoplossende methode, werd getracht een oplossing te bieden door de ontwikkeling van *HGPSS*. HGPSS staat voor *Hierarchical General-Purpose Simulation System* en is een discrete-event process-interaction simulatie-taal voor het ontwikkelen en simuleren van hiërarchische simulatie-modellen. De taal is een onvolmaakte superset van GPSS/360[1] zoals deze beschreven wordt in [Schriber 1974].

De problemen waaraan HGPSS een oplossing tracht te bieden zijn:

- De al dan niet onoverkomelijke moeilijkheden om een discrete-event simulatie-systeem te koppelen aan een continue simulatie-partner.
- De stroeve en onnatuurlijke wijze waarop een simulatie-systeem met externe software kan uitgebreid worden voor preprocessing, postprocessing of modelconstructie. Sommige pakketten bieden zelfs totaal geen dergelijke faciliteiten.
- Het ontbreken van voorzieningen voor de constructie van hiërarchische modellen.

De oplossingen die HGPSS biedt aan deze problemen vormen tevens de belangrijkste punten van onderscheid met GPSS. De belangrijke verschilpunten kunnen als volgt worden samengevat:

- Het belangrijkste en morfologisch meest opvallend verschil tussen GPSS en HGPSS zijn de HGPSS-constructies ter ondersteuning van hiërarchisch modelleren. Hierarchie wordt mogelijk gemaakt door de invoering van
  - vier zogenaamde *sectie markering-statements*,
  - vier nieuwe blokdeclaratie-statements,
  - één nieuw entiteitsdeclaratie-statement en
  - twee nieuwe commando-statements.
- Door het inbedden van C++-statements in een HGPSS-programma wordt het incorporeren van bijkomende software eenvoudig. Binnenin HGPSS-statements kan naar C++-variabelen en -functies worden gerefereerd.

[1] In het verdere verloop van dit hoofdstuk wordt met GPSS steeds GPSS/360 bedoeld.

- Communicatie tussen het model en een continue simulatie wordt mogelijk door de invoering van twee additionele blokdeclaratie-statements.

Naast belangrijke zijn er ook nog een aantal verschillen van ondergeschikt belang.

- Ter vervanging van de stroeve GPSS codeervorm werd een vrijere vorm ingevoerd.
- Enkele van de GPSS-commando's kregen binnen HGPSS een andere of uitgebreide betekenis. Voor andere commando's werd de parameter-lijst aangepast.
- Enkele achterhaalde kenmerken van GPSS werden binnen HGPSS gesupprimeerd.
- Een aantal andere achterhaalde kenmerken maken nog deel uit van de syntax van HGPSS om compatibiliteit te verzekeren, maar hebben geen effect bij de uiteindelijke uitvoering van de simulatie.

Ondanks de verschillen, wijzigingen en uitbreidingen blijft HGPSS sterk bij GPSS aanleunen en is een GPSS-programma door het uitvoeren van een beperkt aantal ingrepen snel om te vormen tot een HGPSS-programma.

In hetgeen volgt wordt een overzicht gegeven van de belangrijkste kenmerken van HGPSS. In Appendix A is evenwel een uitgebreidere Engelstalige handleiding van HGPSS te vinden die zich richt op het gebruik van de taal in de praktijk. In deze handleiding bevindt zich een *extended Backus Naur form*-beschrijving waarin de volledige syntax van de taal gedetailleerd is weergegeven. In de handleiding is ook een overzicht van de taal opgenomen in de vorm van tabellen. Deze tabellen geven op een bondige en informele wijze de syntax en de semantiek weer van de HGPSS-statements en zijn handig als leidraad bij het construeren van een HGPSS-programma.

5.2 Vorm van een programma

5.2.1 Model- en commandosecties

Een HGPSS-programma bestaat uit een opeenvolging van verschillende secties. Een sectie is een verzameling van bij elkaar horende statements in een bepaalde volgorde. Elke sectie wordt begrensd door twee sectie markering-statements. Er bestaan twee soorten secties:

- *modelsecties* en
- *commandosecties.*

In een programma bevinden zich eerst een aantal modelsecties gevolgd door één commandosectie. De commandosectie moet zich verplicht ná de modelsecties bevinden. Zowel modelsecties als commandosectie kunnen eventueel weggelaten worden.

Een modelsectie wordt gebruikt om één van de submodellen waaruit het totale model van het te modelleren systeem bestaat, te specifiëren. Binnen een modelsectie kunnen zich enkel entiteitsdeclaratie- en blokdeclaratie-statements bevinden. Alle door deze statements gedeclareerde entiteiten en blokken worden als lokaal binnen het submodel beschouwd. Ook alle gebruikte namen zijn lokaal. Een modelsectie wordt afgebakend door twee sectie markering-statements.

- Het MODEL-statement duidt het begin van een modelsectie aan.
- Het ENDMODEL-statement sluit een modelsectie af.

Een MODEL-statement bestaat uit het sleutelwoord `MODEL` gevolg door een parameter. Deze parameter is de naam van de modelklasse gespecifieerd door de sectie. Een modelklasse kan worden vergeleken met een klasse in een object-georiënteerde programmeertaal als C++. De modelsecties zijn de definities van deze modelklassen. Van een modelklasse kunnen één of meerdere instanties worden gecreëerd en gebruikt als object. De in het MODEL-statement op te geven naam is dus de naam van de modelklasse en niet van een instantie van de klasse. Het ENDMODEL-statement bestaat enkel uit het sleutelwoord `ENDMODEL`.

Een commandosectie bestaat enkel en alleen uit commando-statements en is zoals een modelsectie begrensd door twee sectie markering-statements.

- Het COMMAND-statement duidt het begin van een commandosectie aan.
- Het ENDCOMMAND-statement sluit de commandosectie af.

Beide statements bestaan enkel uit een sleutelwoord, respectievelijk `COMMAND` en `ENDCOMMAND`.

Het gebruik van model- en commandosectie wordt in Figuur 5.1[2] geïllustreerd aan de hand van het warenhuis-systeem uit vorige hoofdstukken. Het programma bestaat uit twee secties: een model- en een commandosectie. Lijn (1)-(13) vormt de modelsectie terwijl lijn (14)-(22) de commandosectie voorstelt. Naast de indeling in secties is het enige belangrijke verschil tussen dit programma en het GPSS-programma in Figuur 3.3 het gebruik van het SIMULATE-statement in lijn (18). De betekenis van dit statement wordt verder verklaard. Een tweede minder belangrijk verschil met het programma in Figuur 3.3 is de vrijere codeervorm. Labels, sleutelwoorden en parameters hoeven in HGPSS niet in een welbepaalde positie te beginnen. Er moet wel steeds voor gezorgd worden dat binnen een statement label, sleutelwoord, parameters en commentaar van elkaar gescheiden blijven. Zoals in GPSS beslaat een statement in HGPSS exact één programmalijn. Voor het scheiden van parameters mogen enkel comma's gebruikt worden. De vrijere codeervorm laat toe om door middel van intanding duidelijk het verschil aan te geven tussen sectie markering-statements en overige statements. Op deze manier wordt de indeling in secties in één oogopslag duidelijk.

Door het scheiden van blokdeclaratie- en entiteitsdeclaratie-statements enerzijds en commando-statements anderzijds, wordt een simulatie-programma opgedeeld in een *model frame* en een *environmental frame* zoals beschreven in [Zeigler 1976]. In het model frame dat in HGPSS bestaat uit de collectie modelsecties, wordt het volledige model van het te simuleren systeem beschreven. Een systeem kan echter onderhevig zijn aan een veranderende omgeving. In het environmental frame of de commandosectie in HGPSS, worden de eventueel veranderende omgevingsomstandigheden geschetst waaraan het model wordt onderworpen. Een analoge opdeling in model en evironmental frame wordt bijvoorbeeld ook in de discrete-event simulatie-taal *HSL* [Sanderson 1991] gemaakt.

5.2.2 Identifiers

In GPSS moet een identifier gebruikt als label verplicht uit niets dan hoofdletters en cijfers bestaan. Een identifier mag bovendien uit niet meer dan vijf karakters opgebouwd zijn. Het afleiden van betekenisvolle identifiers is, gebruik makend van deze regels, vrijwel onmogelijk. Dit euvel werd in HGPSS uit de weg geruimd door het gebruik van identifiers van maximaal vijftig karakters toe te laten, opgebouwd uit

- hoofdletters,
- kleine letters[3],

[2] De in deze en volgende figuren gebruikte lijnnummers maken geen deel uit van de HGPSS-taal. Ze werden enkel in de figuren opgenomen om op een eenvoudige wijze naar bepaalde lijnen te kunnen refereren.

[3] HGPSS is case-sensitive.

```
 (1) MODEL Warenhuis
 (2) *
 (3) * Blokdeclaraties
 (4) *
 (5)           GENERATE        10,7              Binnentreden winkel
 (6)           ADVANCE         12,5              Winkelen
 (7)           QUEUE           WACHT             Aansluiten bij wachtlijn
 (8)           SEIZE           KASSA             Verkrijgen aandacht kassier
 (9)           DEPART          WACHT             Verlaten van wachtlijn
(10)           ADVANCE         3,1               Afrekenen
(11)           RELEASE         KASSA             Opgeven aandacht kassier
(12)           TERMINATE       1                 Verlaten winkel
(13) ENDMODEL
(14) COMMAND
(15) *
(16) * Commando's
(17) *
(18)           SIMULATE        Warenhuis         Installeer model
(19)           START           500               Simulatie van 500 klanten
(20)           PRINT           /                 Druk alle informatie af
(21)           END                               Beeindigen simulatie
(22) ENDCOMMAND
```

Figure 5.1: Model- en commandosectie

- cijfers,
- _ en
- $

Dergelijke identifiers laten niet enkel toe om betekenisvolle namen te kiezen maar ook om door gebruik te maken van hoofdzakelijk kleine letters een duidelijk onderscheid te creëren tussen sleutelwoorden en identifiers. Zoals in GPSS bestaan sleutelwoorden in HGPSS immers steeds uit hoofdletters. In Figuur 5.1 werd bijvoorbeeld in lijn (1) en (18) de identifier `Warenhuis` gebruikt. Deze vorm van naamgeving draagt de voorkeur boven de in GPSS gebruikte manier zoals geïllustreerd in lijn (7), (8), (9) en (11).

5.2.3 Commentaar

Commentaar kan in een HGPSS-programma op dezelfde wijzen worden opgenomen als in een GPSS-programma. In Figuur 5.1 zijn beide wijzen geïllustreerd. In eerste instantie kan commentaar op het einde van een regel opgenomen worden, ná de statement-parameters. Een volledige lijn wordt als commentaar beschouwd als het eerste betekenisvolle karakter op de lijn een * is. Om de duidelijkheid te bevorderen mogen overal in een programma waar dit nodig geacht wordt, één of meerdere lege lijnen opgenomen worden.

5.3 Interactie met C++

5.3.1 Ingebedde C++

Een HGPSS-programma wordt vertaald naar een C++-programma waarbij tussen gewone C++-statements en -constructies, HGPSS++-statements zijn opgenomen. Deze HGPSS++-statements zijn niets anders

```
 (1) ...
 (2) -  Dit is C++-code
 (3) ...
 (4) %{
 (5)    Dit is C++-code
 (6) %}
 (7) ...
 (8) MODEL Name
 (9)           ...
(10) -         Dit is C++-code
(11)           ...
(12) %{
(13)           Dit is C++-code
(14) %}
(15) ...
(16) ENDMODEL
(17) ...
```

Figure 5.2: Ingebedde C++-code

dan aanroepen van functies uit de ondersteunende HGPSS++-kernel. In een HGPSS-programma kunnen C++-statements worden opgenomen die bij de vertaling ongewijzigd zullen worden overgenomen. De C++-statements binnen een HGPSS-programma moeten door de vertaler van de HGPSS-statements kunnen worden onderscheiden. Er werden twee methodes voorzien om aan te geven dat één of meerdere programmalijnen niet als HGPSS- maar als C++-code moeten worden beschouwd.

- Door een `-` als eerste betekenisvolle karakter wordt een volledige lijn als C++-code gedeclareerd. Deze situatie is te vergelijken met de declaratie van commentaar door een `*`.
- Door een blok bestaande uit één of meerdere lijnen te plaatsen tussen `%{` en `%}`[4] wordt het volledige blok als C++-code gedeclareerd. De sequentie `%}` moet de eerste betekenisvolle karakter-sequentie op een lijn zijn terwijl `%}` de laatste betekenisvolle sequentie op een lijn moet zijn.

C++-code mag overal in een programma opgenomen worden, zowel binnen een model- of commandosectie als tussen de secties in. De situatie is analoog aan maar toch verschillend van de systemen waarbij binnen een gasttaal, statements van een andere taal zijn opgenomen en als dusdaning aangeduid. Deze statements worden door een vertaler omgezet naar statements en functie-aanroepen in de gasttaal. Zo kunnen bijvoorbeeld in het database-systeem DB2, SQL-statements opgenomen worden in een PL/I-programma [Date 1990]. De SQL-statements worden voorafgegaan door `EXEC SQL` om ze te onderscheiden van de PL/I-statements. De SQL-statements worden door een precompiler omgezet naar PL/I statements. Alhoewel C++ uiteindelijk de gasttaal is voor HGPSS, komt dit niet tot uiting in een HGPSS-programma. HGPSS is niet ingebed in C++, maar C++ in HGPSS. Het zijn met andere woorden niet de HGPSS-statements die van de C++-statements worden onderscheiden, maar de C++-statements die van de HGPSS-statements worden onderscheiden door `-` of `%{ %}`. De reden voor dit verschil ligt in het feit dat de HGPSS-statements door de vertaler niet enkel vertaald worden naar de overeenkomstige C++-statements (HGPSS++-statements), maar dat terzelfdertijd automatisch een volledig omhullend C++-programma wordt gegenereerd. Bij de bespreking van de implementatie van de vertaler in Hoofdstuk 7 zal op dit en andere aspecten van de vertaler verder ingegaan worden. In Figuur 5.2 zijn de methodes om C++-code in een HGPSS-programma op te nemen geïllustreerd.

[4] De keuze voor deze karakter-sequenties werd geïnspireerd door de wijze waarop gasttaal-statements in *LEX* en *YACC* worden aangeduid [Mason & Brown 1990].

```
 (1) ...
 (2) -          #include <math.h>
 (3) ...
 (4) %{
 (5)            value_type value_1,value_2;
 (6)            number_type number_1,number_2;
 (7) %}
 (8) ...
 (9)            GENERATE        "value_1,"5.23*sqrt(value_2)"
(10) ...
(11)            ADVANCE         P"number_1",P*"number_2"
(12) ...
```

Figure 5.3: Refereren naar C++-variabelen in HGPSS-parameters

5.3.2 Refereren naar C++-variabelen en -functies

Het is vrij nutteloos om C++-code te kunnen inbedden tussen HGPSS-statements als er geen mogelijkheid tot interfacing bestaat tussen de C++- en HGPSS-statements. Daarom werd de mogelijkheid voorzien om binnenin HGPSS-statements, C++-variabelen of -functies te gebruiken. Er zijn verschillende omstandigheden onder dewelke in een HGPSS-statement naar C++-variabelen of -functies kan gerefereerd worden.

- Voor de hand liggend is het gebruik van een C++-variabele als vervanging van een constante numerieke waarde binnen een parameter (Figuur 5.3). Om in deze omstandigheid een referentie naar een C++-variabele te kunnen onderscheiden van een zuivere HGPSS-parameter, moet de C++-variabele tussen dubbele aanhalingstekens geplaatst worden. Niet enkel een C++-variabele kan gebruikt worden, ook het resultaat van een aanroep van een C++-functie of een volledige C++-expressie kan op deze wijze worden aangewend.

- Een tweede manier om C++ en HGPSS te koppelen situeert zich in het gebruik van C++-functies als entiteiten binnen HGPSS, meer bepaald als functie, variabele, booleaanse variabele of fullword variabele (Figuur 5.4[5]). Dit werd mogelijk gemaakt door de GPSS-statements die instaan voor de declaratie van een dergelijke entiteit, uit te breiden. Meer bepaald kunnen de FUNCTION-, VARIABLE-, BVARIABLE- en FVARIABLE-statements naast de manier waarop ze in GPSS gebruikt worden, ook op een andere manier worden ingezet. Door als enige parameter na het `FUNCTION`-, `VARIABLE`-, `BVARIABLE`- of `FVARIABLE`-sleutelwoord een verwijzing naar een C++-functie op te nemen, wordt aan het simulatie-systeem kenbaar gemaakt dat de aangeduide functie als HGPSS-entiteit te interpreteren is. De C++-functies die op een dergelijke wijze aangewend worden, moeten wel aan een aantal voorwaarden voldoen. Zo mogen de functies geen argumenten hebben en moeten ze een resultaat van een geschikt type teruggeven. De vereiste types van de resultaten van de als entiteit gebruikte C++-functies zijn in Tabel 5.1 terug te vinden. Deze types zijn door het simulatie-systeem gekend.

5.4 Interactie met externe software

Voor de koppeling van een HGPSS-simulatie met externe software werden ten opzichte van GPSS twee bijkomende blokken voorzien. In vorige sectie werd reeds besproken hoe met C++ kan geïnterageerd

[5]De reden voor het noodzakelijk gebruik van het -karakter in lijn (13), (14), (15), (16) zal in Hoofdstuk 7 verduidelijkt worden.

Entity	Result type
FUNCTION	value_type
VARIABLE	value_type
BVARIABLE	boolean_type
FVARIABLE	value_type

Table 5.1: Resultaattypes van als entiteit te gebruiken C++-functies

```
 (1) ...
 (2) %{
 (3)           value_type V_Func(void)
 (4)              {
 (5)                ...
 (6)              }
 (7)           boolean_type B_Func(void)
 (8)              {
 (9)                ...
(10)              }
(11) %}
(12) ...
(13) Func      FUNCTION        /"V_Func"
(14) Var       VARIABLE        /"V_Func"
(15) FVar      FVARIABLE       /"V_Func"
(16) BVar      BVARIABLE       /"B_Func"
(17) ...
```

Figure 5.4: Gebruik van C++-functies als HGPSS-entiteiten

worden. Deze interactie vindt echter plaats op het moment dat het HGPSS-*programma* wordt uitgevoerd en niet de *simulatie.* Een HGPSS-programma is slechts actief bij de declaratie van de modelsecties en bij het uitvoeren van de HGPSS-commando's. Bij de uitvoering van het START-commando wordt de simulatie actief. Op dit moment worden niet de in het HGPSS-programma opgenomen statements uitgevoerd maar de acties geassocieerd met de gedeclareerde blokken.

De bijkomende HGPSS-blokken voor de koppeling met externe software laten toe om externe software uit te voeren tijdens de eigenlijke simulatie. Globaal gezien komt het erop neer dat een "leeg" blok van een aantal acties wordt voorzien die in C++ werden geïmplementeerd. Als een transactie het blok op haar weg ontmoet zullen de acties worden uitgevoerd. De blokken die op een dergelijke manier kunnen worden gebruikt zijn:

- Het INTERN-blok. Bij de declaratie van dit blok wordt één parameter opgegeven. Deze parameter is de naam van een C++-functie die zal worden aangeroepen als een transactie het blok betreedt. Aangezien het om een C++-functie gaat moet de naam ook tussen dubbele aanhalingstekens geplaatst worden zoals in de andere gevallen waarbij C++ en een HGPSS-statement gecombineerd worden.

- Het EXTERN-blok. Bij de declaratie van dit blok worden twee parameters gespecifieerd. De eerste parameter is de naam van een C++-functie die zal aangeroepen worden als een transactie het blok betreedt. De tweede parameter is de naam van een C++-functie die zal aangeroepen worden op elk actief tijdspunt. De functienamen moeten eveneens tussen dubbele aanhalingtekens geplaatst worden. Een EXTERN-blok kan gebruikt worden om de discrete event simulatie te koppelen aan externe software die asynchroon is ten opzichte van de discrete event simulatie. De functie die op elk actief tijdspunt wordt aangeroepen kan immers steeds de externe software ondervragen en nagaan of er acties moeten worden ondernomen. Voor het uitvoeren van een dergelijke ondervraging

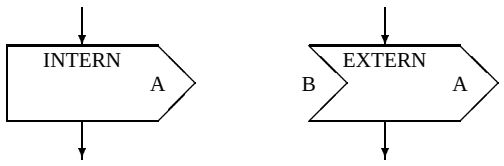

Figure 5.5: Symbool INTERN- en EXTERN-blok

is geen transactie vereist. Meer specifiek kan een EXTERN-blok gebruikt worden om de discrete event simulatie te koppelen aan een continue simulatie. Als een bericht vanuit de discrete event-simulatie naar de continue simulatie moet gestuurd worden kan dit gebeuren door een transactie het EXTERN-blok te laten betreden waardoor de functie geassocieerd met de eerste parameter van het blok, zal aangeroepen worden. Als de continue simulatie een bericht stuurt naar de discrete event simulatie kan dit bericht opgevangen worden door de functie die met de tweede parameter van het EXTERN-blok is geassocieerd. Als de continue simulatie over een analoog systeem beschikt is de koppeling verzekerd.

Om INTERN- en EXTERN-blokken ook te kunnen opnemen in de grafische voorstelling van een model, wordt het gebruik van de symbolen uit Figuur 5.5 voorgesteld.

5.5 Hiërarchisch modelleren

Door de indeling van een HGPSS-programma in modelsecties en een commandosectie is reeds de basis gelegd voor het hiërarchisch modelleren. Door de opdeling in modelsecties kunnen de verschillende componenten waaruit de hiërarchie is opgebouwd, beschreven worden. Er moet echter ook een manier zijn om de ondergeschiktheid van bepaalde submodellen aan andere aan te duiden. Dit gebeurt door de invoering van een ten opzichte van GPSS nieuw entiteitsdeclaratie-statement, namelijk het SUBMODEL-statement. Binnen de beschrijving van een model worden met behulp van dit statement de modellen aangeduid die moeten beschouwd worden als submodel ten opzichte van dat model. Elk submodel wordt beschouwd als een lokale entiteit en moet bijgevolg ook een naam krijgen. Bovendien moet voor elk submodel aangeduid worden welke de modelklasse is waartoe het submodel behoort. In Figuur 5.6 is het gebruik van het SUBMODEL-statement geïllustreerd. Door het declareren van modellen als submodel ten opzichte van andere modellen onstaat een hiërarchie-structuur in de vorm van een boom die het totale model van het te simuleren systeem voorstelt (Figuur 5.7). Bij een boomvormige hiërarchie is er steeds één model dat zich bovenaan bevindt. Dit model moet aan het systeem kenbaar worden gemaakt. Daartoe werd het SIMULATE-commando voorzien. Dit commando bestaat niet in GPSS/360 maar werd ontleend aan GPSS/H [Banks]. Daar heeft het echter een andere betekenis en syntax als in HGPSS. Door gebruik te maken van het statement wordt aangeduid welk model zich bovenaan de hiërarchie bevindt en als *hoofdmodel* moet worden beschouwd. Daartoe wordt ná het sleutelwoord `SIMULATE` de naam opgenomen van de modelklasse waartoe het hoofdmodel behoort. Bij de uitvoering van het commando zal ervoor gezorgd worden dat een instantie van de gewenste modelklasse wordt gecreëerd en geïnstalleerd voor simulatie. Het hoofdmodel wordt dus niet expliciet benoemd, dit in tegenstelling met de modellen die als submodel ten opzichte van een ander model worden gebruikt. Aangezien er terzelfdertijd slechts één model het hoofdmodel kan zijn, levert het weglaten van een expliciete benoeming geen moeilijkheden op. Om de in Figuur 5.6 geschetste hiërarchie te simuleren moet de commandosectie er als in Figuur 5.8 uitzien.

```
(1) ...
(2) MODEL Model1
(3)           ...
(4) A         SUBMODEL            Model2
(5) B         SUBMODEL            Model3
(6)           ...
(7) ENDMODEL
(8)
(9) MODEL Model2
(10)          ...
(11) ENDMODEL
(12)
(13) MODEL Model3
(14)          ...
(15) ENDMODEL
(16) ...
```

Figure 5.6: Gebruik SUBMODEL-statement

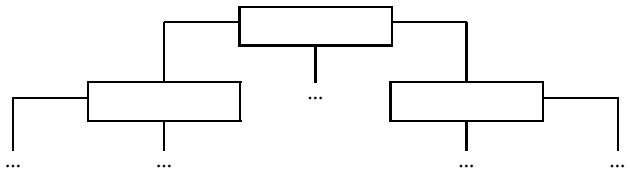

Figure 5.7: Boomvormige hiërarchie

```
(1) COMMAND
(2) ...
(3)           SIMULATE        Model1
(4) ...
(5) ENDCOMMAND
```

Figure 5.8: Gebruik SIMULATE-statement

Aangezien alle entiteiten en blokken lokaal zijn aan een model, kan er zonder verdere voorzieningen geen communicatie worden bewerkstelligd tussen de verschillende componenten van de hiërarchie. In HGPSS werden daarom vier blokken ingevoerd die met het oog op communicatie kunnen gebruikt worden. De communicatie wordt uitgevoerd door het uitwisselen van transacties. Transacties kunnen bijgevolg niet als lokale entiteiten beschouwd worden. Zij transporteren gegevens doorheen het volledige systeem en zijn daarom globaal. Een model wordt in het kader van inter-model communicatie beschouwd als een zwarte doos met een aantal in- en uitgangen waarlangs transacties het model kunnen betreden en verlaten (Figuur 5.9). De transacties kunnen enkel de paden volgen die door de hiërarchie opgelegd zijn. Een transactie kan met andere woorden vanuit een model enkel haar weg voortzetten naar een submodel of naar het bovenliggende model. Er zijn dus vier mogelijkheden (Figuur 5.10):

1. De transactie betreedt een model vanuit *het* bovenliggend model.
2. De transactie verlaat een model naar *het* bovenliggend model.
3. De transactie betreedt een model vanuit *een* submodel.
4. De transactie verlaat een model naar *een* submodel.

Voor elk van deze gevallen werd in HGPSS een specifiek blok voorzien.

1. Een INPUT-blok laat een transactie toe om vanuit het bovenliggende model, het model te betreden waarin het blok zich bevindt. Aangezien er verschillende plaatsen in een model kunnen zijn waar transacties vanuit het bovenliggend het model betreden, moeten deze plaatsen een unieke naam krijgen. Een INPUT-blok heeft dan ook als enige parameter de naam van de *input.*
2. Een OUTPUT-blok laat een transactie toe om het model waarin het blok zich bevindt te verlaten naar het bovenliggende model. Aangezien er verschillende plaatsen in een model kunnen zijn waar transacties het model verlaten naar het bovenliggend model, moeten deze plaatsen eveneens een unieke naam krijgen. Een OUTPUT-blok heeft dan ook als enige parameter de naam van de *output.*
3. Een LEAVEMODEL[6]-blok laat een transactie toe om een model te betreden vanuit een submodel. Aangezien een model verschillende submodellen kan bezitten moet de naam van het submodel in kwestie als eerste parameter worden gespecifieerd. Aangezien bovendien een transactie een model pas kan betreden vanuit een submodel als deze transactie het submodel eerst verlaat, zal als tweede parameter de naam van de output van het submodel waarlangs de transactie het submodel verlaat moeten worden opgegeven. De plaatsen waar een transactie een model betreedt vanuit een submodel worden *leaves* genoemd.
4. Een ENTERMODEL[7]-blok laat een transactie toe om een model te verlaten naar een submodel. Aangezien een model verschillende submodellen kan bezitten moet de naam van het submodel in kwestie als eerste parameter worden gespecifieerd. Aangezien bovendien een transactie een submodel na het verlaten van een model ook moet betreden, zal als tweede parameter de naam van de input van het submodel waar de transactie het submodel betreedt moeten worden opgegeven. De plaatsen waar een transactie een model verlaat naar een submodel worden *enters* genoemd.

[6] "Leave" is te interpreteren vanuit het oogpunt van het submodel.
[7] "Enter" is te interpreteren vanuit het oogpunt van het submodel.

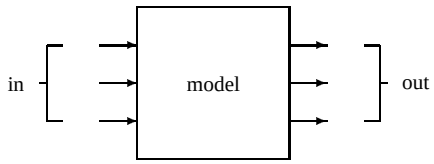

Figure 5.9: Een model als zwarte doos

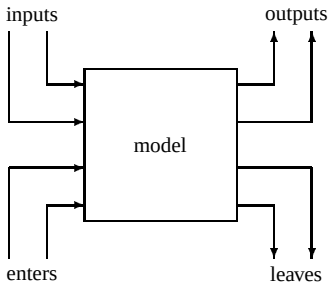

Figure 5.10: Een model als uitgebreide zwarte doos

Een ENTERMODEL-blok moet dus steeds met een INPUT-blok geassocieerd zijn, terwijl een LEAVEMODEL-blok steeds met een OUTPUT-blok moet overeenstemmen. Om INPUT-, OUTPUT-, ENTERMODEL- en LEAVEMODEL-blokken ook in een grafische beschrijving te kunnen opnemen, wordt het gebruik van de symbolen uit Figuur 5.12 voorgesteld.

Tussen een model en een submodel hoeft niet noodzakelijk een transactiestroom te bestaan. De transactiestroom kan eventueel ook enkel in één richting lopen. De mogelijkheden worden in Figuur 5.11 abstraherend weergegeven. De transactiestroom kan uiteraard ook in één of beide richtingen meervoudig zijn. In dergelijke gevallen zijn er meerdere paden in een bepaalde richting tussen beide modellen. Tenslotte wordt in Figuur 5.13 het gebruik van INPUT-, OUTPUT-, ENTERMODEL- en LEAVEMODEL-blokken geïllustreerd voor een configuratie als in Figuur 5.6. De situatie uit Figuur 5.13 wordt nog eens verduidelijkt met behulp van Figuur 5.5.

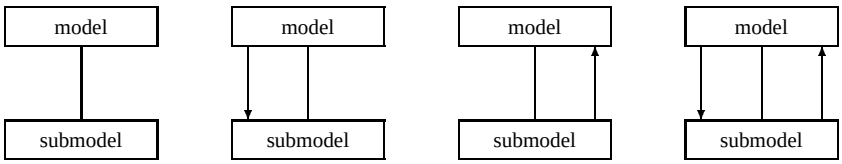

Figure 5.11: Transactiestroom tussen model en submodel

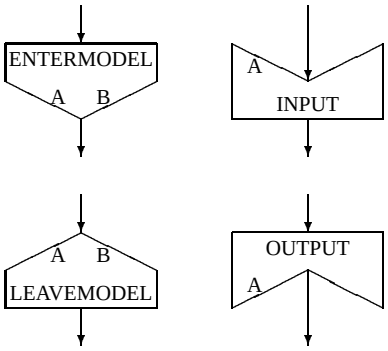

Figure 5.12: Symbool INPUT-, OUTPUT-, ENTERMODEL- en LEAVEMODEL-blok

```
 (1) MODEL Model1
 (2) A       SUBMODEL            Model2
 (3) B       SUBMODEL            Model3
 (4)         ...
 (5)         ENTERMODEL          A,In
 (6)         LEAVEMODEL          A,Out
 (7)         ENTERMODEL          B,In
 (8)         LEAVEMODEL          B,Out
 (9)         ...
(10) ENDMODEL
(11)
(12) MODEL Model2
(13)         INPUT               In
(14)         ...
(15)         OUTPUT              Out
(16) ENDMODEL
(17)
(18) MODEL Model3
(19)         INPUT               In
(20)         ...
(21)         OUTPUT              Out
(22) ENDMODEL
(23)
(24) COMMAND
(25)         ...
(26)         SIMULATE            Model1
(27)         ...
(28) ENDCOMMAND
```

Figure 5.13: Gebruik INPUT-, OUTPUT, ENTERMODEL- en LEAVEMODEL-blok

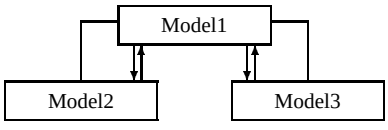

Figure 5.14: Hiërarchie en transactiestroom van vorige figuur

```
 (1) MODEL Warenhuis(time_type _tijd)
 (2) *
 (3) * Entiteitsdeclaraties
 (4) *
 (5) KLOK      SUBMODEL        Klok(_tijd)
 (6) *
 (7) * Blokdeclaraties
 (8) *
 (9)           GENERATE        10,7               Binnentreden winkel
(10)           ADVANCE         12,5               Winkelen
(11)           QUEUE           WACHT              Aansluiten bij wachtlijn
(12)           SEIZE           KASSA              Verkrijgen aandacht kassier
(13)           DEPART          WACHT              Verlaten van wachtlijn
(14)           ADVANCE         3,1                Afrekenen
(15)           RELEASE         KASSA              Opgeven aandacht kassier
(16)           TERMINATE       1                  Verlaten winkel
(17) ENDMODEL
(18) MODEL Klok(time_type _tijd)
(19)           GENERATE        "_tijd"
(20)           TERMINATE       1
(21) ENDMODEL
(22) COMMAND
(23) *
(24) * Commando's
(25) *
(26)           SIMULATE        Warenhuis(1000)    Installeer model
(27)           START           1                  Simulatie
(28)           PRINT           /                  Druk alle informatie af
(29)           END                                Beeindigen simulatie
(30) ENDCOMMAND
```

Figure 5.15: Geparameteriseerde modellen

5.6 Geparameteriseerde modellen

Om het gebruik van modelklassen zo flexibel mogelijk te maken kunnen bij de beschrijving van een modelklasse een aantal formele parameters voorzien worden. Deze parameters zullen dan een actuele waarde moeten krijgen bij het creëren van een instantie van de klasse. De manier waarop de formele en actuele parameters worden gespecifieerd, werd zodanig gekozen dat de analogie met de manier waarop dit in C++ gebeurt maximaal zou zijn. Bij de beschrijving van de modelklasse wordt het MODEL-statement - ná de naam van de klasse - uitgebreid met een formele parameter-lijst. Bij de creatie van een instantie van de klasse in een SUBMODEL- of SIMULATE-statement wordt - ná de naam van de klasse - de actuele parameter-lijst opgenomen. De formele parameters kunnen op dezelfde manier in HGPSS-statements worden gebruikt als gewone C++-expressies. Om de manier waarop geparameteriseerde modellen kunnen beschreven worden te verduidelijken, werd in Figuur 5.15 een aangepaste versie van Figuur 5.1 opgenomen. In tegenstelling met Figuur 5.1 wordt nu niet een bepaald aantal klanten gesimuleerd, maar gedurende een bepaalde tijd. De simulatie wordt afgebroken als de vooropgestelde tijd is verstreken. Het model `Warenhuis` maakt gebruik van een submodel met de naam `Klok` dat zal aangegeven wanneer de simulatie moet afgebroken worden. Bij het installeren van `Warenhuis` als hoofdmodel wordt als parameter de te simuleren tijd meegegeven. Deze waarde wordt dan doorgespeeld naar het submodel `Klok`.

5.7 Uitvoer

In GPSS wordt na afloop van de simulatie automatisch alle informatie uitgevoerd die tijdens deze simulatie vergaard werd in verband met de in het model aanwezige entiteiten. Er is geen GPSS-commando waarmee een beperking kan opgelegd worden zodat enkel deze informatie wordt uitgevoerd die interessant is. Afzonderlijke gegevens kunnen wel uitgevoerd worden tijdens de simulatie zelf door gebruik te maken van een PRINT-blok. In HGPSS zou dit zich vertalen naar een mogelijkheid om informatie te selecteren in de modelsecties, maar niet in de commandosectie. Om de uit te voeren gegevens zowel in de modelsecties als in de commandosectie te kunnen selecteren, werd in HGPSS het PRINT-commando ingevoerd. Dit commando heeft dezelfde syntax als het blok met dezelfde naam. Enkel de informatie geselecteerd met één of meerdere PRINT-commando's zal uitgevoerd worden. Als er in de commandosectie geen PRINT-commando voorkomt zal dus geen automatische uitvoer van alle informatie geschieden.

De naam van het bestand waarin de uitvoer zal terecht komen, kan ook vanuit de commandosectie worden gespecifieerd. Dit gebeurt door de naam van het bestand als tweede parameter op te geven bij het SIMULATE-commando. Als deze parameter wordt weggelaten, wordt een welbepaalde voor dergelijke gevallen voorziene naam gebruikt.

De uitvoer gegenereerd tijdens of na een HGPSS-simulatie ziet er anders uit dan deze gegenereerd door een GPSS-simulatie. Om dit feit te benadrukken werden de GPSS-mnemonics die in het PRINT-blok moeten worden gebruikt, in HGPSS vervangen door andere en sprekender mnemonics voor gebruik in het PRINT-blok en -commando.

5.8 Doorkruisen van de modelboom vanuit de commandosectie

In de HGPSS-commandosectie is terzelfdertijd maar één van de modelboom deel uitmakend model zichtbaar. De commando's die inwerken op entiteiten zullen dan ook inwerken op de entiteiten van dit zichtbare model. Initieel, met andere woorden na gebruik van het SIMULATE-commando, is het hoofdmodel zichtbaar. Door gebruik te maken van het DOWN-commando kan één van de submodellen van het momenteel zichtbare model als dusdanig worden geïnstalleerd. Daartoe moet ná het `DOWN`-sleutelwoord, de naam van de te selecteren submodel worden gespecifieerd. Het DOWN-commando laat dus toe om de modelboom af te dalen. De omgekeerde operatie wordt door het UP-commando bewerkstelligd. Het gebruik van dit commando maakt van het bovenliggende model, het zichtbare model.

Chapter 6

HGPSS++

6.1 Inleiding

6.1.1 Principes

Ter ondersteuning van een simulatie-taal als GPSS en meer specifiek HGPSS, is een stuk software nodig dat kan instaan voor de low-level afhandeling van het eigenlijke simulatie-proces. Een dergelijk stuk software bestaat typisch uit een verzameling functies en variabelen gebundeld in objecten, waarvan de werking en het gebruik bepaald worden door een controlerend orgaan. Het controlerend orgaan orchestreert de werking en het gebruik van deze basiseenheden zodaning dat een bepaalde aan het orgaan opgelegde taak tot uitvoering wordt gebracht. Deze uit te voeren taken worden meegedeeld vanuit hogere software-lagen via een interface bestaande uit een aantal gespecialiseerde functies.

Ter ondersteuning van HGPSS werd een software-omgeving ontwikkeld die de geschetste taken afhandelt en de naam *HGPSS++-kernel* meekreeg. De taal die gebruikt wordt om deze kernel aan te spreken, werd kortweg *HGPSS++* gedoopt. Deze taal bestaat uit niets anders dan de verzameling functie-aanroepen horende bij de interface-functies van de kernel.

De HGPSS++-kernel bestaat slechts uit één statisch object: het controlerend orgaan of de processor. De absolute klok, de relatieve klok en de beëindigingsteller maken deel uit van de processor. De processor zal interageren met entiteiten. Deze entiteiten zijn de implementaties van de abstracte entiteiten waarvan bij de modelspecificatie gebruik gemaakt wordt. Alle entiteiten worden dynamisch gecreëerd. Entiteiten behorende tot eenzelfde klasse worden gegroepeerd op een keten.

De verzameling van alle functies die kunnen gebruikt worden om de processor en de entiteiten aan de spreken, de HGPSS++-taal, kan opgedeeld worden in drie deelverzamelingen. Elke deelverzameling bevindt zich op een verschillend hiërarchisch niveau (Figuur 6.1).

- Op het hoogste niveau staan de functies die in rechtstreeks verband staan met de HGPSS-statements. Deze functies worden *gebruikersfuncties* genoemd. In de implementatie ervan worden functies behorende tot de twee onderste niveaus aangeroepen.

- Op het intermediaire niveau staan de functies die rechtstreeks de processor aanspreken. Deze functies worden dan ook *processorfuncties* genoemd. In de implementatie ervan kunnen functies van het laagste niveau aangeroepen worden.

- Het laagste niveau wordt gevormd door de *entiteitsfuncties*. Deze functies werken rechtstreeks in op entiteiten zonder eerst om te gaan langs de processor.

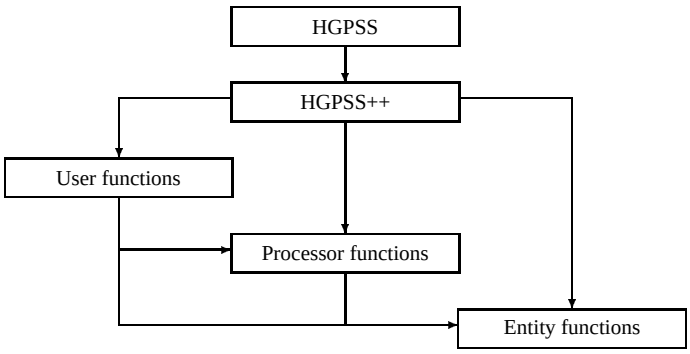

Figure 6.1: Hiërarchie der HGPSS++-functies

HGPSS++-kernel	Microprocessor environment
HGPSS	High-level language
HGPSS to HGPSS++ compiler	High-level language compiler
User functions	Symbolic assembly instructions
Processor functions	Binary assembly instructions
Entity functions	Microcode instructions
Entities	Resources
Processor	Central processing unit

Table 6.1: Analogie HGPSS++-kernel en microprocessor-omgeving

Het concept van de HGPSS++-kernel en de manieren om deze aan te spreken kunnen worden vergeleken met een microprocessor-omgeving zoals in Tabel 6.1 wordt geïllustreerd.

De HGPSS++-kernel zoals geschetst, werd in C++ uitgewerkt door de implementatie van een aantal klassen. De totale verzameling klassen kan in een aantal deelverzamelingen opgedeeld worden:

- De *ondersteuningsklassen* vormen de basis van het systeem. Zij worden ofwel als basisklasse voor andere klassen gebruikt, ofwel vormen instanties van één van de ondersteuningsklassen een lidobject van een ander object, zonder dat dit evenwel naar buiten toe expliciet tot uiting komt.

- De *systeemklassen* modelleren objecten die deel uitmaken van de processor. Aangezien deze objecten volledig opgenomen zijn in de processor, zullen ze naar buiten toe onzichtbaar zijn.

- De *entiteitsklassen* modelleren de entiteiten waarvan binnen het te simuleren systeem gebruik gemaakt wordt. Alle entiteiten zijn wat implementatie betreft op eenzelfde leest geschoeid. Typisch aan een entiteit is het identificatienummer. Elke entiteit bezit een binnen de klasse waartoe de entiteit behoort, uniek identificatienummer.

- Entiteiten behorende tot eenzelfde klasse worden geketend en vormen aldus zogenaamde entiteitsketens. De *entiteitsketenklassen* modelleren deze ketens.

- De *blokklassen* modelleren de originele GPSS-blokken en een aantal blokken specifiek voor HGPSS. Alhoewel blokken binnen de HGPSS++-kernel ook als entiteiten beschouwd worden, worden ze toch in een afzonderlijke groep ingedeeld omwille van de duidelijkheid.

- De *processorklasse* tenslotte vormt de implementatie van de processor.

6.1.2 Nomenclatuur

Binnen de broncode van de HGPSS++-kernel werden een aantal regels in acht genomen in verband met de nomenclatuur van onder andere variabelen, functies, klassen en types.

- De eerste letter van elk deel van de namen van klassen en functies die geen lid zijn van een object, is een hoofdletter.
- Aan de naam van een klasse wordt steeds `Class` toegevoegd.
- De eerste letter van elk deel van de namen van publieke lidvariabelen en publieke lidfuncties, is een hoofdletter.
- Voor de namen van niet-publieke lidvariabelen en niet-publieke lidfuncties worden uitsluitend kleine letters gebruikt.
- Formele functie-parameters worden voorafgegaan door een _.
- Een _ wordt ook gebruikt als eerste karakter van de naam van blokklassen als deze dezelfde is als de naam van een entiteitsklasse.
- De namen van gebruikersfuncties bestaan uitsluitend uit hoofdletters om duidelijk het verband met de HGPSS-statements aan te geven.
- Aan de namen van alle datatypes wordt `_type` toegevoegd.

6.1.3 Datatypes

In de broncode worden een aantal types gedeclareerd en gebruikt. Deze types kunnen in drie groepen verdeeld worden:

- De *systeemtypes* zijn basistypes die gebruikt worden in plaats van de standaard numerieke C++-types als `char`, `int` en `float`.
  - `boolean_type` wordt gebruikt als type voor booleaanse variabelen.
  - `count_type` wordt gebruikt als type voor variabelen die een teller voorstellen die enkel positieve gehele waarden kan aannemen.
  - `number_type` wordt gebruikt als type voor identificatienummers van entiteiten. Deze nummers moeten gehele positieve waarden aannemen. Variabelen van het type `number_type` waar om één of andere reden geen relevante waarde kan aan toegekend worden, worden veelal geïnitialiseerd met de macro `NUMBER_NONE`.
  - `time_type` wordt gebruikt als type voor variabelen die de simulatie-tijd of gerelateerde tijden voorstellen. Tijden hoeven niet geheel te zijn maar moeten wel steeds positief blijven. Variabelen van het type `time_type` waar om één of andere reden geen relevante waarde kan aan toegekend worden, worden veelal geïnitialiseerd met de macro `TIME_NONE`.
  - `value_type` tenslotte wordt gebruikt als type voor variabelen die resultaten van berekeningen of één van de vorige grootheden voorstellen. Dergelijke variabelen hoeven niet geheel te zijn en mogen zowel positieve als negatieve waarden aannemen. Variabelen van het type `value_type` waar om één of andere reden geen relevante waarde kan aan toegekend worden, worden veelal geïnitialiseerd met de macro `VALUE_NONE`.

- De *enumeratietypes* worden gebruikt voor variabelen die slechts waarden kunnen aannemen uit een beperkte verzameling. De enumeratietypes zijn;
  - `buffer_option_type`
  - `fullword_option_type`
  - `function_type`
  - `gate_type`
  - `halfword_option_type`
  - `information_type`
  - `logic_type`
  - `logical_type`
  - `operation_type`
  - `print_type`
  - `priority_option_type`
  - `remove_option_type`
  - `select_type`
  - `SNA_type`
  - `state_type`
  - `test_type`
  - `wait_for_type`
  - `weighted_option_type`
  - `word_type`

  Deze types worden meestal gebruikt voor het weergeven van de parameters of onderdelen van parameters horende bij HGPSS-statements. De elementen van de verzamelingen aangeduid door de enumeratietypes werden allemaal voorzien van een naam. Deze naam bestaat volledig uit hoofdletters. Het eerste deel van de naam is dezelfde als de naam van het type om verwarring met elementen van andere verzamelingen te vermijden. Het tweede deel is gescheiden van het eerste door een _ en refereert naar de eigenlijke betekenis van het element. Elke verzameling heeft één element dat gebruikt wordt om aan te geven dat er in een bepaalde omstandigheid geen ander relevant element kan opgegeven worden. Dit element wordt typisch als don't care- of default-element gebruikt en heeft een naam bestaande uit de naam van het type gevolgd door `_NONE`.

- De laatste categorie, de *structuurtypes,* bevat types die de vorm hebben van een structuur bestaande uit een aantal velden. De categorie kan op haar beurt ingedeeld worden in drie deelcategorieën:
  - Variabelen van het type `entity_chains_type` zijn structuren met als velden wijzers naar alle lokale entiteitsketens van een model.
  - In de meeste HGPSS-statements komen parameters voor. Deze parameters zijn meestal ofwel een getal, ofwel bestaan ze uit een mnemonic wijzend op een standard numerical attribute, een identificatienummer en een teken dat aangeeft of er indirectie moet worden toegepast. Het gebruik van *parametertypes* laat toe om dergelijke parameters weer te geven in de vorm van een structuur.

    - `initial_type` wordt gebruikt als type voor parameters zoals die voorkomen in het `INITIAL`-commando.
    - `link_type` wordt gebruikt als type voor parameters zoals die voorkomen in de `LINK`-blokdeclaratie.
    - `parameter_type` wordt in alle andere gevallen waar parameters moeten weergegeven worden, gebruikt.
  - Van een aantal entiteiten kan informatie opgevraagd worden in de vorm van een structuur. De types die hiervoor gebruikt worden zijn *informatietypes.* De informatietypes zijn de volgende:
    - `block_information_type`
    - `facility_information_type`
    - `logic_switch_information_type`
    - `m_savevalue_information_type`
    - `queue_information_type`
    - `savevalue_information_type`
    - `storage_information_type`
    - `table_information_type`
    - `user_chain_information_type`

6.1.4 Ondersteuningsfuncties

Een aantal functies zijn naar buiten toe van weinig belang en worden enkel gebruikt ter ondersteuning van sommige activiteiten.

- `boolean_type parameter_type::operator==(parameter_type _parameter)`: Definitie van de operator `==` op twee argumenten van het type `parameter_type`. Twee variabelen van dit type zullen als gelijk beschouwd worden als al hun velden gelijk zijn.

- `boolean_type parameter_type::operator!=(parameter_type _parameter)`: Definitie van de operator `!=` op twee argumenten van het type `parameter_type`. Twee variabelen van dit type zullen als ongelijk beschouwd worden als één of meer velden verschillend zijn.

- `void *operator new(size_t _size)`: Herdefinitie van de operator `new` zodat naast het toekennen van geheugen ook gecontroleerd wordt of de gevraagde hoeveelheid geheugen vrij is.

- `void operator delete(void *_memory_area)`: Herdefinitie van de operator `delete` zodat vóór het vrijgeven van het geheugen gecontroleerd wordt of de wijzer `_memory_area` niet gelijk is aan `NULL`.

- ```
  initial_type I(initial_enum _initial_kind,
                 number_type  _entity_nbr,
                 number_type  _row_nbr=NUMBER_NONE,
                 number_type  _column_nbr=NUMBER_NONE):
  ```

  Creëert een structuur van het type `initial_type` waarbij in de velden respectievelijk `_initial_kind`, `_entity_nbr`, `_row_nbr` en `_column_nbr` wordt ingevuld.
  ```

- `link_type L(link_enum _link_kind=LINK_NONE,`
  `number_type _parameter_nbr=NUMBER_NONE):`

  Creëert een structuur van het type `link_type` waarbij in de velden respectievelijk `_link_kind` en `_parameter_nbr` wordt ingevuld.

- `parameter_type P(parameter_enum _parameter_kind=PARAMETER_NONE,`
  `value_type _value=VALUE_NONE,`
  `SNA_type _SNA_kind=SNA_NONE,`
  `number_type _row_nbr=NUMBER_NONE,`
  `number_type _column_nbr=NUMBER_NONE):`

  Creëert een structuur van het type `parameter_type` waarbij in de velden respectievelijk `_parameter_kind`, `_value`, `_SNA_kind`, `_row_nbr` en `_column_nbr` wordt ingevuld.

6.2 Klassen

Alle klassen waarvan in de HGPSS++-kernel gebruik gemaakt wordt, worden hieronder besproken. De klassen zijn onderverdeeld naargelang de groepen waartoe ze behoren. Voor elke klasse wordt eerst kort het doel geschetst. Daarna volgt een overzicht van alle lidvariabelen en lidfuncties. Hierbij wordt telkens de C++-declaratie van de variabele of functie opgenomen. Bij elk item wordt uitleg verschaft behalve in het geval van accessor-functies, gezien het doel van deze functies triviaal is. Indien er voor een bepaalde variabele zowel een functie is die lees-toegang tot de variabele verleent als een functie die schrijf-toegang verleent, worden de namen van deze functies respectievelijk voorafgegaan door `Get` en `Set`. In de gevallen waarin slechts één van beide mogelijkheden optreedt, blijkt uit de declaratie van de functie of het gaat om een functie die lees- of schrijf-toegang verleent.

Uitzonderingen op al deze regels zijn mogelijk maar worden steeds eerst aangegeven. Een overzicht van de HGPSS++-klassen en hun hiërarchie is in Figuur 6.2 weergegeven.

6.2.1 Ondersteuningsklassen

De klassen ingedeeld bij de ondersteuningsklassen zijn deze die als basisklasse voor andere klassen gebruikt worden of waarvan de instanties lidobjecten van een ander object zijn.

- Instanties van `DataSumClass` zijn data-collectoren. Het hoofddoel van de objecten is de som te bepalen van vergaarde waarden.
- Instanties `DataIntegralClass` zijn eveneens data-collectoren. De objecten zijn echter eerder gericht op het bepalen van de integraal over de tijd van de vergaarde waarden.
- `ElementClass` is de basisklasse van alle andere klassen die in een keten op te nemen objecten modelleren.
- Instanties van `MonitorElementClass` worden gebruikt om bepaalde gegevens tijdelijk vast te houden.
- Instanties van `EventClass` zijn de in de event lists op te nemen event notices.
- `EntityClass` is de basisklasse van alle entiteitsklassen.

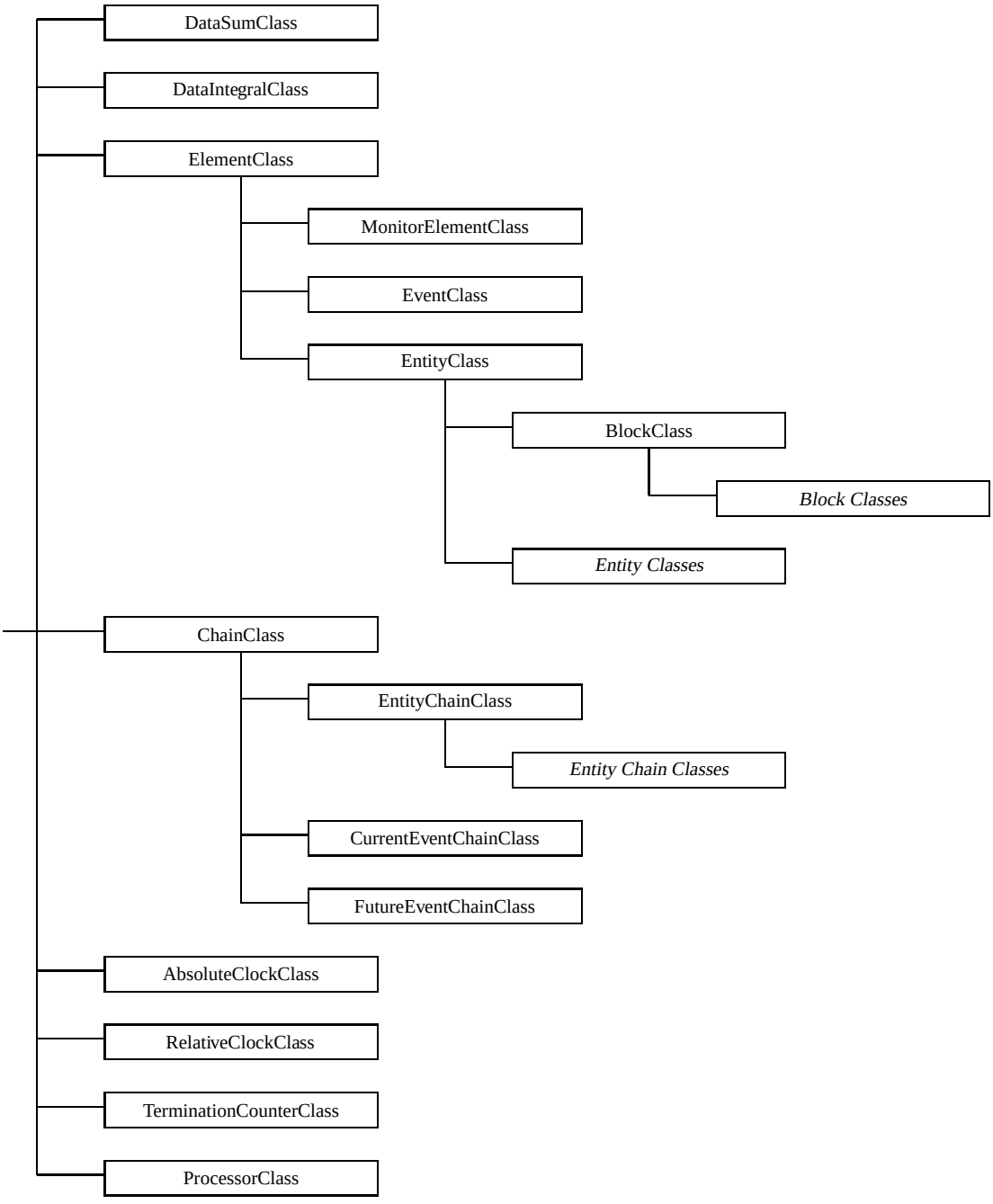

Figure 6.2: Hiërarchie der HGPSS++-klassen

- `ChainClass` is de basisklasse van alle ketenklassen. Deze ketens kunnen ketens van entiteiten zijn, maar ook andere.
- `EntityChainClass` is de basisklasse van alle entiteitsketenklassen.
- `ModelClass` is de basisklasse van de door de modelbouwer te construeren modellen.

DataSumClass

Objecten (Figuur 6.3) lezen waarden in en houden op een dynamische wijze een aantal statistieken hierover bij:

- Het aantal ingelezen waarden.
- Het aantal van nul verschillende waarden.
- De som van de waarden.
- De som van de kwadraten van de waarden.
- Het gemiddelde van de waarden.
- Het gemiddelde van de van nul verschillende waarden.
- Het minimum van alle waarden.
- Het maximum van alle waarden.
- De standaard afwijking van de waarden.
- De standaard afwijking van de van nul verschillende waarden.

De huidige stand van de statistieken kan op elk ogenblik worden uitgelezen. De statistieken worden bijgehouden in een aantal velden van een structuur. Een bepaalde toestand kan worden onthouden terwijl het object naar een nieuwe toestand evolueert door de initiële toestand vast te houden in een hulp-structuur.

Lidvariabelen

- `struct {...} state`: Houdt de huidige toestand bij.
- `struct {...} saved_state`: Houdt een vorige toestand bij.

Lidfuncties

- `DataSumClass(void)`: Initialiseert alle velden van `state` en `saved_state` met nul.
- `void Reset(void)`: Initialiseert de velden van `state` terwijl `saved_state` ongewijzigd blijft.
- `void Store(void)`: Bewaart de huidige toestand door de inhoud van `state` te kopiëren naar `saved_state`.
- `void Recall(void)`: Installeert een vastgehouden toestand opnieuw door de inhoud van `saved_state` te kopiëren naar `state`.

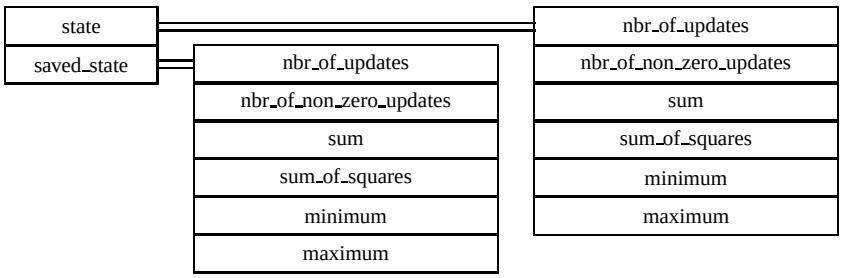

Figure 6.3: DataSumClass

- `void Update(value_type _value)`: Leest een nieuwe waarde in en bepaalt uitgaande van deze waarde en de vorigen, een nieuwe waarde voor de velden van `state`.
- `count_type NbrOfUpdates(void)`
- `count_type NbrOfNonZeroUpdates(void)`
- `value_type Sum(void)`
- `value_type SumOfSquares(void)`
- `value_type Average(void)`
- `value_type NonZeroAverage(void)`
- `value_type Minimum(void)`
- `value_type Maximum(void)`
- `value_type Deviation(void)`
- `value_type NonZeroDeviation(void)`

DataIntegralClass

Objecten (Figuur 6.4) lezen waarden in en houden op een dynamische wijze een aantal statistieken hierover bij:

- Het aantal ingelezen waarden.
- Het aantal van nul verschillende waarden.
- Het gemiddelde van de waarden.
- Het gemiddelde van de van nul verschillende waarden.
- De integraal van de waarden. Deze integraal is de som van de produkten van de waarden met de tijd tussen de registratie van de waarde en de registratie van de vorige waarde.
- Het minimum van alle waarden.

- Het maximum van alle waarden.

De huidige stand van de statistieken kan op elk ogenblik worden uitgelezen. De statistieken worden bijgehouden in een aantal velden van een structuur. Een bepaalde toestand kan worden onthouden terwijl het object naar een nieuwe toestand evolueert door de initiële toestand vast te houden in een hulp-structuur.

Lidvariabelen

- `struct {...} state`: Houdt de huidige toestand bij.
- `struct {...} saved_state`: Houdt een vorige toestand bij.

Lidfuncties

- `DataIntegralClass(void)`: Initialiseert alle velden van `state` en `saved_state` met nul.
- `void Reset(void)`: Initialiseert de velden van `state` terwijl `saved_state` ongewijzigd blijft.
- `void Store(void)`: Bewaart de huidige toestand door de inhoud van `state` te kopiëren naar `saved_state`.
- `void Recall(void)`: Installeert een vastgehouden toestand opnieuw door de inhoud van `saved_state` te kopiëren naar `state`.
- `void Update(value_type _value)`: Leest een nieuwe waarde in en bepaalt uitgaande van deze waarde en de vorigen, een nieuwe waarde voor de velden van `state`.
- `count_type NbrOfUpdates(void)`
- `count_type NbrOfNonZeroUpdates(void)`
- `value_type Average(void)`
- `value_type NonZeroAverage(void)`
- `value_type Integral(void)`
- `value_type Minimum(void)`
- `value_type Maximum(void)`

ElementClass

Objecten (Figuur 6.5) zijn erg eenvoudig maar vormen de basis voor alle objecten die in een keten moeten opgenomen worden. De enige lidvariabelen zijn een wijzer naar een vorig en een volgend object van dezelfde klasse.

Lidvariabelen

- `ElementClass *Previous`: Wijzer naar vorige object in keten.
- `ElementClass *Next`: Wijzer naar volgende object in keten.

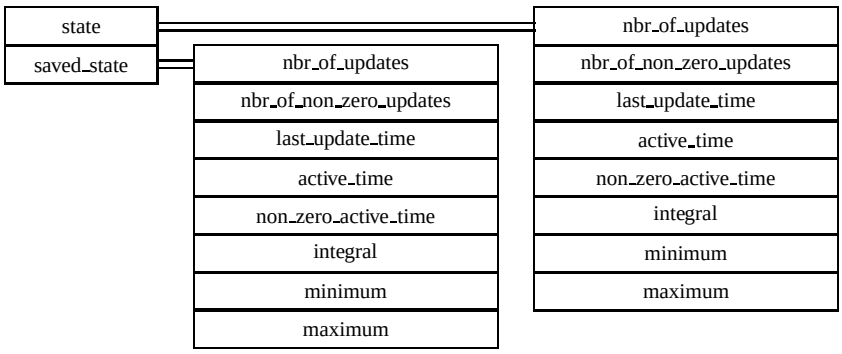

Figure 6.4: DataIntegralClass

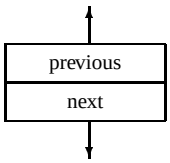

Figure 6.5: ElementClass

Lidfuncties

- `ElementClass(void)`: Initialiseert van de wijzers met `NULL`.

MonitorElementClass

Objecten (Figuur 6.6) zijn afgeleid van `ElementClass`. Ze worden in een aantal uiteenlopende omstandigenheden gebruikt wanneer moet bijgehouden worden op welk ogenblik een bepaalde gebeurtenis met betrekking tot een bepaalde transactie heeft plaatsgevonden.

Lidvariabelen

- `time_type time`: Tijdstip waarop de gebeurtenis plaatsvond.
- `number_type transaction_nbr`: Identificatienummer van de transactie bij de gebeurtenis betrokken.

Lidfuncties

- `MonitorElementClass(time_type _time,number_type _transaction_nbr)`: Initialiseert `time` met `_time` en `transaction_nbr` met `_transaction_nbr`.
- `void SetTime(time_type _time)`
- `time_type GetTime(void)`
- `number_type TransactionNbr(void)`

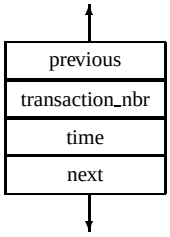

Figure 6.6: MonitorElementClass

EventClass

Objecten (Figuur 6.7) zijn afgeleid van `ElementClass` en worden gebruikt om event notices voor te stellen.

Lidvariabelen

- `time_type time`: Tijdstip waarop het event gescheduled is.
- `TransactionClass *transaction`: Wijzer naar de transactie waarop het event betrekking heeft.
- `boolean_type current_event`: Geeft aan of het event zich op de current of future event list bevindt.

Lidfuncties

- `EventClass(time_type _time,TransactionClass *_transaction)`: Initialiseert `time` en `transaction` met `_time` en `_transaction`, `current_event` met `FALSE` en koppelt het event aan de transactie waarmee het correspondeert door vanuit de transactie een wijzer naar het event te initialiseren.
- `time_type Time(void)`
- `TransactionClass *Transaction(void)`
- `void SetCurrentEvent(boolean_type _current_event)`
- `boolean_type GetCurrentEvent(void)`

EntityClass

Objecten (Figuur 6.8) zijn erg eenvoudig, afgeleid van `ElementClass` en de basis van alle entiteiten. Typisch voor alle entiteiten is hun identificatienummer. Dit nummer moet uniek zijn binnen de klasse waartoe de entiteit behoort en mag de waarde `NUMBER_NONE` niet aannemen.

Lidvariabelen

- `number_type nbr`: Identificatienummer om elke entiteit uniek te kunnen refereren.

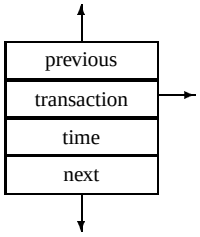

Figure 6.7: EventClass

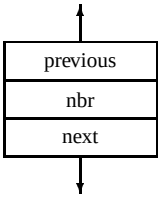

Figure 6.8: EntityClass

Lidfuncties

- `number_type Nbr(void)`

ChainClass

Objecten (Figuur 6.9) worden gebruikt om wijzers naar het eerste en laatste element van een keten bij te houden, alsook de lengte van de keten. De klasse is de basisklasse voor alle entiteitsketenklassen en enkele andere ketenklassen.

Lidvariabelen

- `count_type length`: Lengte van de keten.
- `ElementClass first`: Wijzer naar het eerste element van de keten.
- `ElementClass last`: Wijzer naar het laatste element van de keten.

Lidfuncties

- `ChainClass(void)`: Initialiseert `first` en `last` met `NULL` en `length` met nul.
- `count_type Length(void)`
- `ElementClass *First(void)`
- `ElementClass *Last(void)`

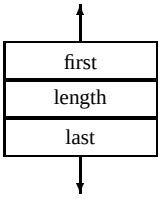

Figure 6.9: ChainClass

- `boolean_type Empty(void)`: Geeft aan of keten leeg is, wat neerkomt op lengte nul.
- `void InsertBefore(ElementClass *_next_element,ElementClass *_element)`: Brengt een extra element aangeduid door `_element` in de keten, vóór het element aangeduid door `_next_element`.
- `void InsertAfter(ElementClass *_previous_element, ElementClass *_element)`: Brengt een extra element aangeduid door `_element` in de keten, ná het element aangeduid door `_previous_element`.
- `void Append(ElementClass *_element)`: Voegt een element aangeduid door `_element` toe aan de keten, door het ná het laatste element van de keten te plaatsen.
- `void Prepend(ElementClass *_element)`: Voegt een element aangeduid door `_element` toe aan de keten, door het vóór het eerste element van de keten te plaatsen.
- `void Remove(ElementClass *_element)`: Verwijdert het element aangeduid door `_element` uit de keten.
- `ElementClass *RemoveFirst(void)`: Verwijdert het eerste element uit de keten en levert een wijzer naar dit element.
- `ElementClass *RemoveLast(void)`: Verwijdert het laatste element uit de keten en levert een wijzer naar dit element.

EntityChainClass

Objecten (Figuur 6.10) zijn afgeleid van `ChainClass`. `EntityChainClass` is de basisklasse van alle entiteitsketenklassen. Er zijn geen additionele lidvariabelen ten opzichte van `ChainClass`.

Lidfuncties

- `void Register(EntityClass *_entity)`: Registreert de entiteit aangeduid door `_entity`. De entiteit wordt met andere woorden opgenomen in de keten. Hierbij wordt ervoor gezorgd dat twee entiteiten met hetzelfde identificatienummer niet kunnen voorkomen in de keten en dat de entiteiten gesorteerd zijn in opklimmende volgorde van identificatienummer.
- `EntityClass *Entity(number_type _nbr)`: Levert een wijzer naar de entiteit met identificatienummer `_nbr`.

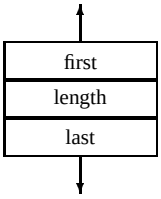

Figure 6.10: EntityChainClass

- `boolean_type Present(number_type _nbr)`: Geeft aan of de entiteit met `_nbr` als identificatienummmer zich in de keten bevindt.

ModelClass

Deze klasse (Figuur 6.11) is de basis voor de door de modelbouwer geconstrueerde modellen. Belangrijk is dat een object van deze klasse een wijzer bezit naar de verzameling lokale entiteitsketens. Dit zijn de ketens die alle entiteiten bevatten waaruit het model bestaat.

Lidvariabelen

- `entity_chains_type *entity_chains`: Wijzer naar een structuur die wijzers naar alle lokale entiteitsketens bevat.
- `number_type submodel_nbr`: Identificatienummer van het object als submodel van het bovenliggende model.
- `ModelClass *supermodel`: Wijzer naar het model waarvan het object een submodel is.

Lidfuncties

- `ModelClass(void)`: Creëert de lokale entiteitsketens, initialiseert `submodel_nbr` met `NUMBER_NONE` en `supermodel` met het door de processor als actief beschouwde model. Een object van `ModelClass` moet dus geconstrueerd worden op het ogenblik dat het bovenliggende model actief is.
- `~ModelClass(void)`: Verwijdert de lokale entiteitsketens en alle entiteiten die zich op deze ketens bevinden uit het geheugen.
- `void StartConstruct(void)`: Declareert het object als actief model aan de processor zodat alle entiteiten die vanaf nu geregistreerd worden als lokaal ten opzichte van het object zullen beschouwd worden.
- `void EndConstruct(void)`: Draagt de processor op om vanaf nu het bovenliggende model terug als actief te beschouwen. Geregistreerde entiteiten zullen niet meer als lokaal ten opzichte van het object beschouwd worden.
- `entity_chains_type *EntityChains(void)`
- `void SetSubModelNbr(number_type _submodel_nbr)`

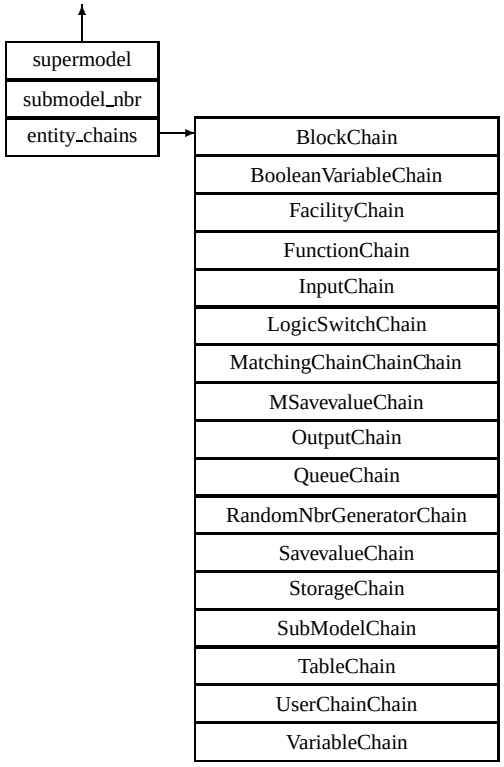

Figure 6.11: ModelClass

- `number_type GetSubModelNbr(void)`
- `ModelClass *SuperModel(void)`

6.2.2 Systeemklassen

De klassen ingedeeld bij de systeemklassen modelleren enkele lidobjecten van de processor. Van elke systeemklasse wordt slechts één instantie gecreëerd, dit bij de creatie van de processor. De systeemklassen zijn de volgende:

- `AbsoluteClockClass`
- `RelativeClockClass`
- `TerminationCounterClass`
- `CurrentEventChainClass`
- `FutureEventChainClass`

AbsoluteClockClass

Een instantie van deze klasse is de absolute klok waarvan de processor gebruik maakt. Deze klok kan enkel vooruit gedraaid en geïnitialiseerd worden. Bij het begin van een nieuwe simulatie-sessie wordt de absolute klok geïnitialiseerd.

Lidvariabelen

- `time_type time`: Huidige waarde van de absolute klok.

Lidfuncties

- `AbsoluteClockClass(void)`: Initialiseert `time` met nul.
- `void Print(ofstream *_output_file)`: Drukt de waarde van de absolute klok af in het bestand aangeduid door `_output_file` .
- `void Reset(void)`: Initialiseert `time` met nul.
- `void Update(time_type _time)`: Stelt de klok in op `_time`.
- `time_type Time(void)`

RelativeClockClass

Een instantie van deze klasse is de relatieve klok waarvan de processor gebruik maakt. Deze klok kan enkel vooruit gedraaid en geïnitialiseerd worden. Bij het begin van een nieuwe simulatie-taak binnen een sessie wordt de relatieve klok geïnitialiseerd.

Lidvariabelen

- `time_type time`: Huidige waarde van de relatieve klok.

Lidfuncties

- `RelativeClockClass(void)`: Initialiseert `time` met nul.
- `void Print(ofstream *_output_file)`: Drukt de waarde van de relatieve klok af in het bestand aangeduid door `_output_file`.
- `void Reset(void)`: Initialiseert `time` met nul.
- `void Update(time_type _time)`: Stelt de klok in op de huidige waarde vermeerderd met `_time`.
- `time_type Time(void)`

TerminationCounterClass

Een instantie van deze klasse is de beëindigingsteller waarvan de processor gebruik maakt. Deze teller kan enkel op een bepaalde waarde geïnitialiseerd worden om daarna telkens met variërende waarden te worden gedecrementeerd, totdat de teller negatief of nul wordt.

Lidvariabelen

- `value_type value`: Huidige waarde van de beëindigingsteller.

Lidfuncties

- `void Print(oftream *_output_file)`: Drukt de waarde van de teller af in het bestand aangeduid door `_output_file`.
- `void Reset(value_type _value)`: Initialiseert de teller met `_value`.
- `void Update(value_type _decrement)`: Decrementeert de teller met `_value`.
- `boolean_type Terminated(void)`: Geeft aan of de teller afgelopen, met andere woorden niet positief, is.

CurrentEventChainClass

Een instantie van deze klasse is de current event chain waarvan de processor gebruik maakt. De klasse is afgeleid van `ChainClass`. Event notices deel uitmakend van deze keten zijn gesorteerd op prioriteit van de gerefereerde transacties. Binnen een prioriteitsklasse zijn de event notices gesorteerd via het FIFO-principe. Event notices met de hoogste prioriteit bevinden zich vooraan in de keten. Binnen een prioriteitsklasse bevindt het eerst geschedulede event notice zich vooraan. Een object heeft geen additionele lidvariabelen ten opzichte van objecten van `ChainClass`.

Lidfuncties

- `~CurrentEventChainClass(void)`: Verwijdert alle event notices opgenomen in de keten uit het geheugen.
- `void Clear(void)`: Verwijdert alle event notices opgenomen in de keten uit het geheugen.
- `void Print(ofstream *_output_file)`: Drukt voor elk event notice opgenomen in de keten informatie af in het bestand aangeduid door `_output_file`.
- `void Schedule(EventClass *_event)`: Neemt het event notice aangeduid door `_event` op de geschikte plaats op in de keten.

FutureEventChainClass

Een instantie van deze klasse is de future event chain waarvan de processor gebruik maakt. De klasse is afgeleid van `ChainClass`. Event notices deel uitmakend van deze keten zijn gesorteerd op tijd en daarna op prioriteit van de gerefereerde transacties. Binnen een prioriteitsklasse zijn de event notices gesorteerd via het FIFO-principe. Event notices voor events gescheduled op het vroegste tijdstip bevinden zich vooraan in de keten. Binnen een tijdsklasse bevinden de hoogste prioriteiten zich vooraan. Binnen een prioriteitsklasse tenslotte bevindt het eerst geschedulede event notice zich vooraan. Een object heeft geen additionele lidvariabelen ten opzichte van objecten van `ChainClass`.

Lidfuncties

- `~FutureEventChainClass(void)`: Verwijdert alle event notices opgenomen in de keten uit het geheugen.
- `void Clear(void)`: Verwijdert alle event notices opgenomen in de keten uit het geheugen.
- `void Print(ofstream *_output_file)`: Drukt voor elk event notice opgenomen in de keten informatie af in het bestand aangeduid door `_output_file`.
- `void Schedule(EventClass *_event)`: Neemt een event notice aangeduid door `_event` op de geschikte plaats op in de keten.

6.2.3 Entiteitsklassen

De klassieke GPSS-entiteiten en enkele entiteiten specifiek aan HGPSS worden door de entiteitsklassen gemodelleerd. Alle entiteitsklassen hebben `EntityClass` als basisklasse. Typisch voor de entiteiten is dat ze een binnen de entiteitsklasse uniek en van `NUMBER_NONE` verschillend identificatienummer moeten bezitten. De klassen behorende tot de groep van de entiteitsklassen zijn:

- `BlockClass`
- `BooleanVariableClass`
- `EnterClass`
- `ExternClass`
- `FacilityClass`
- `FunctionClass`
- `InputClass`
- `LeaveClass`
- `LogicSwitchClass`
- `MatchingChainClass`
- `MatchingChainChainClass`
- `MSavevalueClass`
- `OutputClass`
- `QueueClass`
- `RandomNbrGeneratorClass`
- `SavevalueClass`
- `StorageClass`
- `SubModelClass`

- `TableClass`
- `TransactionClass`
- `UserChainClass`
- `VariableClass`

Alle entiteiten bezitten een lidfunctie (`void Print(ofstream *_output_file)`) die, wanneer aangeroepen, informatie over de entiteit afdrukt in een bestand waarvoor een bestandswijzer in de functie-aanroep wordt meegegeven. Sommige entiteiten bezitten daarenboven ook een functie (`..._information_type *Information(`) die een structuur creëert van één van de informatietypes. Deze structuur wordt gevuld met informatie over de huidige toestand van de entiteit.

BlockClass

Deze klasse is een virtuele basisklasse, dit betekent dat er geen instanties van de klasse zelf maar wel van afgeleide klassen kunnen worden gecreëerd. Alle blokklassen zijn van `BlockClass` afgeleid.

Lidvariabelen

- `number_type next_block_nbr`: Identificatienummer van het sequentiële blok.
- `char name[MAX_BLOCK_NAME_LENGTH+1]`: Karaktersliert die de klassenaam van het blok weergeeft.
- `count_type current`: Het aantal transacties die zich momenteel in het blok bevinden.
- `count_type total`: Het totale aantal transacties die het blok hebben aangedaan sinds de laatste initialisatie van de teller.

Lidfuncties

- `BlockClass(void)`: Initialiseert het identificatienummer en `next_block_nbr` met `NUMBER_NONE`, de naam met `"NONE"` en `current` en `total` met nul.
- `void Clear(void)`: Deze functie wordt aangeroepen door de processor bij de uitvoering van het CLEAR-commando. `current` en `total` worden met nul geïnitialiseerd.
- `void Reset(void)`: Deze functie wordt aangeroepen door de processor bij de uitvoering van het RESET-commando. `total` wordt met nul geïnitialiseerd.
- `block_information_type *Information(void)`
- `void Print(ofstream *_output_file)`
- `void Arrival(void)`: Deze functie wordt door de processor aangeroepen als een transactie het blok betreedt. `current` en `total` worden met één geïncrementeerd. De transactie wordt ervan verwittigd dat het zich momenteel in het blok bevindt met als identificatienummer, het identificatienummer van het object. De transactie wordt er ook van verwittigd dat het volgende blok waarschijnlijk het blok aangeduid door `next_block_nbr` zal zijn.
- `void Departure(void)`: Deze functie wordt aangeroepen door de processor als een transactie het blok verlaat. `current` wordt met één gedecrementeerd.

- `virtual void Check(boolean_type  &_available,`
  `wait_for_type &_wait_for,`
  `number_type   &_wait_for_nbr)=0:`

  Deze functie moet door de afgeleide blokklassen verplicht worden gedefinieerd. De functie zal door de processor worden aangeroepen om na te gaan of het blok de volgende transactie kan ontvangen. De referentie-parameters `_available`, `_wait_for` en `_wait_for_nbr` moeten op een geschikte waarde worden ingesteld. Via `_available` wordt aangegeven of het blok de transactie kan ontvangen. Als dit niet het geval is moet via `_wait_for` en `_wait_for_nbr` worden aangegeven waarop de transactie moet wachten om het blok te mogen betreden.

- `virtual void Body(void)=0`: Deze functie moet verplicht worden gedefinieerd door de afgeleide blokklassen. Deze functie wordt door de processor aangeroepen als een transactie het blok betreedt en geeft de eigenlijke functionaliteit van het blok weer.

- `number_type NextBlockNbr(void)`

- `char *Name(void)`

- `count_type Total(void)`

- `count_type Current(void)`

BooleanVariableClass

Objecten zijn booleaanse variabelen.

Lidvariabelen

- `boolean_type (*boolean_variable)(void)`: Wijzer naar een door de modelbouwer geconstrueerde functie. Deze functie zal worden uitgevoerd als de booleaanse variabele wordt gebruikt in een model. De functie mag geen parameters hebben en moet een resultaat van het type `boolean_type` teruggeven.

Lidfuncties

- `BooleanVariableClass(number_type    _nbr,`
  `boolean_type (*_boolean_variable)(void)):`

  Initialiseert het identificatienummer met `_nbr` en `boolean_variable` met `_boolean_variable`.

- `void Print(ofstream *_output_file)`

- `boolean_type Value(void)`: Voert de functie aangeduid door `boolean_variable` uit en geeft het resultaat terug.

EnterClass

Objecten worden gebruikt om de plaatsen binnen een model te markeren waar transacties vanuit het model in een submodel worden gebracht. De objecten hebben geen eigenlijke functionaliteit, hun enige nut is om bij de uitvoer informatie te krijgen over de verschillende plaatsen waar transacties het model verlaten en om te beletten dat op meerdere plaatsen in het model transacties naar eenzelfde ingang van eenzelfde submodel worden geleid.

Lidvariabelen

- `number_type block_nbr`: Identificatienummer van het blok waar de transacties het model verlaten.

Lidfuncties

- `EnterClass(number_type _nbr,number_type _block_nbr)`: Initialisatie van het identificatienummer met `_nbr` en van `block_nbr` met `_block_nbr`.
- `void Print(ofstream *_output_file)`
- `number_type BlockNbr(void)`

ExternClass

Objecten zijn elementen van de door de processor bijgehouden keten die alle op elk actief simulatietijdspunt te activeren functies weergeeft.

Lidvariabelen

- `void (*body)(void)`: Wijzer naar op elk actief simulatie-tijdspunt te activeren functie. Deze functie mag geen parameters hebben en mag geen resultaat teruggeven.

Lidfuncties

- `ExternClass(number_type _nbr,void (*_body)(void))`: Initialiseert het identificatienummer met `_nbr` en `body` met `_body`.
- `void Print(ofstream *_output_file)`
- `void Body(void)`: Roept de functie aangeduid door `body` aan.

FacilityClass

Objecten zijn facilities. Facilities kunnen aan maximaal één transactie toegekend zijn. Transacties kunnen echter wel verplicht worden om voor onbepaalde tijd de facility af te staan. De facility houdt zelf statistieken bij in verband met de gemiddelde duur van de toekenningen en de toekenningsgraad.

Lidvariabelen

- `MonitorElementClass *facility`: Wijzer naar een monitor-element. Dit element bevat het identificatienummer van de transactie waaraan de facility momenteel is toegekend en het tijdstip waarop deze transactie de facility in handen kreeg.
- `ChainClass interrupt_chain`: Verdringingsketen, dit is de keten van monitor-elementen corresponderende met transacties die de facility tijdelijk hebben moeten afstaan ten gevolge van verdringing door een andere transactie.
- `boolean_type preempted`: Geeft aan of de facility zich in de verdringingstoestand bevindt.

- `DataSumClass time`: Data-collectie object voor het verzamelen van gegevens omtrent de tijd dat de facility toegekend is aan transacties.
- `DataIntegralClass content`: Data-collecte object voor het verzamelen van gegevens omtrent de bezetting of toekenning van de facility.

Lidfuncties

- `FacilityClass(number_type _nbr)`: Initialiseert het identificatienummer met `_nbr`, `facility` met `NULL` en `preempted` met `FALSE`.
- `~FacilityClass(void)`: Verwijdert het monitor-element en de verdringingsketen uit het geheugen.
- `void Reset(void)`: Deze functie wordt door de processor aangeroepen bij de verwerking van het RESET-commando. De twee data-collectoren worden geïnitialiseerd.
- `facility_information_type *Information(void)`
- `void Print(ofstream *)`
- `void Seize(number_type _transaction_nbr)`: Kent de facility toe aan de transactie met identificatienummer `_transaction_nbr`. De transacties die ergens in het model wachten op het in gebruik nemen van de facility worden ervan op de hoogte gebracht dat ze hun weg kunnen verder zetten. Uiteraard moet de facility vrij zijn vooraleer deze functie te gebruiken.
- `void Release(number_type _transaction_nbr)`: Maakt de facility afhandig van de transactie waaraan ze toegekend is. De facility moet effectief toegekend zijn om deze functie te kunnen gebruiken. Het identificatienummer van de transactie waaraan de facility is toegekend moet gelijk zijn aan `_transaction_nbr`. De transacties die ergens in het model wachten op de beëindiging van het gebruik van de facility worden ervan op de hoogte gebracht dat ze hun weg kunnen verder zetten.
- `void Preempt(number_type _transaction_nbr)`: De facility wordt in de verdringingstoestand gebracht, zelfs in het geval de facility niet bezet is. Als de facility niet bezet is, wordt ze benomen door de verdringende transactie. De transacties die ergens in het model wachten op het in gebruik nemen van de facility worden ervan op de hoogte gebracht dat ze hun weg kunnen verder zetten. Als de facility bezet is wordt gecontroleerd of de verdringende transactie niet dezelfde is als de verdrongen transactie. Als dit niet het geval is, wordt de facility afgenomen van de verdrongen transactie en wordt een monitor-element corresponderend met de transactie vooraan in de verdringingsketen geplaatst. In het monitor-element wordt bijgehouden hoelang de facility nog aan de verdrongen transactie zou toegekend blijven mocht er geen verdringing opgetreden zijn. Het event notice corresponderende met de verdrongen transactie wordt verwijderd uit de event list waarin het zich bevindt.
- `void Return(number_type _transaction_nbr)`: Er wordt gecontroleerd of de facility zich in de verdringingstoestand bevindt, de facility toegekend is aan een transactie en het identificatienummer van deze transactie gelijk is aan `_transaction_nbr`. De facility wordt afgenomen van de transactie. Als er zich geen verdrongen transacties op de verdringingsketen bevinden wordt de verdringingstoestand afgesloten. De transacties die ergens in het model wachten op het niet in gebruik zijn of het zich niet in de verdringingstoestand bevinden van de facility worden hiervan

op de hoogte gebracht zodat ze hun weg kunnen verder zetten. Als de verdringingsketen niet leeg is, wordt het eerste monitor-element van de keten verwijderd en wordt aan de corresponderende transactie terug de facility toegekend. De periode dat deze transactie de facilty zal gebruiken wordt ingesteld op de in het monitor-element opgeslagen resterende gebruikstijd. Er wordt terug een event notice voor de transactie opgenomen in één van de event lists. Als na het terugplaatsen van de transactie de verdringingsketen leeg is, worden de transacties wachtend op het niet in de verdringingstoestand zijn van de facility ervan op de hoogste gebracht dat ze hun weg kunnen vervolgen.

- `boolean_type Full(void)`: Geeft aan of de facility toegekend is aan een transactie.
- `boolean_type Preempted(void)`
- `TransactionClass *Transaction(void)`: Levert een wijzer naar de transactie die momenteel de facility bezet.
- `count_type Total(void)`: Totaal aantal transacties die de facility gebruikt hebben sinds de vorige initialisatie van de data-collectoren.
- `count_type NonZeroTotal(void)`: Totaal aantal transacties die de facility gebruikt hebben en waarvoor de gebruikstijd verschillend is van nul.
- `value_type AvTime(void)`: Gemiddelde gebruikstijd van de facility door transacties.
- `value_type AvNonZeroTime(void)`: Gemiddelde van nul verschillende gebruikstijd.
- `value_type AvUtilisation(void)`: Fractionele waarde die de gemiddelde bezetting van de facility voorstelt.

FunctionClass

Objecten zijn functies. Naast de GPSS-manier om functies te construeren is er in HGPSS een additionele mogelijkheid om dit te doen ingevoerd. In GPSS wordt een functie gedefinieerd door een argument op te geven, een verzameling punten en een type dat aangeeft hoe de verzameling punten moet worden geïnterpreteerd. In HGPSS kan aan de functie-entiteit gewoon een wijzer naar een C++-functie geleverd worden.

Lidvariabelen

- `function_type function_kind`: Type van een functie gedefinieerd op de GPSS-manier.
- `value_type (*function)(void)`: Wijzer naar C++-functie te gebruiken bij de HGPSS-manier van functie-definitie. Deze C++-functie mag geen argumenten hebben en moet een resultaat van het type `value_type` teruggeven.
- `parameter_type argument`: Argument van een op de GPSS-manier gedefinieerde functie.
- `count_type nbr_of_points`: Aantal punten van een op de GPSS-manier gedefinieerde functie.
- `value_type *abscis`: Wijzer naar een één-dimensionale tabel van abscissen horende bij de punten van een op de GPSS-manier gedefinieerde functie.
- `void *ordinate`: Wijzer naar een één-dimensionale tabel van ordinaten horende bij de punten van een op de GPSS-manier gedefinieerde functie.

Lidfuncties

- `value_type interpolate(value_type _x1,`
  `value_type _y1,`
  `value_type _x2,`
  `value_type _y2,`
  `value_type _x):`

  Voert een lineaire interpolatie (of extrapolatie) uit in het punt met abscis `_x` tussen de punten `(_x1,_y1)` en `(_x2,_y2)`.

- `FunctionClass(number_type _nbr,value_type (*_function)(void))`: Deze constructor moet gebruikt worden bij een HGPSS functie-definitie. Het identificatienummer wordt geïnitialiseerd met `_nbr` en `function` met `_function`.

- `FunctionClass(number_type _nbr,`
  `parameter_type _argument,`
  `function_type _function_kind,`
  `count_type _nbr_of_points,`
  `value_type *_abscis,`
  `void *_ordinate):`

  Deze constructor moet gebruikt worden bij een GPSS functie-definitie. Het identificatienummer wordt geïnitialiseerd met `_nbr`, `argument` met `_argument`, `function_kind` met `_function_kind`, `nbr_of_points` met `_nbr_of_points`, `abscis` met `_abscis` en `ordinate` met `_ordinate`.

- `void Print(ofstream *_output_file)`

- `value_type Value(void)`: Levert het resultaat van de functie. Dit resultaat wordt als volgt bepaald:

  - Als `function` niet `NULL` is, wordt de C++-functie waar `function` naar verwijst aangeroepen en wordt het resultaat teruggegeven.
  - In het andere geval zijn er naargelang `function_kind` vijf mogelijkheden:
    - Als `function_kind` de waarde `FUNCTION_C` heeft, definieert de verzameling punten een *continue functie.* De punten worden verbonden door lijnstukken en er wordt lineair geïnterpoleerd tussen de twee punten waarvoor de abscissen respectievelijk kleiner of gelijk aan en groter zijn dan de abscis bepaald door `argument` (Figuur 6.12, geval 2). Als de abscis bepaald door `argmument` kleiner is dan alle abscissen van de verzameling punten, wordt het lijnstuk bepaald door het eerste en het tweede punt geëxtrapoleerd (Figuur 6.12, geval 1). Als de abscis bepaald door `argument` groter is dan alle abscissen van de verzameling punten, wordt de ordinaat genomen van het punt met de grootste abscis (Figuur 6.12, geval 3).
    - Als `function_kind` de waarde `FUNCTION_D` heeft, definieert de verzameling punten een *discrete functie.* Vanuit elk punt wordt een lijnstuk getrokken tot juist vóór het vorige punt. De drie mogelijke gevallen worden in Figuur 6.13 geïllustreerd.
    - Als `function_kind` de waarde `FUNCTION_E` heeft, definieert de verzameling punten een *attribuutfunctie.* De ordinaten die bij een dergelijke functie gebruikt worden zijn niet numeriek maar zijn van het type `parameter_type`. De manier waarop de punten gebruikt worden is identitiek als bij de discrete functies. Bij het teruggeven van het resultaat van de functie wordt de geselecteerde ordinaat geëvalueerd.

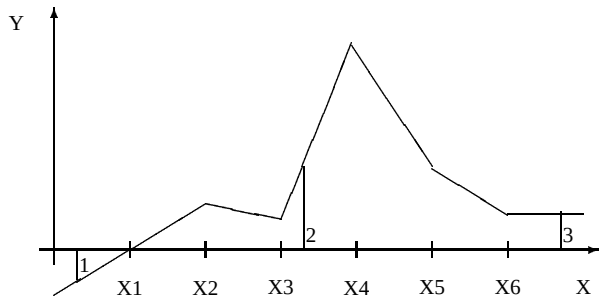

Figure 6.12: Continue GPSS-functie

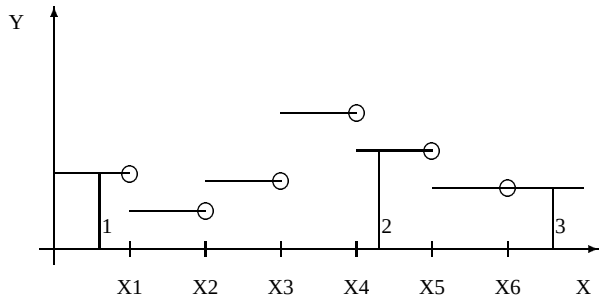

Figure 6.13: Discrete GPSS-functie

* Als `function_kind` de waarde `FUNCTION_L` heeft, definieert de verzameling punten een *discrete lijst.* De abscis gedefinieerd door `argument` moet in dit geval steeds de abscis zijn van één van de punten uit de verzameling punten (Figuur 6.14).
* Als `function_kind` de waarde `FUNCTION_M` heeft, definieert de verzameling punten een *attribuutlijst.* De ordinaten zijn niet numeriek maar van het type `parameter_type`. De manier waarop de punten gebruikt worden is dezelfde als bij de discrete lijst.

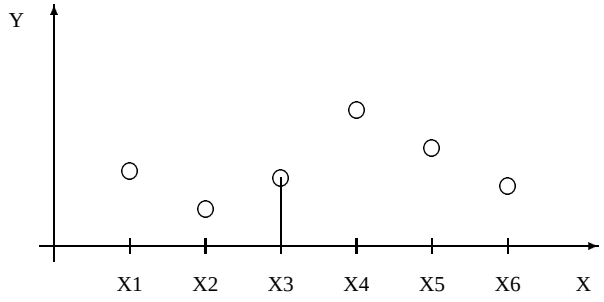

Figure 6.14: Discrete lijst GPSS-functie

InputClass

Objecten worden gebruikt om de plaatsen binnen een model te markeren waar transacties vanuit het bovenliggend model in het model worden gebracht. De objecten hebben geen eigenlijke functionaliteit, hun enige nut is om bij de uitvoer informatie te krijgen over de verschillende plaatsen waar transacties het model betreden en om te beletten dat meerdere inputs met hetzelfde identificatienummer worden gecreëerd.

Lidvariabelen

- `number_type block_nbr`: Identificatienummer van het blok waar de transacties het model betreden.

Lidfuncties

- `InputClass(number_type _nbr,number_type _block_nbr)`: Initialiseert het identificatienummer met `_nbr` en `block_nbr` met `_block_nbr`.
- `void Print(ofstream *_output_file)`
- `number_type BlockNbr(void)`

LeaveClass

Objecten worden gebruikt om de plaatsen binnen een model te markeren waar transacties vanuit een submodel terug in het model worden gebracht. De objecten hebben geen eigenlijke functionaliteit, hun enige nut is om bij de uitvoer informatie te krijgen over de verschillende plaatsen waar transacties het model terug betreden en om te beletten dat meerdere leaves met hetzelfde identificatienummer worden gecreëerd.

Lidvariabelen

- `number_type block_nbr`: Identificatienummer van het blok waar de transacties het model terug betreden.

Lidfuncties

- `LeaveClass(number_type _nbr,number_type _block_nbr)`: Initialiseert het identificatienummer met `_nbr` en `block_nbr` met `_block_nbr`.
- `void Print(ofstream *_output_file)`
- `number_type BlockNbr(void)`

LogicSwitchClass

Objecten zijn logic switches. Deze switches kunnen slechts twee toestanden aannemen.

Lidvariabelen

- `boolean_type value`: Huidige toestand van de logic switch.

Lidfuncties

- `LogicSwitchClass(number_type _nbr)`: Initialiseert identificatienummer met `_nbr` en `value` met `FALSE`.
- `logic_switch_information_type *Information(void)`
- `void Print(ofstream *_output_file)`
- `void Set(void)`: Stelt de logic switch in op `TRUE` en signaleert dit aan alle transacties die wachten totdat de logic switch de waarde `TRUE` aanneemt.
- `void Reset(void)`: Stelt de logic switch in op `FALSE` en signaleert dit aan alle transacties die wachten totdat de logic switch de waarde `FALSE` aanneemt.
- `void Invert(void)`: Complementeert de toestand van de logic switch en signaleert de huidige toestand aan alle transacties die wachten totdat de logic switch de waarde `TRUE` of `FALSE` aanneemt.
- `boolean_type Value(void)`

MatchingChain

Matching chains worden gebruikt om een aantal transacties behorende tot een bepaalde assembly set te groeperen. Matching-operaties kunnen slechts bij blokken uit een beperkt aantal klassen optreden, namelijk bij ASSEMBLE-, GATHER- en MATCH-blokken. In dergelijke blokken kunnen terzelfdertijd matching-operaties plaatsvinden die betrekking hebben op verschillende assembly sets. In elk van deze blokken kunnen bijgevolg meerdere matching chains onstaan, één per assembly set. In principe kunnen er maximaal zoveel matching chains onstaan binnen een blok als er binnen het model assembly sets voorkomen. Alle matching chains binnen een blok worden op een keten van matching chains bijgehouden, de zogenaamde matching chain chain. Een matching chain-object bestaat naast de eigenlijke keten ook uit een teller die het aantal te verzamelen transacties aangeeft.

Lidvariabelen

- `ChainClass matching_chain`: Keten van elementen behorende tot `MonitorElementClass`. Elke monitor element verwijst naar een transactie.
- `count_type assembly_count`: Aantal transacties aan dat moet verzameld worden op de matching chain. Deze teller is enkel van belang bij gebruik van de matching chain in ASSEMBLE- en GATHER-blokken.

Lidfuncties

- `MatchingChainClass(number_type _nbr)`: Initialiseert het identificatienummer met `_nbr`.
- `~MatchingChainClass(void)`: Verwijdert de volledige matching chain uit het geheugen.
- `void Print(ofstream *_output_file)`

- `void Append(MonitorElementClass *_monitor_element)`: Voegt het monitor-element aangeduid door `_monitor_element` toe aan de keten. Als dit element het eerste is dat op de keten wordt geplaatst, wordt gesignaleerd dat er een nieuwe matching chain onstaan is aan de transacties die wachten op het ontstaan van een matching chain in een blok voor de assembly set waartoe zij behoren.
- `MonitorElementClass *RemoveFirst(void)`: Verwijdert het eerste monitor element van de keten en levert een wijzer naar dit element. Als na het verwijderen van het element de keten leeg is, wordt dit gesignaleerd aan de transacties die wachten op de afwezigheid van een matching chain in een blok voor de assembly set waartoe zij behoren.
- `count_type Length(void)`: Levert het aantal elementen op de keten.
- `void SetAssemblyCount(count_type _assembly_count)`
- `count_type GetAssemblyCount(void)`

MatchingChainChainClass

Alle matching chains die kunnen onstaan in een blok worden bijgehouden op een keten. Deze keten is een keten van ketens en wordt de matching chain chain genoemd.

Lidvariabelen

- `EntityChainClass matching_chain_chain`: Keten van matching chains.

Lidfuncties

- `MatchingChainChainClass(number_type _nbr)`: Initialiseert het identificatienummer met `_nbr`.
- `~MatchingChainChainClass(void)`: Verwijdert alle matching chains uit het geheugen.
- `void Print(ofstream *_output_file)`
- `MatchingChain *MatchingChain(number_type _nbr)`

MSavevalueClass

Objecten zijn matrix savevalues, twee-dimensionale tabellen die enkelvoudige elementen bevatten.

Lidvariabelen

- `word_type word_kind`: Geeft aan of de elementen van de tabel fullword of halfword zijn. In de huidige implementatie wordt met dit verschil geen rekening gehouden. De elementen van de tabel zijn steeds van het type `value_type`.
- `number_type number_of_rows`: Aantal tabelrijen.
- `number_type number_of_columns`: Aantal tabelkolommen.
- `value_type *value`: Wijzer naar de tabel.

Lidfuncties

- `MSavevalueClass(number_type _nbr,`
  `word_type _word_kind,`
  `number_type _nbr_of_rows,`
  `number_type _nbr_of_columns):`

  Initialiseert het identificatienummer met `_nbr`, `word_kind` met `_word_kind`, `nbr_of_rows` met `_nbr_of_rows` en `nbr_of_columns` met `_nbr_of_columns`. De tabel wordt gecreëerd en alle elementen worden met `VALUE_NONE` geïnitialiseerd.

- `void Clear(void)`: Deze functie wordt aangeroepen door de processor bij het afhandelen van het CLEAR-commando. Alle elementen uit de matrix worden met `VALUE_NONE` geïnitialiseerd.

- `m_savevalue_information_type *Information(void)`

- `void Print(ofstream *_output_file)`

- `void SetValue(number_type _row_nbr,`
  `number_type _column_nbr,`
  `value_type _value)`

- `value_type GetValue(number_type _row_nbr,number_type _column_nbr)`

OutputClass

Objecten worden gebruikt om de plaatsen binnen een model te markeren waar transacties het model verlaten om terug te keren naar het bovenliggend model. De objecten hebben geen eigenlijke functionaliteit, hun enige nut is om bij de uitvoer informatie te krijgen over de verschillende plaatsen waar transacties het model verlaten en om te beletten dat meerdere outputs met hetzelfde identificatienummer worden gecreëerd.

Lidvariabelen

- `number_type block_nbr`: Identificatienummer van het blok waar de transacties het model verlaten.

Lidfuncties

- `OutputClass(number_type _nbr,number_type _block_nbr)`: Initialiseert het identificatienummer met `_nbr` en `block_nbr` met `_block_nbr`.

- `void Print(ofstream *_output_file)`

- `number_type BlockNbr(void)`

QueueClass

Objecten worden gebruikt om wachtlijnen te analyseren die onstaan in een model. Monitor-elementen corresponderend met transacties worden in een rij geplaatst en gegevens in verband met de gemiddelde lengte van de rij en de gemiddelde tijd doorgebracht op de wachtrij worden verzameld. Een object kan aan een table gekoppeld worden. In dit geval worden de respectievelijke tijden doorgebracht in de rij automatisch in een table opgenomen. Van een transactie kunnen meerdere kopieën in de rij opgenomen worden door aan de transactie een aantal eenheden toe te kennen. In de praktijk wordt in een dergelijk geval niet effectief meerdere malen een monitor-element corresponderend met de transactie in de rij opgenomen. In plaats daarvan wordt een afzonderlijke teller bijgehouden die niet slaat op het aantal monitor-elementen in de rij, maar op de lengte die de rij zou hebben rekening gehouden met het aantal eenheden.

Lidvariabelen

- `count_type length`: Lengte van de rij, rekening houdend met het aantal eenheden.
- `number_type table_nbr`: Identificatienummer van de geconjugeerde table.
- `ChainClass queue`: Keten van monitor-elementen die de wachtlijn voorstelt.
- `DataSumClass Time`: Data-collectie object voor het verzamelen van gegevens omtrent de tijd doorgebracht in de wachtlijn.
- `DataIntegralClass content`: Data-collectie object voor het verzamelen van gegevens omtrent de lengte van de wachtlijn.

Lidfuncties

- `QueueClass(number_type _nbr)`: Initialiseert het identificatienummer met `_nbr`, `length` met nul en `table_nbr` met `NUMBER_NONE`.
- `~QueueClass(void)`: Verwijdert de elementen van `queue` uit het geheugen.
- `void Reset(void)`: Deze functie wordt aangeroepen door de processor bij het afhandelen van het RESET-commando. De data-collectoren worden geïnitialiseerd.
- `queue_information_type *Information(void)`
- `void Print(ofstream *_output_file)`
- `void QTable(number_type _table_nbr)`: Initialiseert `table_nbr` met `_table_nbr`.
- `void Queue(number_type _transaction_nbr,count_type _nbr_of_units)`: Laat de transactie met identificatienummer `_transaction_nbr` toe op de wachtlijn. Daartoe wordt de huidige lengte van de wachtlijn opgenomen in de statistieken en een monitor-element gecreëerd dat achteraan `queue` wordt gevoegd. De lengte van de wachtlijn wordt vermeerderd met `_nbr_of_units`.
- `void Depart(number_type _transaction_nbr,count_type _nbr_of_units)`:
  De transactie met identificatienummer `_transaction_nbr` wordt uit de wachtlijn gehaald. Daartoe worden de huidige lengte van de wachtlijn en de tijd doorgebracht in de wachtlijn door de

transactie opgenomen in de statistieken en het monitor-element corresponderende met de transactie uit `queue` gehaald en uit het geheugen verwijderd. De lengte van de wachtlijn wordt verminderd met `_nbr_of_units`. Als er een geconjugeerde table is, met andere woorden als `table_nbr` niet gelijk is aan `NUMBER_NONE`, wordt de tijd doorgebracht in de wachtlijn door de transactie ook opgenomen in de table.

- `count_type Total(void)`: Totaal aantal transacties die de wachtlijn tot nog toe bezochten.
- `count_type NonZeroTotal(void)`: Totaal aantal transacties die de wachtlijn tot nog toe bezochten en die een van nul verschillende tijd in de wachtlijn doorbrachten.
- `count_type Content(void)`: Huidige lengte van de wachtlijn.
- `value_type AvContent(void)`: Gemiddelde lengte van de wachtlijn.
- `value_type AvNonZeroContent(void)`: Gemiddelde lengte van de wachtlijn, enkel rekening houdend met de transacties die een van nul verschillende tijd in de wachtlijn doorbrachten.
- `count_type MaximumContent(void)`: Maximale lengte van de wachtlijn tot nog toe.
- `value_type AvTime(void)`: Gemiddelde tijd doorgebracht in de wachtlijn.
- `value_type AvNonZeroTime(void)`: Gemiddelde van nul verschillende tijd doorgebracht in de wachtlijn.

RandomNbrGeneratorClass

De meest populaire methode voor het genereren van pseudo-random numbers is de *congruentiële methode* [Neelamkavil 1987]. Deze methode is gebaseerd op het mathematische concept van congruentie en residu's in de getaltheorie en werd geïntroduceerd door D.H. Lehmer is 1951. De methode is gebaseerd op de formule

$$X_{n+1} = (CX_n + C_0) moduloM$$

waarin C_0, C en M niet-negatieve constanten zijn en X_n initieel gelijk aan X_0 gekozen wordt. X_0 wordt de *seed* genoemd en moet op één of andere manier worden gespecifieerd. Methodes gebaseerd op de voorgaande formule worden *lineaire congruentiële methodes* genoemd om ze te onderscheiden van de *multiplicatieve congruentiële methodes.* Deze methodes maken gebruik van de formule

$$X_{n+1} = (CX_n) moduloM$$

De random numbers generators binnen de HGPSS++-kernel maken gebruik van deze laatste methode. Voor de constanten C en M werden dezelfde waarden gekozen als diegenen die binnen de IBM 32-bit machines gebruikt worden of werden, namelijk $C = 630360016$ en $M = 2147483647$.

Random number generators kunnen binnen HGPSS++ op twee wijzen gebruikt worden: ofwel wordt een random number generator gecreëerd als een entiteit met een identificatienummer, ofwel wordt een random number generator gebruikt als lidobject binnen een blokklasse. In het laatste geval is er geen sprake van een identificatienummer. In geen van de twee gevallen moet een seed opgegeven worden, deze wordt afgeleid uit het *instantienummer* van de random number generator. Naargelang random number generators gecreëerd worden, worden deze consecutief genummerd met een instantienummer. De processor houdt het aantal reeds gecreëerde generators bij.

Lidvariabelen

- `number_type instance_nbr`: Instantienummer van de random number generator.
- `double X`: Wordt initieel gelijk gesteld aan de seed en wordt daarna telkens berekend uit de formule.
- `double C`: Constante C.
- `double M`: Constante M.
- `parameter_type mean`: Gemiddelde waarde van de te genereren getallen.
- `parameter_type spread`: Spreiding op de te genereren getallen.

Lidfuncties

- `RandomNbrGeneratorClass(void)`: Het vanuit de processor geleverd instantienummer wordt toegekend aan `instance_nbr`. Een uit het instantienummer afgeleide seed wordt toegekend aan `X`. `C` en `M` worden op de gepaste waarde ingesteld.
- `RandomNbrGeneratorClass(number_type _nbr)`: Het identificatienummer wordt met `_nbr` geïnitialiseerd. `instance_nbr`, `X`, `C` en `M` worden op de gepaste waarde ingesteld. Aan `mean` en `spread` wordt `P(PARAMETER_VALUE,0.5)` toegekend.
- `void Print(ofstream *)`
- `void Initialise(parameter_type _mean,parameter_type _spread)`: `spread` en `mean` worden op `_mean` en `_spread` ingesteld.
- `void Reset(void)`: De originele seed wordt terug aan `X` toegekend.
- `value_type Sample(void)`: Levert een sample van de random number generator.

SavevalueClass

Objecten zijn savevalues en kunnen een enkelvoudige waarde bevatten.

Lidvariabelen

- `value_type value`: Waarde van de savevalue.

Lidfuncties

- `SavevalueClass(number_type _nbr)`: Initialiseert het identificatienummer met `_nbr` en `value` met `VALUE_NONE`.
- `savevalue_information_type *Information(void)`
- `void Print(ofstream *_output_file)`
- `void SetValue(value_type _value)`
- `value_type GetValue(void)`

StorageClass

Objecten bezitten een gelimiteerd aantal eenheden die kunnen toegekend worden aan transacties. Er worden statistieken bijgehouden in verband met de tijd gedurende dewelke eenheden werden toegekend en de verhouding van de toegekende capaciteit ten opzichte van de totale capaciteit. Net als bij queues kan een bepaalde transactie meerdere eenheden van een storage innemen door bij de aanvraag tot toekenning naast de transactie ook een aantal eenheden te vermelden. De transacties waaraan eenheden werden toegekend, worden op een keten bijgehouden. Transacties die meerdere eenheden in bruikleen hebben, worden slechts éénmaal in de keten geplaatst. Een teller houdt de effectieve bezetting van de storage bij rekening houdend met de respectievelijke aantallen eenheden.

Lidvariabelen

- `count_type capacity`: Capaciteit van de storage, dit is het maximaal aantal eenheden dat kan toegekend worden.
- `count_type length`: Huidige bezetting van de storage, rekening houdend met het aantal eenheden.
- `ChainClass storage`: Keten waarop voor alle transacties die een deel van de storage in bruikleen hebben, een monitor-element geplaatst wordt.
- `DataSumClass time`: Data-collectie object voor het verzamelen van gegevens omtrent de tijd gedurende dewelke eenheden werden toegekend aan transacties.
- `DataIntegralClass content`: Data-collectie object voor het verzamelen van gegevens omtrent de bezetting van de storage.

Lidfuncties

- `StorageClass(number_type _nbr,count_type _capacity)`: Initialiseert het identificatienummer met `_nbr`, `capacity` met `_capacity` en `length` met nul.
- `~StorageClass(void)`: Verwijdert alle monitor-elementen die zich op de keten bevinden uit het geheugen.
- `void Clear(void)`: Deze functie wordt door de processor aangroepen bij de afhandeling van het CLEAR-commando. Alle monitor-elementen worden uit het geheugen verwijderd, `length` wordt met nul geïnitialiseerd en de data-collectoren worden geïnitialiseerd.
- `void Reset(void)`: Deze functie wordt door de processor aangeroepen bij de afhandeling van het RESET-commando. De data-collectoren worden geïnitialiseerd.
- `storage_information_type *Information(void)`
- `void Print(ofstream *_output_file)`
- `void Enter(number_type _transaction_nbr,count_type _nbr_of_units)`:
  Als het gevraagde aantal eenheden beschikbaar is, wordt de huidige bezetting in de statistieken opgenomen, `length` met `_nbr_of_units` vermeerderd en wordt een monitor-element corresponderende met de transactie aangeduid door `_transaction_nbr` in de keten `storage` opgenomen.

Als hiermee de totale capaciteit bereikt wordt, wordt dit aan de transacties die hierop wachten gemeld. Als door het toelaten van de transactie, de storage niet meer leeg is, wordt dit eveneens gemeld aan de geïnteresseerde transacties.

- `void Leave(number_type _transaction_nbr,count_type _nbr_of_units)`:
  Als het aantal bezette eenheden van de storage minimaal `_nbr_of_units` is en de transactie met identificatienummer `_transaction_nbr` een aantal eenheden in bruikleen heeft, wordt de huidige bezetting evenals de tijd gedurende dewelke de eenheden werden gebruikt, opgenomen in de statistieken. `length` wordt met `_nbr_of_units` verminderd en het monitor-element corresponderende met de transactie wordt uit de keten gehaald en uit het geheugen verwijderd. Als na het vertrek van de transactie, de storage niet meer tot haar volle capaciteit bezet is, wordt dit gemeld aan de belanghebbende transacties. Dit gebeurt ook als de storage onbezet is na het vertrek van de transactie.
- `count_type Capacity(void)`
- `count_type RemainingCapacity(void)`: Verschil van de capaciteit en de huidige bezetting.
- `boolean_type Full(void)`: Geeft aan of de storage tot haar volle capaciteit bezet is.
- `boolean_type Empty(void)`: Geeft aan of de storage onbezet is.
- `count_type Total(void)`: Totaal aantal transacties die eenheden van de storage gebruikt hebben.
- `count_type NonZeroTotal(void)`: Totaal aantal transacties die eenheden van de storage gedurende een van nul verschillende tijd gebruikt hebben.
- `count_type Content(void)`: Huidige bezetting van de storage, met andere woorden met het aantal uitgeleende eenheden.
- `value_type AvContent(void)`: Gemiddelde bezetting van de storage.
- `value_type AvNonZeroContent(void)`: Gemiddelde bezetting van de storage enkel rekening houdend met de van nul verschillende gebruikstijden.
- `count_type MaximumContent(void)`: Maximale bezetting van de storage.
- `value_type AvTime(void)`: Gemiddelde gebruikstijd van een eenheid.
- `value_type AvNonZeroTime(void)`: Gemiddelde gebruikstijd van een eenheid enkel rekening houdend met de van nul verschillende tijden.
- `value_type AvUtilisation(void)`: Fractionele gemiddelde bezettingsgraad van de storage.

SubModelClass

Een object van `SubModelClass` mag niet verward worden met een object van `ModelClass`. Instanties van `ModelClass` modelleren de modellen die een onderdeel zijn van de het volledige model van een systeem. Elk submodel kan op haar beurt een aantal submodellen bezitten. Deze submodellen zijn uiteraard ook instanties van `ModelClass`. Een model bezit echter geen rechtstreekse wijzers naar al haar submodellen. In plaats daarvan bezit een model een aantal instanties van `SubModelClass`. Deze objecten bestaan naast een wijzer naar het eigenlijke submodel ook uit een keten van instanties van `EnterClass`

en een keten van instanties van `LeaveClass`. Deze objecten houden verband met de plaatsen in het model waar transacties het model verlaten en het submodel in kwestie betreden en de plaatsen waar transacties vanuit het submodel terug in het model komen.

Lidvariabelen

- `EnterChainClass enter_chain`: Keten van enters.
- `LeaveChainClass leave_chain`: Keten van leaves.
- `ModelClass *model`: Wijzer naar eigenlijke submodel.

Lidfuncties

- `SubModelClass(number_type _nbr)`: Initialiseert het identificatienummer met `_nbr` en `model` met `NULL`.
- `~SubModelClass(void)`: Verwijdert beide ketens en het submodel uit het geheugen.
- `void Print(ofstream *_output_file)`
- `void Register(EnterClass *_enter)`: Neemt de enter aangeduid door `_enter` op in de keten.
- `void Register(LeaveClass *_leave)`: Neemt de leave aangeduid door `_leave` op in de keten.
- `void Register(ModelClass *_model)`: Initialiseert `model` met `_model`.
- `EnterClass *Enter(number_type _nbr)`
- `LeaveClass *Leave(number_type _nbr)`
- `ModelClass *model(void)`

TableClass

Objecten worden gebruikt om een aantal statistieken bij te houden in verband met ingelezen waarden. De belangrijkste statistiek is een frequentietabel. De ingelezen waarden worden in een aantal klassen verdeeld en per klasse wordt de frequentie van voorkomen in de tabel bijgehouden. Het aantal klassen en de breedte van de klassen kan worden ingesteld. Alle klassen zijn even breed en beslaan aansluitende intervals. Links van de eerste klasse en rechts van de laatste klasse worden twee additionele klassen voorzien om de waarden op te vangen die anders uit de boot vallen.

Lidvariabelen

- `parameter_type arguments`: De waarden waarover statistieken moeten bijgehouden worden.
- `value_type start`: Onderste limiet van de eerste klasse.
- `value_type width`: Breedte van de klassen.
- `count_type count`: Aantal klassen inclusief de twee klassen die gebruikt worden om de waarden op te vangen die buiten de intervals vallen behorende tot de eigenlijke klassen.

- `weighted_option_type weighted_option`: Geeft aan of het gaat om een gewogen table. Als deze optie af wordt gezet, wordt geen rekening gehouden met de gewichten opgegeven bij de in de table op te nemen waarden.
- `time_type time_interval`: Tijdsinterval gebruikt bij RT-mode tables.
- `count_type *table`: Wijzer naar de frequentietabel.
- `time_type previous_time`: Stand van de absolute klok bij het opnemen van de vorige waarde in de table. Deze variabele wordt enkel gebruikt bij RT- en IA-mode tables.
- `count_type nbr_of_arrivals`: Het aantal referenties naar de table binnen het huidige tijdlot bij RT-mode tables.
- `DataSumClass information`: Data-collectie object voor het verzamelen van gegevens omtrent de som van de waarden opgenomen in de table.

Lidfuncties

- `void update(value_type _value,value_type _weighting_factor)`:
  `_value` wordt opgenomen in het data-collectie object. De klasse waarin de waarde valt wordt geselecteerd en de frequentie wordt, indien de table als gewogen table gedeclareerd is, met `_weighting_factor` vermeerderd. Als de table niet gewogen is, wordt de frequentie wars van `_weighting_factor` met één vermeerderd.
- ```
  TableClass(number_type          _nbr,
             parameter_type       _arguments,
             value_type           _start,
             value_type           _width,
             count_type           _count,
             weighted_option_type _weighted_option,
             time_type            _time_interval):
  ```
  Initialiseert het identificatienummer met `_nbr`, `arguments` met `_arguments`, `start` met `_start`, `width` met `_width`, `count` met `_count`, `weighted_option` met `_weighted_option` en `time_interval` met `_time_interval`. De frequentietabel wordt gecreëerd rekening houdend met het aantal gewenste klassen en alle frequenties worden met nul geïnitialiseerd. `previous_time` en `nbr_of_arrivals` worden met nul geïnitialiseerd.
- `~TableClass(void)`: De frequentietabel wordt uit het geheugen verwijderd.
- `void Clear(void)`: Deze functie wordt door de processor aangeroepen bij de afhandeling van het CLEAR-commando. Alle frequenties en `nbr_of_arrivals` worden met nul geïnitialiseerd en `previous_time` met de huidige waarde van de absolute klok. Het data-collectie object wordt geïnitialiseerd.
- `table_information_type *Information(void)`
- `void Print(ofstream *_output_file)`
- `void UpdateTable(value_type _weighting_factor)`: De werking van deze functie hangt af van de aard van de table: IA-mode, RT-mode of gewone table. Het al dan niet gewogen zijn van de table speelt hier geen rol.
  ```

  - Als het om een IA- of *interarrival*-mode table gaat (het veld `ParameterKind` van `arguments` moet dan gelijk zijn aan `PARAMETER_IA`), wordt de functie `update` aangeroepen met als argumenten het verschil van de huidige waarde van de absolute klok en `previous_time`, en `_weighting_factor`. `previous_time` wordt daarna gelijk gesteld aan de waarde van de absolute klok. In de table worden dus steeds de tijden tussen de opeenvolgende aanroepen van de functie `Update` opgenomen.
  - Als het om een RT- or *rate*-mode table gaat (het veld `ParameterKind` van `arguments` moet dan gelijk zijn aan `PARAMETER_RT`), wordt nagegaan of het verschil tussen de huidige waarde van de absolute klok en `previous_time` groter is dan `time_interval`. Als dit niet zo is, wordt `nbr_of_arrivals` met één vermeerderd. In het andere geval wordt de functie `update` aangeroepen met als argumenten `nbr_of_arrivals` en `_weighting_factor`. Daarna wordt `previous_time` ingesteld op de huidige waarde van de absolute klok en `nbr_of_arrivals` op één. In de table wordt dus steeds het aantal aanroepen van de functie `Update` opgenomen binnen het tijdslot bepaald door `time_interval`.
  - Als het om een gewone table gaat tenslotte, wordt de functie `update` gewoon aangeroepen met als argumenten de waarde resulterend uit de evaluatie van `arguments`, en `_weighting_factor`.
  - `void UpdateQTable(value_type _value)`: Deze functie wordt door een queue aangeroepen als de table als geconjugeerd met de queue werd gedeclareerd door een QTABLE-declaratie. De functie `update` wordt aangeroepen met als argumenten `_value` en één.

- `value_type Start(void)`
- `value_type Width(void)`
- `count_type Count(void)`
- `count_type Total(void)`: Aantal door de table verwerkte waarden.
- `count_type NonZeroTotal(void)`: Aantal van nul verschillende door de table verwerkte waarden.
- `count_type Table(number_type _class_nbr)`: Frequentie horende bij de klasse aangeduid door `_class_nbr`. Deze statistiek is de enige waarbij rekening gehouden wordt met het gewicht, tenminste als de table als gewogen table gedeclareerd is.
- `value_type Minimum(void)`: Kleinste door de table verwerkte waarde.
- `value_type Maximum(void)`: Grootste door de table verwerkte waarde.
- `value_type Average(void)`: Gemiddelde van de door de table verwerkte waarden.
- `value_type NonZeroAverage(void)`: Gemiddelde van de van nul verschillende door de table verwerkte waarden.
- `value_type Deviation(void)`: Standaard afwijking van de door de table verwerkte waarden.
- `value_type NonZeroDeviation(void)`: Standaard afwijking van de van nul verschillende door de table verwerkte waarden.

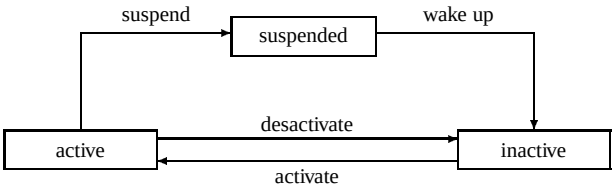

Figure 6.15: Mogelijke toestanden en toestandstransities van een transactie

TransactionClass

Transacties zijn zonder twijfel de belangrijkste entiteiten binnen een model. Ze zijn de enige entiteiten die zich doorheen het model voortbewegen. In tegenstelling tot de conceptuele mobiliteit van de transacties zullen de objecten die de implementatie van de transactie-entiteiten zijn, zich niet verplaatsen doorheen de implementatie van het model. Transactie- objecten zitten vastgeankerd in de transactieketen maar bezitten wel een aantal attributen die aanduiden op welke plaats in het model de transactie-entiteit zich bevindt. In tegenstelling tot de andere entiteitsketen, bestaan er geen lokale transactieketens. Er is slechts één transactieketen, bijgehouden door de processor. Een gevolg hiervan is dat een transactie een attribuut heeft dat aanduidt in welk submodel de transactie zich bevindt. Transacties kunnen zich in drie toestanden bevinden. De toestanden en de mogelijke toestandstransities worden in Figuur 6.15 aangeduid.

- Een transactie bevindt zich in de *actieve toestand* als ze de onverdeelde aandacht van de processor krijgt om doorheen het model voortbewogen te worden. Slechts één enkele transactie kan zich op een bepaald moment in de actieve toestand bevinden. Deze transactie wordt de *actieve transactie* genoemd.
- Een transactie bevindt zich in *slapende toestand* als de transactie omwille van een bepaalde blokkerende conditie haar weg niet kan voortzetten. Transacties in de slapende toestand zullen door de processor ongemoeid worden gelaten totdat de blokkerende conditie verdwijnt. Enkel een transactie die zich in de actieve toestand bevindt kan in de slapende toestand belanden.
- De *inactieve toestand* tenslotte is die toestand waarin transacties de aandacht van de processor niet hebben maar er wel om vragen. Als de processor de actieve transactie niet verder kan bewegen, zal hij zijn aandacht verleggen naar één van de transacties in de inactieve toestand.

Lidvariabelen

- `number_type assembly_set_nbr`: Nummer van de assembly set waartoe de transactie behoort.
- `ModelClass *model`: Wijzer naar het model waarin de transactie zich bevindt.
- `EventClass *event`: Wijzer naar het event notice dat betrekking heeft op de transactie.
- `boolean_type generated`: Geeft aan of de transactie gegenereerd is. Een transactie kan zich al op de transactieketen bevinden zonder dat ze gegenereerd is. Dergelijke transacties worden *virtuele transacties* genoemd omdat ze eigenlijk nog niet in het model aanwezig zijn.
- `number_type block_nbr`: Identificatienummer van het blok waarin de transactie zich bevindt.

- `number_type next_block_nbr`: Identificatienummer van het blok waarnaar de transactie zich zal bewegen bij het verlaten van het blok waarin ze zich bevindt.
- `state_type state`: Toestand waarin de transactie zich bevindt.
- `wait_for_type wait_for`: Als de transactie zich in slapende toestand bevindt, betekent dit dat ze wacht totdat een blokkerende conditie verdwijnt. `wait_for` duidt de conditie aan die moet gelden vooraleer de transactie haar weg kan verder zetten.
- `number_type wait_for_number`: Over het algemeen staat de conditie die moet gelden vooraleer een slapende transactie haar weg kan verder zetten, in verband met een bepaalde entiteit. Het identificatienummer van deze entiteit wordt gegeven door `wait_for_number`.
- `count_type priority`: Prioriteit van de transactie.
- `count_type nbr_of_parameters`: Aantal parameters die de transactie bezit.
- `value_type *parameter`: Wijzer naar de tabel met parameters.
- `time_type mark_time`: Deze variabele bevat initieel de absolute tijd waarop de transactie gecreëerd werd. Deze tijd kan echter vervangen worden door de huidige waarde van de absolute klok door gebruik te maken van een MARK-blok.

Lidfuncties

- `TransactionClass(void)`: Het identificatienummer wordt geïnitialiseerd met een door de processor geleverd nummer. Het is noodzakelijk dat de processor een nieuw en van de bestaande nummers verschillend identificatienummer genereerd aangezien transacties automatisch worden gecreëerd en ze nooit expliciet via hun identificatienummer behandeld worden door de modelbouwer. Het assembly set-nummer wordt geïnitaliseerd met het identificatienummer, `model` met het actieve model, `event` met `NULL`, `generated` met `FALSE`, `block_nbr` en `next_block_nbr` met `NUMBER_NONE`, `state` met `STATE_INACTIVE`, `wait_for` met `WAIT_FOR_NONE`, `wait_for_nbr` met `NUMBER_NONE`, `priority` en `nbr_of_parameters` met nul, `parameter` met `NULL` en `mark_time` met de waarde van de absolute klok. Tenslotte wordt de transactie automatisch opgenomen in de transactieketen.
- `~TransactionClass(void)`: De tabel met parameters wordt uit het geheugen verwijderd en de transactie wordt automatisch uit de transactieketen verwijderd.
- `void Print(ofstream *_output_file)`
- `void SetAssemblySetNbr(number_type _assembly_set_nbr)`
- `number_type GetAssemblySetNbr(void)`
- `void SetModel(ModelClass *_model)`
- `ModelClass *GetModel(void)`
- `void SetEvent(EventClass *_event)`
- `EventClass *GetEvent(void)`

- `void SetGenerated(void)`
- `boolean_type GetGenerated(void)`
- `void SetBlockNbr(number_type _block_nbr)`
- `number_type GetBlockNbr(void)`
- `void SetNextBlockNbr(number_type _next_block_nbr)`
- `number_type GetNextBlockNbr(void)`
- `state_type State(void)`
- `wait_for_type WaitFor(void)`
- `number_type WaitForNbr(void)`
- `void Suspend(wait_for_type _wait_for,number_type _wait_for_nbr)`: Initialiseert `state` met `STATE_SUSPENDED`, `wait_for` met `_wait_for` en `wait_for_nbr` met `_wait_for_nbr`.
- `void WakeUp(void)`: Initialiseert `state` met `STATE_INACTIVE`, `wait_for` met `WAIT_FOR_NONE` en `wait_for_nbr` met `NUMBER_NONE`.
- `void Activate(void)`: Initialiseert `state` met `STATE_ACTIVE`.
- `void Desactivate(void)`: Initialiseert `state` met `STATE_INACTIVE`.
- `void SetPriority(count_type _priority)`
- `count_type GetPriority(void)`
- `void CreateParameters(count_type _nbr_of_parameters)`: Maakt een tabel aan bestaande uit een aantal parameters gelijk aan `_nbr_of_parameters`. Er kan slechts éénmaal een tabel gecreëerd worden. Alle elementen van de tabel worden met `VALUE_NONE` geïnitialiseerd. `nbr_of_parameters` wordt geïnitialiseerd met `_nbr_of_parameters`.
- `count_type NbrOfParameters(void)`
- `void SetParameter(number_type _nbr,value_type _value)`
- `value_type GetParameter(number_type _nbr)`
- `void SetMarkTime(time_type _time)`
- `time_type GetMarkTime(void)`

UserChainClass

User chains en queues bezitten een aantal gemeenschappelijke karakteristieken. User chains zijn ketens waarop vertegenwoordigers van transacties kunnen worden geplaatst. Elke user chain bezit een *link indicator*. Dit is een vlag die door blokken die van de user chain gebruik maken kan worden aangewend.

Lidvariabelen

- `boolean_type link_indicator`: Link indicator.
- `ChainClass user_chain`: Keten van monitor-elementen refererend naar transacties.
- `DataSumClass time`: Data-collectie object gebruikt voor het verzamelen van gegevens omtrent de tijd doorgebracht door transacties op de keten.
- `DataIntegralClass content`: Data-collectie object gebruikt voor het verzamelen van gegevens omtrent de lengte van de keten.

Lidfuncties

- `UserChainClass(number_type _nbr)`: Initialiseert het identificatienummer met `_nbr` en `link_indicator` met `FALSE`.
- `~UserChainClass(void)`: Verwijdert alle monitor-elementen op de keten uit het geheugen.
- `void Reset(void)`: Deze functie wordt door de processor aangeroepen bij de afhandeling van een RESET-commando. `link_indicator` wordt met `FALSE` geïnitialiseerd en de data-collectie objecten worden eveneens geïnitialiseerd.
- `user_chain_information_type *Information(void)`
- `void Print(ofstream *_output_file)`
- `void Link(number_type _transaction_nbr,link_type _link_kind)`: De huidige lengte van de keten wordt opgeslagen in `content`. Naargelang de inhoud van het veld `LinkKind` van `_link_kind` wordt een monitor-element correspondende met de transactie met identificatienummer `_transaction_nbr` op een bepaalde plaats opgenomen in de keten.
  - Als de inhoud van het veld `LINK_LIFO` is, wordt het monitor-element vooraan in de keten opgenomen.
  - Als de inhoud van het veld `LINK_FIFO` is, wordt het monitor-element achteraan opgenomen.
  - Als de inhoud van het veld `LINK_P` is, wordt de locatie waar het monitor-element wordt geplaatst zodanig gekozen dat de keten gesorteerd is volgens de inhoud van de transactie-parameter met het nummer aangegeven door het veld `ParameterNbr` van `_link_kind`.
- `number_type Unlink(parameter_type _parameter_nbr,`
  `parameter_type _match_argument):`

  Naargelang de inhoud van de parameters `_parameter_nbr` en `_match_argument` wordt een bepaald monitor-element uit de keten geselecteerd en uit de keten verwijderd. Als het veld `ParameterKind` van `_parameter_nbr` gelijk is aan `PARAMETER_BACK`, wordt het laatste monitor-element geselecteerd. Als zowel `_parameter_nbr` als `_match_argument` gelijk zijn aan `P()` wordt het eerste monitor-element geselecteerd. In alle andere gevallen wordt het eerste monitor-element geselecteerd waarvoor de inhoud van de parameter van de transactie aangeduid door het element met het nummer `_parameter_nbr` gelijk is aan `_match_argument`. Als er geen monitor-element gevonden wordt dat aan de voorwaarde voldoet of als de keten leeg is, wordt `NUMBER_NONE` als resultaat teruggegeven door de functie. In het andere geval is dit het identificatienummer van de

transactie aangeduid door het geselecteerde monitor-element. Vóór de verwijdering van het geselecteerde monitor-element uit de keten en het geheugen wordt de huidige lengte van de keten eerst opgenomen in `content`. De tijd doorgebracht op de keten door de corresponderende transactie wordt opgenomen in `time`.

- `void SetLinkIndicator(void)`
- `void ResetLinkIndicator(void)`
- `boolean_type LinkIndicatorSet(void)`
- `count_type Total(void)`: Totale aantal in de keten opgenomen transacties sinds de laatste initialisatie.
- `count_type NonZeroTotal(void)`: Totale aantal in de keten opgenomen transacties die een van nul verschillende tijd in de keten doorbrachten, sinds de laatste initialisatie.
- `count_type Content(void)`: Aantal momenteel op de keten residerende transacties.
- `value_type AvContent(void)`: Gemiddelde aantal op de keten residerende transacties.
- `value_type AvNonZeroContent(void)`: Gemiddelde aantal op de keten residerende transacties enkel rekening houdend met de transacties die een van nul verschillende tijd op de keten doorbrachten.
- `count_type MaximumContent(void)`: Maximale lengte van de keten.
- `value_type AvTime(void)`: Gemiddelde tijd doorgebracht op de keten door transacties.
- `value_type AvNonZeroTime(void)`: Gemiddelde tijd doorgebracht op de keten door transacties enkel rekening houdend met de van nul verschillende tijden.

VariableClass

Objecten zijn arithmetische variabelen.

Lidvariabelen

- `value_type (*variable)(void)`: Wijzer naar een door de modelbouwer geconstrueerde functie. Deze functie zal worden uitgevoerd als de variabele wordt gebruikt in een model. De functie mag geen parameters hebben en moet een resultaat van het type `value_type` teruggeven.

Lidfuncties

- `VariableClass(number_type _nbr,value_type (*_variable)(void))`: Initialiseert het identificatienummer met `_nbr` en `variable` met `_variable`.
- `void Print(ofstream *_output_file)`
- `value_type Value(void)`: Voert de functie aangeduid door `variable` uit en geeft het resultaat terug.

6.2.4 Entiteitsketenklassen

Entiteitsketens worden gebruikt om alle entiteiten behorende tot een bepaalde klasse te groeperen. Alle entiteitsketenklassen zijn afgeleid van `EntityChainClass`. Buiten dit gemeenschappelijk kenmerk zijn er nog een aantal overeenkomsten tussen de entiteitsketenklassen:

- Alle klassen bezitten een destructor die ervoor zal zorgen dat alle entiteiten waaruit de keten bestaat uit het geheugen worden verwijderd.
- Alle klassen hebben een functie die toegang verleent tot de entiteit op de keten met een bepaald identificatienummer. De functie levert een wijzer naar de entiteit.
- Indien de entiteiten op de keten bepaalde acties moeten uitvoeren als reactie op een CLEAR- of RESET-commando, heeft de klasse een functie die door de processor zal aangeroepen worden bij de afhandeling van een dergelijk commando. Deze functies zullen niets anders doen dan voor elke entiteit op de keten de `Clear`- en `Reset`-functies aanroepen. Het CLEAR- of RESET-commando wordt met andere woorden via de entiteitsketen gedistribueerd naar de entiteiten toe. Als de entiteiten geen `Clear`- of `Reset`-functie bezitten is het effect van de `Clear`- of `Reset`-functie bij een entiteitsketen de verwijdering uit het geheugen van alle elementen van de keten. In Tabel 6.2 wordt een overzicht gegeven van de reactie van de entiteitsketens op een CLEAR- of RESET-commando.
- Sterk analoog is de `Print`-functie die elke klasse bezit. Informatie over alle entiteiten op de keten zal worden afgedrukt door de functie `Print` voor elke entiteit aan te roepen.

De entiteitsketenklassen zijn:

- `BlockChainClass`
- `BooleanVariableChainClass`
- `EnterChainClass`
- `ExternChainClass`
- `FacilityChainClass`
- `FunctionChainClass`
- `InputChainClass`
- `LeaveChainClass`
- `LogicSwitchChainClass`
- `MatchingChainChainChainClass`
- `MSavevalueChainClass`
- `OutputChainClass`
- `QueueChainClass`
- `RandomNbrGeneratorChainClass`

ENTITY CHAIN	Clear	Reset
Block chain	Clear	Reset
Boolean variable chain	-	-
Enter chain	-	-
Extern chain	-	-
Facility chain	Delete	Reset
Function chain	-	-
Input chain	-	-
Leave chain	-	-
Logic switch chain	Delete	-
Matching chain chain chain	Delete	-
Matrix savevalue chain	Clear	-
Output chain	-	-
Queue chain	Delete	Reset
Random number generator chain	Delete	-
Savevalue chain	Delete	-
Storage chain	Clear	Reset
Submodel chain	-	-
Table chain	Clear	Reset
Transaction chain	Delete	-
User chain chain	Delete	Reset
Variable chain	-	-

Table 6.2: Reactie van entiteitsketens op CLEAR- of RESET-commando

- `SavevalueChainClass`
- `StorageChainClass`
- `SubModelChainClass`
- `TableChainClass`
- `TransactionChainClass`
- `UserChainChainClass`
- `VariableChainClass`

BlockChainClass

- `~BlockChainClass(void)`
- `void Clear(void)`
- `void Reset(void)`
- `Print(ofstream *_output_file)`
- `BlockClass *Block(number_type _nbr)`

BooleanVariableClass

- `~BooleanVariableChainClass(void)`
- `Print(ofstream *_output_file)`
- `BooleanVariableClass *BooleanVariable(number_type _nbr)`

EnterChainClass

- `~EnterChainClass(void)`
- `Print(ofstream *_output_file)`
- `EnterClass *Enter(number_type _nbr)`

ExternChainClass

- `~ExternChainClass(void)`
- `Print(ofstream *_output_file)`
- `ExternClass *Extern(number_type _nbr)`
- `void Scan(void)`: De keten wordt afgelopen en voor elk element wordt de functie `Body` aangeroepen.

FacilityChainClass

- `~FacilityChainClass(void)`
- `void Clear(void)`
- `void Reset(void)`
- `Print(ofstream *_output_file)`
- `FacilityClass *Facility(number_type _nbr)`

FunctionChainClass

- `~FunctionChainClass(void)`
- `Print(ofstream *_output_file)`
- `FunctionClass *Function(number_type _nbr)`

InputChainChainClass

- `~InputChainClass(void)`
- `Print(ofstream *_output_file)`
- `InputClass *Input(number_type _nbr)`

LeaveChainClass

- `~LeaveChainClass(void)`
- `Print(ofstream *_output_file)`
- `LeaveClass *Leave(number_type _nbr)`

LogicSwitchChainClass

- `~LogicSwitchChainClass(void)`
- `void Clear(void)`
- `Print(ofstream *_output_file)`
- `LogicSwitchClass *LogicSwitch(number_type _nbr)`

MatchingChainChainChainClass

- `~MatchingChainChainChainClass(void)`
- `void Clear(void)`
- `Print(ofstream *_output_file)`
- `MatchingChainChainClass *MatchingChainChain(number_type _nbr)`

MSavevalueChainClass

- `~MSavevalueChainClass(void)`
- `void Clear(void)`
- `Print(ofstream *_output_file)`
- `MSavevalueClass *MSavevalue(number_type _nbr)`

OutputChainClass

- `~OutputChainClass(void)`
- `Print(ofstream *_output_file)`
- `OutputClass *Output(number_type _nbr)`

QueueChainClass

- `~QueueChainClass(void)`
- `void Clear(void)`
- `void Reset(void)`
- `Print(ofstream *_output_file)`
- `QueueClass *Queue(number_type _nbr)`

RandomNbrGeneratorChainClass

- ~RandomNbrGeneratorChainClass(void)
- void Clear(void)
- Print(ofstream *_output_file)
- RandomNbrGeneratorClass *RandomNbrGenerator(number_type _nbr)

SavevalueChainClass

- ~SavevalueChainClass(void)
- void Clear(void)
- Print(ofstream *_output_file)
- SavevalueClass *Savevalue(number_type _nbr)

StorageChainClass

- ~StorageChainClass(void)
- void Clear(void)
- void Reset(void)
- Print(ofstream *_output_file)
- StorageClass *Storage(number_type _nbr)

SubModelChainClass

- ~SubModelChainClass(void)
- Print(ofstream *_output_file)
- SubModelClass *SubModel(number_type _nbr)

TableChainClass

- ~TableChainClass(void)
- void Clear(void)
- void Reset(void)
- Print(ofstream *_output_file)
- TableClass *Table(number_type _nbr)

TransactionChainClass

- `~TransactionChainClass(void)`
- `void Clear(void)`
- `Print(ofstream *_output_file)`
- `void DeRegister(TransactionClass *_transaction)`: Verwijdert de transactie aangeduid door `_transaction` uit de keten.
- `TransactionClass *Transaction(number_type _nbr)`
- `void Signal(wait_for_type _wait_for,number_type _wait_for_nbr)`:
  De keten wordt afgelopen en alle transacties die wachten op een toestand aangeduid door `_wait_for` voor een entiteit met identificatienummer `_wait_for_nbr`, worden vanuit de slapende naar de inactieve toestand gebracht.

UserChainChainClass

- `~UserChainChainClass(void)`
- `void Clear(void)`
- `void Reset(void)`
- `Print(ofstream *_output_file)`
- `UserChainClass *UserChain(number_type _nbr)`

VariableChainClass

- `~VariableChainClass(void)`
- `Print(ofstream *_output_file)`
- `VariableClass *Variable(number_type _nbr)`

6.2.5 Processorklasse

De processor (Figuur 6.16) is het kloppend hart van de HGPSS++-kernel. Het hele simulatie-proces wordt gedirigeerd vanuit de processor. De relatie tussen entiteiten en de processor is als die van slaven tegenover de meester. De processor is het enige statische object binnen de kernel, alle andere objecten worden onder impuls van de processor gecreëerd naargelang de noden.

Lidvariabelen

- `ofstream *output_file`: Wijzer naar het bestand waarin de uitvoer belandt, het zogenaamde *uitvoerbestand.*
- `AbsoluteClockClass absolute_clock`: Absolute klok.
- `RelativeClockClass relative_clock`: Relatieve klok.

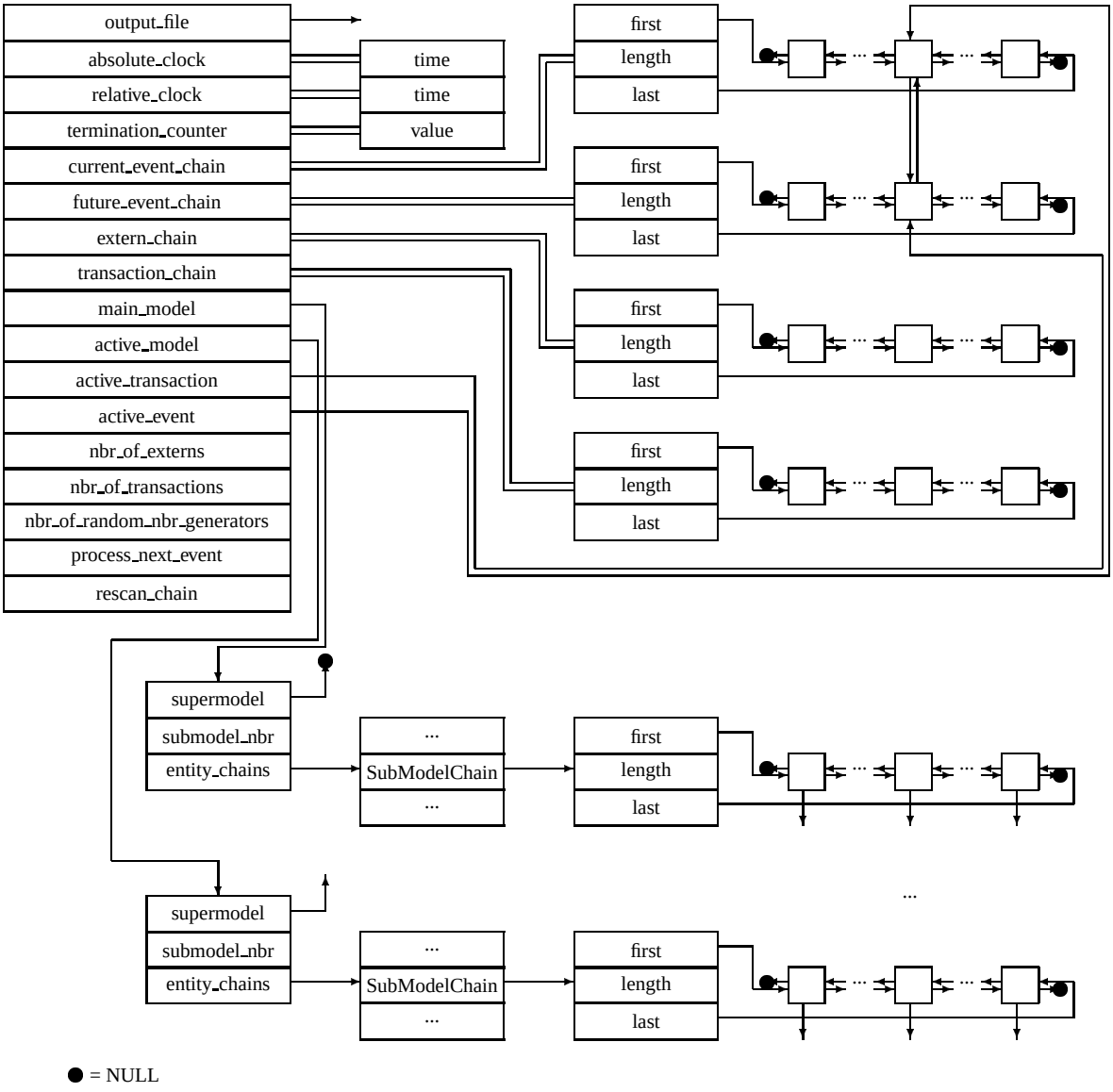

Figure 6.16: ProcessorClass

- `TerminationCounterClass termination_counter`: Beëindigingsteller.
- `CurrentEventChainClass current_event_chain`: Current event chain.
- `FutureEventChainClass future_event_chain`: Future event chain.
- `ExternChain extern_chain`: Keten bestaande uit instanties van `ExternClass`. Deze entiteiten bevatten wijzers naar op elk actief simulatie-tijdspunt aan te roepen functies.
- `TransactionChainClass transaction_chain`: Keten van alle transacties die zich in de verschillende submodellen bevinden deel uitmakend van het totale model.
- `ModelClass *main_model`: Wijzer naar het model dat zich volledig bovenaan de model-hiërarchie bevindt.
- `ModelClass *active_model`: Wijzer naar het actieve model. Dit is het model dat de actieve transactie bevat.
- `TransactionClass *active_transaction`: Wijzer naar de actieve transactie.
- `EventClass *active_event`: Wijzer naar het event notice dat hoort bij de actieve transactie.
- `number_type nbr_of_externs`: Laatste aan een extern-entiteit toegekende identificatienummer.
- `number_type nbr_of_transactions`: Laatste aan een transactie toegekende identificatienummer.
- `number_type nbr_of_random_nbr_generators`: Laatste aan een random number generator toegekende identificatienummer.
- `boolean_type process_next_event`: Interne vlag die aangeeft dat de processor de actieve transactie moet buiten beschouwing laten en overgaan tot de verwerking van het volgende event.
- `boolean_type rescan_chain`: Interne vlag die aangeeft dat na de afhandeling van de actieve transactie niet moet overgegaan worden naar het volgende event, maar tot een rescan van de current event chain.

Lidfuncties

Constructor

- `ProcessorClass(void)`: Initialiseert `output_file`, `main_model`, `active_model`, `active_transaction` en `active_event` met `NULL`, `nbr_of_externs`, `nbr_of_transactions` en `nbr_of_random_nbr_generators` met nul, `process_next_event` met `FALSE` en `rescan_chain` met `TRUE`.

Functies voor intern gebruik

- `void open_output_file(char _output_file_name[])`:
  Opent indien mogelijk een tekstbestand met als naam `_output_file_name`, drukt een hoofding af in dit bestand en initialiseert `output_file` met een wijzer naar het bestand.
- `void close_output_file(void)`: Sluit indien mogelijk het bestand aangeduid door `output_file` en wijst `NULL` toe aan `output_file`.

- `void message(char _message[])`: Drukt een bericht aangegeven door `_message` af in het uitvoerbestand.
- `void schedule_initial_events(void)`: De volledige model-boom wordt afgelopen en in elk model worden de GENERATE-blokken opgezocht. Voor elke van deze blokken wordt de functie `ScheduleInitialEvent` aangeroepen. Deze functie zal een initieel event schedulen voor het blok. Bij het aanroepen van `schedule_initial_events` moet het actieve model het hoofdmodel zijn.
- `void scan_extern_chain(void)`: Roept de functie `extern_chain.Scan` aan.
- `void update_current_event_chain(void)`: Als er zich events op de future event list bevinden wordt `rescan_chain` op `TRUE` geplaatst en worden de absolute en de relatieve klok verplaatst zodat de simulatie-tijd aangeduid door het eerste event op de future event chain wordt gereflecteerd. Daarna worden alle events op de future event chain die op de deze tijd gescheduled zijn, overgebracht naar de current event chain.
- `void scan_current_event_chain(void)`: Deze functie is zonder twijfel één van de belangrijkste van de hele HGPSS++-kernel. Globaal gezien wordt de current event chain herhaaldelijk afgelopen waarbij de events die kunnen afgehandeld worden ook effectief worden verwerkt. Het proces loopt ten einde als `rescan_chain` de waarde `FALSE` aanneemt of als de beëindigingsteller is afgelopen. Gezien het belang van de functie wordt ze in Figuur 6.17 in haar totaliteit weergegeven.

Functies voor de registratie van entiteiten Deze functies instrueren de processor een bepaalde entiteit op te nemen in, of in één geval te verwijderen uit, de geschikte entiteitsketen. Om entiteiten op te nemen in of te verwijderen uit één van de rechtstreeks door de processor beheerde ketens als de externketen en de transactie-keten is het gebruik van deze functies noodzakelijk. In de andere gevallen kan ook rechtstreeks op de entiteitsketens ingewerkt worden door eerst de entititeitsketen zelf te selecteren en daarna de entiteit te registreren. Deze werkwijze is echter omslachtiger. Entiteiten worden steeds geregistreerd in de entiteitsketens van het actieve model. De functies voor de registratie van entiteiten zijn:

- `void Register(BlockClass *_block)`
- `void Register(BooleanVariableClass *_boolean_variable)`
- `void Register(ExternClass *_extern)`
- `void Register(FunctionClass *_function)`
- `void Register(InputClass *_input)`
- `void Register(MSavevalueClass *_m_savevalue)`
- `void Register(OutputClass *_output)`
- `void Register(StorageClass *_storage)`
- `void Register(SubModelClass *_submodel)`
- `void Register(TableClass *_table)`

```
void ProcessorClass::scan_current_event_chain(void)
   {
     EventClass *event;
     EventClass *next_event;
     boolean_type available;
     wait_for_type wait_for;
     number_type wait_for_nbr;

     do
        {
          rescan_chain=FALSE;
          event=(EventClass *)current_event_chain.First();
          while ( (event!=NULL) &&
                  (!rescan_chain) &&
                  (!termination_counter.Terminated()) )
             {
               active_event=event;
               next_event=(EventClass *)active_event->Next;
               if (active_event->Transaction()->State()!=STATE_SUSPENDED)
                  {
                    active_transaction=event->Transaction();
                    active_model=active_transaction->GetModel();
                    active_transaction->Activate();
                    do
                       {
                         process_next_event=FALSE;
                         Block(active_transaction->GetNextBlockNbr())->
                            Check(available,wait_for,wait_for_nbr);
                         if (available)
                            {
                              if (active_transaction->GetBlockNbr()!=NUMBER_NONE)
                                 Block(active_transaction->GetBlockNbr())->Departure();
                              Block(active_transaction->GetNextBlockNbr())->Arrival();
                              Block(active_transaction->GetBlockNbr())->Body();
                            }
                         else
                            if (wait_for!=WAIT_FOR_NONE)
                               SuspendTransaction(wait_for,wait_for_nbr);
                            else
                               DesactivateTransaction();
                       }
                    while ( (!process_next_event) && (!termination_counter.Terminated()) );
                  }
               active_event=NULL;
               active_transaction=NULL;
               active_model=NULL;
               event=next_event;
             }
        }
     while ( (rescan_chain) && (!termination_counter.Terminated()) );
   }
```

Figure 6.17: De interne processorfunctie “scan_current_event_chain”

- `void Register(TransactionClass *_transaction)`
- `void Register(VariableClass *_variable)`
- `void DeRegister(TransactionClass *_transaction)`

Functies die toegang tot entiteiten verlenen Voor deze functies gelden dezelfde regels als bij de functies die toelaten om entiteiten te registreren. Om toegang tot een entiteit te verkrijgen moet het identificatienummer van de entiteit worden opgegeven. De respectievelijke toegang-verlenende functies zijn:

- `BlockClass *Block(number_type _nbr)`
- `BooleanVariableClass *BooleanVariable(number_type _nbr)`
- `ExternClass *Extern(number_type _nbr)`
- `FacilityClass *Facility(number_type _nbr)`
- `FunctionClass *Function(number_type _nbr)`
- `InputClass *Input(number_type _nbr)`
- `LogicSwitchClass *LogicSwitch(number_type _nbr)`
- `MatchingChainChainClass *MatchingChainChain(number_type _nbr)`
- `MSavevalueClass *MSavevalue(number_type _nbr)`
- `OutputClass *Output(number_type _nbr)`
- `QueueClass *Queue(number_type _nbr)`
- `RandomNbrGeneratorClass *RandomNbrGenerator(number_type _nbr)`
- `SavevalueClass *Savevalue(number_type _nbr)`
- `StorageClass *Storage(number_type _nbr)`
- `SubModelClass *SubModel(number_type _nbr)`
- `TableClass *Table(number_type _nbr)`
- `TransactionClass *Transaction(number_type _nbr)`
- `UserChainClass *UserChain(number_type _nbr)`
- `VariableClass *Variable(number_type _nbr)`

Functies die toegang tot lidvariabelen verlenen Een aantal functies verlenen toegang tot de lidvariabelen van de processor of zijn er sterk mee gerelateerd:

- `ofstream *OutputFile(void)`: Levert een wijzer naar het uitvoerbestand.
- `time_type AbsoluteTime(void)`: Levert de absolute tijd.
- `void UpdateTerminationCounter(value_type _termination_counter_decrement)`: Vermindert de beëindigingsteller met `_termination_count_decrement`.
- `void SetMainModel(ModelClass *_model)`: Maakt van het model aangeduid door `_model` het hoofdmodel.
- `ModelClass *GetMainModel(void)`: Levert een wijzer naar het hoofdmodel.
- `void SetModel(ModelClass *_model)`: Maakt het model aangeduid door `_model` het actieve model.
- `ModelClass *GetModel(void)`: Levert een wijzer naar het actieve model.
- `TransactionClass *Transaction(void)`: Levert een wijzer naar de actieve transactie.
- `number_type NewExternNbr(void)`: Levert een identificatienummer voor een nieuwe extern-entiteit. Dit is het nummer van de laatst gecreëerde extern-entiteit vermeerderd met één.
- `number_type NewTransactionNbr(void)`: Levert een identificatienummer voor een nieuwe transactie. Dit is het nummer van de laatst gecreëerde transactie vermeerderd met één.
- `number_type NewRandomNbrGeneratorNbr(void)`: Levert een identificatienummer voor een nieuwe random number generator. Dit is het nummer van de laatst gecreëerde random number generator vermeerderd met één.

Functies voor het uitvoeren van acties

- `void Error(int _error_nbr,char _file[],int _line)`: Als er een bestand geopend is als uitvoerbestand, wordt in dit bestand een foutmelding afgedrukt. Als er geen uitvoerbestand is, wordt de foutmelding naar het standaard uitvoerkanaal bij het optreden van fouten gevoerd. De aard van de foutmelding is afhankelijk van het foutnummer, weergegeven door `_error_nbr`. De plaats waar de fout geconstateerd is, weergegeven door `_file` en `_line`, wordt ook afgedrukt.
- `value_type Evaluate(parameter_type _parameter)`: Een parameter van het type `parameter_type`, namelijk `_parameter` wordt geëvalueerd tot een resultaat van het type `value_type`.
- `Signal(wait_for_type _wait_for,number_type _wait_for_nbr)`:
  Roept de functie `transaction_chain.Signal` aan met als argumenten `_wait_for` en `_wait_for_nbr`.
- `DesactivateTransaction(void)`: Plaatst de actieve transactie in de inactieve toestand en `process_next_event` op `TRUE`.
- `SuspendTransaction(wait_for_type _wait_for,number_type _wait_for_nbr)`:
  Plaatst de actieve transactie in de slapende toestand en `process_next_event` op `FALSE`.

- `ScheduleEvent(EventClass *_event)`: Plaatst het event notice aangeduid door `_event` in de current of future event chain naargelang de simulatie-tijd waarop het event moet gescheduled worden. Als het event notice in de current event chain moet worden opgenomen, wordt `rescan_chain` op `TRUE` geplaatst.

- `RescheduleEvent(void)`: Verplaatst het actieve event in de current event chain om de locatie van het event binnen de keten in overeenstemming te brengen met een veranderde prioriteitswaarde van de geconjugeerde transactie.

- `RemoveEvent(EventClass *_event)`: Haalt het event aangeduid door `_event` uit de event chain waarin het zich bevindt.

- `DeleteEvent(void)`: Haalt het actieve event uit de current event chain en verwijdert het uit het geheugen.

- `RescanChain(void)`: Plaatst `rescan_chain` op `TRUE`.

Functies in rechtstreeks verband met HGPSS-commando's Elk van deze functies staat in rechtstreeks verband met één van de HGPSS-commando's. De uitvoering van een functie wordt in het uitvoerbestand gemeld.

- `void Clear(void)`: Initialiseert de absolute klok evenals de beëindigingsteller. Alle elementen op de current event chain, future event chain en de transactie-keten worden uit het geheugen verwijderd. Het hoofdmodel wordt opnieuw het actieve model. `active_transaction` en `active_event` worden met `NULL` en `nbr_of_transactions` en `nbr_of_random_nbr_generators` met nul geïnitialiseerd. `process_next_event` wordt op `FALSE` en `rescan_chain` op `TRUE` geplaatst. Het CLEAR-commando wordt gedistribueerd naar de blok-, facility-, logic switch-, matching chain chain-, matrix savevalue-, queue-, random number generator-, savevalue-, storage-, table- en user chain-keten. Er worden opnieuw initiële events gescheduled voor alle GENERATE-blokken.

- `void Down(number_type _model_nbr)`: Het submodel van het actieve model met identificatienummer `_model_nbr` wordt het actieve model.

- `End(void)`: Het uitvoerbestand wordt gesloten. De absolute en relatieve klok worden teruggedraaid en de beëindigingsteller met `VALUE_NONE` geïnitialiseerd. De volledige current event chain, de future event chain en de transactie-keten worden uit het geheugen verwijderd. De hele modelboom en alle bijhorende entiteitsketens worden uit het geheugen verwijderd. `main_model`, `active_model`, `active_transaction` en `active_event` worden met `NULL` geïnitialiseerd. `nbr_of_externs`, `nbr_of_transactions` en `nbr_of_random_nbr_generators` worden met nul geïnitialiseerd. `process_next_event` wordt op `FALSE` geplaatst en `rescan_chain` op `TRUE`.

- ```
  void *Information(information_type _information_kind,
                    number_type      _entity_nbr):
  ```

  Naargelang de inhoud van `_information_kind` wordt de entiteit aangeduid door `_entity_nbr` op één van de entiteitsketens opgezocht. Als de entiteit niet aanwezig geeft de functie `NULL` terug. In het andere geval wordt de functie `Information` van de entiteit aangeroepen en wordt de teruggegeven informatie-structuur als resultaat afgegeven.

- `void Job(void)`: Werkt in de huidige implementatie op dezelfde wijze als `End`.
  ```

- `void Print(parameter_type _lower_limit,`
  `parameter_type _upper_limit,`
  `print_type _print_kind,`
  `ofstream *_output_file):`

  Informatie over alle bestaande entiteiten behorende tot de klasse aangeduid door `print_kind` met identificatienummers gaande van `_lower_limit` tot `_upper_limit`, wordt afgedrukt. Als `_lower_limit` én `_upper_limit` gelijk zijn aan `P()`, wordt informatie over alle bestaande entiteiten behorende tot de gewenste klasse afgedrukt. Als enkel `_lower_limit` gelijk is aan `P()` wordt hiervoor `P(PARAMETER_VALUE,1)` aangenomen. Als enkel `_upper_limit` gelijk is aan `P()` wordt hiervoor de waarde van `_lower_limit` aangenomen.

  Als `_print_kind` gelijk is aan `PRINT_ABSOLUTECLOCK`, `PRINT_RELATIVECLOCK`, `PRINT_TERMINATIONCOUNTER`, `PRINT_CURRENTEVENTCHAIN` of `PRINT_FUTUREEVENTCHAIN`, gaat het uiteraard niet om informatie over entiteiten. In deze gevallen moeten `_lower_limit` en `_upper_limit` gelijk zijn aan `P()`.

  Als `_output_file` gelijk is aan `NULL` komt alle uitvoer terecht in het uitvoerbestand. In het andere geval wordt de uitvoer naar het bestand aangeduid door `_output_file` gezonden.

- `void Reset(void)`: De relatieve klok wordt teruggedraaid en de beëindigingsteller op `VALUE_NONE` ingesteld. Het hoofdmodel wordt als actieve model geïnstalleerd. `active_transaction` en `active_event` worden met `NULL` geïnitialiseerd. `process_next_event` wordt op `FALSE` en `rescan_chain` op `TRUE` ingesteld. Het RESET-commando wordt gedistribueerd naar de blok-, facility-, queue-, storage-, table- en user chain-keten.

- `Simulate(ModelClass *_model,char _output_file_name[])`:
  Een tekstbestand met naam `_output_file_name` wordt geopend en als uitvoerbestand geïnstalleerd. `_model` wordt als hoofdmodel geïnstalleerd. `active_transaction` en `active_event` worden met `NULL` geïnitialiseerd. `nbr_of_externs`, `nbr_of_transactions` en `nbr_of_random_nbr_generators` worden met nul geïnitialiseerd. `process_next_event` wordt op `FALSE` en `rescan_chain` op `TRUE` ingesteld. Er worden initiële events gescheduled voor alle GENERATE-blokken binnen het model.

- `void Start(count_type _termination_counter)`:
  Deze functie is naast `current_event_chain` één van de belangrijkste binnen de kernel. Het hoofdmodel wordt als actief model geïnstalleerd en de beëindigingsteller met `_termination_count` geïnitialiseerd. Daarna worden herhaaldelijk respectievelijk de functies `scan_extern_chain`, `scan_current_event_chain` en `update_current_event_chain` aangeroepen totdat er geen events meer kunnen afgehandeld worden of de beëindigingsteller afgelopen is. Uiteindelijk wordt om af te sluiten het hoofdmodel terug als actief model geïnstalleerd. De functie is in Figuur 6.18 in haar totaliteit opgenomen.

- `void Up(void)`: Het bovenliggende model wordt als actief model geïnstalleerd.

6.2.6 Blokklassen

De HGPSS-blokken worden geïmplementeerd door instanties van de blokklassen. Alle blokklassen zijn afgeleid van `BlockClass`, die op haar beurt is afgeleid van `EntityClass`. Buiten de gemeenschappelijke basisklasse bezitten de blokklassen nog een aantal additionele gemeenschappelijke kenmerken:

- Elke blokklasse bezit een aantal lidvariabelen die de parameters van het te modelleren blok voorstellen.

```
void ProcessorClass::Start(count_type _termination_count)
   {
     message("START: Simulation begins");
     active_model=main_model;
     termination_counter.Reset(_termination_count);
     while ( (rescan_chain) && (!termination_counter.Terminated()) )
        {
          scan_extern_chain();
          if (!termination_counter.Terminated())
             {
               scan_current_event_chain();
               if (!termination_counter.Terminated())
                  update_current_event_chain();
             }
        }
     active_model=main_model;
     message("       Simulation stops");
   }
```

Figure 6.18: De processorfunctie “Start”

- De klassen `AdvanceClass`, `GenerateClass` en `TransferClass` bezitten naast een aantal lidvariabelen ook een lidobject, namelijk een random number generator.
- Elke klasse bezit een constructor. De parameterlijst van elke constructor bestaat uit volgende formele parameters:
  - `number_type _nbr`: Identificatienummer van de instantie van de klasse.
  - `number_type _next_block_nbr`: Identificatienummer van het sequentiële blok ten opzichte van de instantie van de klasse.
  - Een aantal formele parameters die overeenstemmen met de parameters van het te modelleren blok. De actuele parameters doorgestuurd via deze formele parameters worden in de constructor toegekend aan de overeenstemmende lidvariabelen. Voor bepaalde parameters is een waarde bij verstek voorzien. Deze waarde zal worden toegekend aan de lidvariabele als de corresponderende actuele parameter de bij het type van de parameter horende verstek-waarde heeft.

  Naast de toekenning van parameters aan lidvariabelen wordt in de constructor ook de naam van de blokklasse ingevuld in de variabele `name`, een lid van `BlockClass`.
- In elke blokklasse wordt de functie `Check` die in de klasse `BlockClass` virtueel is, gedefinieerd. Deze functie wordt door de processor aangeroepen om na te gaan of een instantie van de klasse de actieve transactie kan ontvangen. Binnen de functie `Check` moeten de referentie-parameters `_available`, `_wait_for` en `_wait_for_nbr` op een geschikte waarde worden ingesteld.
- Ook de functie `Body` is binnen `BlockClass` virtueel en wordt in elke blokklasse gedefinieerd. Deze functie representeert de eigenlijke acties uitgevoerd door een instantie van de klasse.

De blokklassen zijn:

- `AdvanceClass`
- `AssembleClass`

- AssignClass
- BufferClass
- DepartClass
- _EnterClass
- EnterModelClass
- _ExternClass
- GateClass
- GatherClass
- GenerateClass
- _InputClass
- InternClass
- _LeaveClass
- LeaveModelClass
- LinkClass
- LogicClass
- LoopClass
- MarkClass
- MatchClass
- _MSavevalueClass
- _OutputClass
- PreemptClass
- PrintClass
- PriorityClass
- _QueueClass
- ReleaseClass
- ReturnClass
- _SavevalueClass
- SeizeClass
- SelectClass

- `SplitClass`
- `TabulateClass`
- `TerminateClass`
- `TestClass`
- `TransferClass`
- `UnlinkClass`

In volgend overzicht worden alle blokklassen besproken. Voor elke klasse worden niet telkens alle lidvariabelen en lidfuncties expliciet weergegeven. In plaats daarvan werd geopteerd voor een bespreking bestaande uit drie onderdelen. De respectievelijke onderdelen behandelen volgende items:

1. Het type en de betekenis van de parameters van het blok, evenals hun eventuele waarde bij verstek. Deze waarde wordt tussen `[` en `]` opgenomen. Als er geen waarde bij verstek voorzien is, maar het blok wel een andere functionaliteit vertoont bij het weglaten van een parameter, wordt dit aangegeven door `[]`.
2. De beschikbaarheid van het blok. De beschikbaarheid is het resultaat van de instelling van de drie referentie-parameters in de functie `Check`. In vele gevallen zullen de parameters zo ingesteld worden dat de actieve transactie haar weg gewoon kan vervolgen zonder geblokkeerd te worden. In dit geval is het blok steeds beschikbaar.
3. De acties uitgevoerd door de functie `Body`.

AdvanceClass

Parameters

- `parameter_type mean [P(PARAMETER_VALUE,0)]`: Gemiddelde waarde van de tijd gedurende dewelke de actieve transactie wordt gedesactiveerd.
- `parameter_type spread [P(PARAMETER_VALUE,0)]`: Spreiding ten opzichte van het gemiddelde van de tijd gedurende dewelke de actieve transactie wordt gedesactiveerd.

Beschikbaarheid Steeds beschikbaar.

Acties De actieve transactie wordt gedesactiveerd en het event gekoppeld aan de transactie wordt uit de current event chain gehaald en uit het geheugen verwijderd. Een nieuw event wordt gescheduled op het tijdstip aangegeven door de huidige waarde van de absolute klok vermeerderd met de waarde van een sample van een random number generator. Deze random number generator werd geïnitialiseerd met `mean` en `spread`. In geval het veld `SNAKind` van `spread` gelijk is aan `SNA_FN` wordt de waarde van de absolute klok niet vermeerderd met de waarde van een sample van de random number generator maar met het produkt van `mean` en `spread`.

AssembleClass

Parameters `assembly_count`: Aantal transacties die moeten worden gecombineerd tot één transactie en die behoren tot dezelfde assembly set als de actieve transactie.

Beschikbaarheid Steeds beschikbaar.

Acties

- Als er reeds een assembling-operatie aan de gang is binnen het blok voor de assembly set waartoe de actieve transactie behoort, wordt de actieve transactie gewoon uit het blok en uit het geheugen verwijderd. Het aan de transactie gekoppelde event wordt uit de current event chain gehaald en uit het geheugen verwijderd. De assembly counter van de matching chain corresponderende met het blok en de assembly set van de actieve transactie, wordt met één geïncrementeerd.
- Als er nog geen assembling-operatie aan de gang is, wordt de actieve transactie in de slapende toestand gebracht en op de geschikte matching chain geplaatst.
- Als na de behandeling van de transactie blijkt dat het vereiste aantal samen te voegen transacties bereikt is, wordt de op de matching chain geplaatste transactie van de keten verwijderd en terug in de inactieve toestand geplaatst. Een rescan van de current event chain wordt geïnitieerd.

AssignClass

Parameters

- `parameter_type parameter_nbr`: Nummer van de transactie-parameter waarvan de inhoud moet gewijzigd worden.
- `operation_type operation`: Geeft aan of de nieuwe waarde de inhoud van de transactie-parameter moet vervangen, erbij moet opgeteld worden of ervan moet afgetrokken worden.
- `parameter_type value`: Nieuwe waarde voor de aanpassing van de inhoud van de transactie-parameter.
- `parameter_type function_nbr` []: Identificatienummer van een functie die kan gebruikt worden als modifier.

Beschikbaarheid Steeds beschikbaar.

Acties Als de parameter `function_nbr` opgegeven is, wordt de aangeduide functie gebruikt als modifier. In het andere geval is de modifier één. De inhoud van de aangeduide transactie-parameter wordt gewijzigd door gebruik te maken van de nieuwe waarde en de uit te voeren operatie. De nieuwe waarde wordt vóór gebruik eerst vermenigvuldigd met de modifier.

BufferClass

Parameters Geen parameters.

Beschikbaarheid Steeds beschikbaar.

Acties De actieve transactie wordt gedesactiveerd en er wordt een rescan van de current event chain geïnitieerd.

DepartClass

Parameters

- `parameter_type queue_nbr`: Identificatienummer van de queue waaruit de actieve transactie moet worden verwijderd.
- `parameter_type nbr_of_units [P(PARAMETER_VALUE,1)]`:
  Het aantal eenheden dat moet gebruikt worden bij het aanpassen van de lengte van de queue.

Beschikbaarheid Steeds beschikbaar.

Acties De gewenste queue wordt geselecteerd en de transactie wordt eruit verwijderd.

_EnterClass

Parameters

- `parameter_type storage_nbr`: Identificatienummer van de storage waarvan de actieve transactie een aantal eenheden opeist.
- `parameter_type nbr_of_units [P(PARAMETER_VALUE,1)]`: Het aantal eenheden waarmee de gebruikte capaciteit van de storage moet aangepast worden.

Beschikbaarheid De actieve transactie wordt in de slapende toestand gebracht als de storage niet aan de vraag van de transactie kan voldoen. Deze situatie treedt op als de storage niet meer over het door de actieve transactie gevraagde aantal eenheden beschikt.

Acties De storage wordt geselecteerd en het aantal gewenste eenheden wordt toegekend. Een rescan van de current event chain wordt geïnitieerd.

EnterModelClass

Parameters

- `number_type model_nbr`: Identificatienummer van het submodel waarnaar de actieve transactie zich moet begeven.
- `number_type input_nbr`: Identificatienummer van de input waar de actieve transactie het submodel moet betreden.

Beschikbaarheid Steeds beschikbaar.

Acties Het aangeduide submodel wordt geïnstalleerd als actieve model. Het attribuut van de actieve transactie dat het model aangeeft waarin de transactie zich bevindt, wordt ingesteld op het submodel. De transactie-attributen die het huidige blok en het volgende blok op het pad van de transactie aangeven, worden respectievelijk ingesteld op `NUMBER_NONE` en het identificatienummer van het blok waar de transactie het submodel betreedt.

_ExternClass

Parameters `void (*body)(void)`: Wijzer naar de functie die moet aangeroepen worden als een transactie het blok betreedt.

Beschikbaarheid Steeds beschikbaar.

Acties De opgegeven functie wordt aangeroepen.

GateClass

Parameters

- `gate_type gate_kind`: Het type van de poort geeft de conditie aan onder dewelke de actieve transactie zijn weg verder mag zetten naar het sequentiële blok.
- `parameter_type entity_nbr`: Identificatienummer van de entiteit waarop de conditie betrekking heeft.
- `parameter_type branch_block_nbr` []: Identificatienummer van het alternatieve blok. Dit blok is hetgene waar de transactie naartoe wordt geleid als de conditie vals is.

Beschikbaarheid

- Als er geen alternatief blok opgegeven werd, wordt de conditie geëvalueerd gespecifieerd door het type van de poort en het identificatienummer van de te gebruiken entiteit. Als de conditie vals is, wordt de actieve transactie in de slapende toestand gebracht. Het type van de poort en het identificatienummer van de entiteit worden ingevuld als conditie waaronder de transactie terug mag overgaan naar de inactieve toestand.
- Als er wel een alternatief blok werd opgegeven, wordt de transactie niet geblokkeerd en is het blok steeds beschikbaar.

Acties

- Als er een alternatief blok werd opgegeven, wordt de conditie geëvalueerd aangegeven door het type van de poort en het identificatienummer van de entiteit. Als de conditie waar is, kan de actieve transactie haar weg naar het sequentiële blok verderzetten. In het andere geval zal de transactie haar weg verder moeten zetten via het alternatieve blok.
- Als er geen alternatief blok werd opgegeven, vinden er geen acties plaats. De actieve transactie zet haar weg gewoon verder naar het sequentiële blok.

GatherClass

Parameters `parameter_type gather_count`: Het aantal transacties behorende tot dezelfde assembly set die moeten verzameld worden om ze dan gezamelijk hun weg verder te laten zetten.

Beschikbaarheid Steeds beschikbaar.

Acties De actieve transactie wordt in de slapende toestand gebracht zonder de voorwaarde te vermelden onder dewelke ze terug moet ontwaken. Daarna wordt de transactie op de met het actieve blok en de assembly set corresponderende matching chain geplaatst. Als hiermee het aantal transacties op de matching chain gelijk wordt aan de gather count, worden alle transacties van de keten gehaald en in de inactieve toestand geplaatst. Een rescan van de current event chain wordt dan eveneens geïnitieerd.

GenerateClass

Particulariteit Instanties van deze klasse bezitten een particulariteit. In principe zijn GENERATE-blokken de enige die geen ingang hebben. Er kunnen dus geen transacties dergelijke blokken binnentreden. Om alle blokken op een zelfde wijze te kunnen behandelen werd ervoor geopteerd om instanties van `GenerateClass` ondanks deze particulariteit toch op dezelfde wijze te laten werken als andere blokken. Daartoe wordt elke transactie voorzien van een attribuut dat aangeeft of de transactie al gegenereerd is of niet. Transacties kunnen *gecreëerd* worden zonder dat ze daarom *gegenereerd* zijn. Dergelijke transacties worden *virtuele transacties* genoemd omdat ze op het niveau van het model eigenlijk nog niet bestaan.

Het procédé om transacties te genereren via een instantie van `GenerateClass` verloopt als volgt: bij de initialisatie van een model en al haar submodellen wordt voor elke instantie van de klasse, de functie `ScheduleInitialEvent` aangeroepen. Instanties van `GenerateClass` zijn de enige die een dergelijke functie bezitten. De functie scheduled een event voor de eerste transactie die door het blok moet gegenereerd worden. Dit event zal verwijzen naar een virtuele transactie. Virtuele transacties zullen op dezelfde wijze de blokken binnendringen als gewone transacties dat doen bij blokken behorende tot andere klassen dan `GenerateClass`. Pas bij de afhandeling van het event horende bij de virtuele transactie, zal de transactie binnen het blok worden gegenereerd.

Parameters

- `parameter_type mean [P(PARAMETER_VALUE,0)]`: Gemiddelde waarde van de tijd tussen de opeenvolgende generaties van een transactie.
- `parameter_type spread [P(PARAMETER_VALUE,0)]`: Spreiding ten opzichte van het gemiddelde van de tijd tussen de opeenvolgende generaties van een transactie.
- `parameter_type offset_interval []`: Tijdsinterval na hetwelke de eerste transactie wordt gegenereerd.
- `parameter_type limit_count []`: Aantal te genereren transacties.
- `parameter_type priority_level [P(PARAMETER_VALUE,0)]`:
  Aan de gegenereerde transactie toe te kennen prioriteit.
- `parameter_type nbr_of_parameters [P(PARAMETER_VALUE,0)]`:
  Aantal parameters waarmee de gegenereerde transactie moet worden uitgerust.
- `fullword_option_type fullword_option []`: Heeft in de huidige implementatie geen effect.

Beschikbaarheid Steeds beschikbaar.

Acties Het attribuut van de virtuele actieve transactie dat aangeeft of de transactie gegeneerd is, wordt geactiveerd. De transactie wordt gemerkt met de huidige waarde van de absolute klok. De prioriteit wordt ingesteld op de opgegeven waarde en het gewenste aantal parameters wordt gecreëerd. Aangezien de prioriteit van de transactie mogelijk is veranderd, wordt het actieve event gerescheduled. Een rescan van de current event chain wordt geïnitieerd. Als er een beperking op het aantal gegenereerde transacties werd opgegeven wordt gecontroleerd of deze limiet is bereikt. Indien de limiet nog niet is bereikt, wordt een volgende virtuele transactie gecreëerd en wordt een event voor deze transactie gescheduled op het tijdstip aangeduid door de huidige waarde van de absolute klok vermeerderd met een sample van een random number generator. Deze random number generator werd geïnitialiseerd met `mean` en `spread`. Als het veld `SNAKind` van `spread` gelijk is aan `SNA_FN`, wordt de absolute klok niet met een sample maar met het produkt van `mean` en `spread` vermeerderd.

InputClass

Parameters `number_type input_nbr`: Nummer van de met het blok geassocieerde input.

Beschikbaarheid Steeds beschikbaar.

Acties Geen acties.

InternClass

Parameters `void (*body)(void)`: Wijzer naar de functie die moet aangeroepen worden wanneer een transactie het blok betreedt.

Beschikbaarheid Steeds beschikbaar.

Acties De functie wordt aangeroepen.

_LeaveClass

Parameters

- `parameter_type storage_nbr`: Identificatienummer van de storage waaraan een aantal gebruikte eenheden moet teruggegeven worden.
- `parameter_type nbr_of_units [P(PARAMETER_VALUE,1)]`: Aantal aan de storage terug te geven eenheden.

Beschikbaarheid Steeds beschikbaar.

Acties De gewenste storage wordt geselecteerd en het aangeduide aantal eenheden wordt teruggegeven.

LeaveModelClass

Parameters

- `number_type model_nbr`: Identificatienummer van het submodel waaruit de actieve transactie terug in het actieve model komt.
- `number_type output_nbr`: Nummer van de output van het submodel waaruit de actieve transactie het submodel verlaat.

Beschikbaarheid Steeds beschikbaar.

Acties Geen acties.

LinkClass

Parameters

- `parameter_type user_chain_nbr`: Identificatienummer van de user chain waarop de actieve transactie moet geplaatst worden als aan bepaalde voorwaarden voldaan is.
- `link_type link_kind`: Het link type geeft de volgorde aan die moet gerespecteerd worden bij het op de user chain plaatsen van de actieve transactie.
- `parameter_type branch_block_nbr` []: Identificatienummer van het alternatieve blok. Naar dit blok zal de actieve transactie worden overgebracht als ze niet op de user chain mag worden geplaatst.

Beschikbaarheid Steeds beschikbaar.

Acties

- Als er geen alternatief blok is gespecifieerd of als de link indicator van de user chain geactiveerd is, wordt de current count met één verminderd, de actieve transactie uit het blok verwijderd, in de slapende toestand gebracht en op de user chain geplaatst.
- Indien er wel een alternatief blok is gespecifieerd of als de link indicator niet is geactiveerd, wordt de actieve transactie niet op de keten geplaatst en wordt ze verplicht haar weg voort te zetten langs het alternatieve blok.

LogicClass

Parameters

- `logic_type logic_kind`: Type van de actie die moet uitgevoerd worden op de logic switch.
- `parameter_type logic_switch_nbr`: Identificatienummer van de te behandelen logic switch.

Beschikbaarheid Steeds beschikbaar.

Acties De waarde van de logic switch wordt aangepast zoals opgelegd door het opgegeven operatietype.

LoopClass

Parameters

- `parameter_type parameter_nbr`: Nummer van de parameter van de actieve transactie waarvan de waarde met één moet gedecrementeerd worden en vergeleken met nul.
- `parameter_type branch_block_nbr`: Identificatienummer van het alternatieve blok. Naar dit blok zal de actieve transactie worden geleid als de gedecrementeerde parameter niet gelijk is aan nul.

Beschikbaarheid Steeds beschikbaar.

Acties De aangeduide parameter wordt met één gedecrementeerd en vergeleken met nul. Als de parameter verschillend van nul is, wordt de actieve transactie naar het alternatieve blok geleid. In het andere geval kan de transactie haar weg voortzetten naar het sequentiële blok.

MarkClass

Parameters `parameter_type parameter_nbr` []: Nummer van de parameter van de actieve transactie waarin de waarde van de absolute klok moet worden gekopieerd.

beschikbaarheid Steeds beschikbaar.

Acties Als er een nummer van een parameter gespecifieerd is, wordt de waarde van de absolute klok in de parameter gekopieerd. In het andere geval wordt het transactie-attribuut dat de stand van de absolute klok aangeeft waarop de actieve transactie gegenereerd is, overschreven door de actuele stand van de absolute klok.

MatchClass

Parameters `parameter_type match_block_nbr`: Identificatienummer van het blok dat in de matching-operatie moet worden gebruikt. Dit blok wordt het geconjugeerde blok genoemd.

Beschikbaarheid Steeds beschikbaar.

Acties

- Als er een matching-operatie aan de gang is in het geconjugeerde blok voor de assembly set waartoe de actieve transactie behoort (dit betekent dat de matching chain aldaar uit minstens één transactie bestaat), wordt de eerste transactie op die matching chain van de keten gehaald en in de inactieve toestand gebracht. Ook de actieve transactie wordt in de inactieve toestand gebracht en er wordt een rescan van de current event chain geïnitieerd.

- Als geen matching-operatie aan de gang is in het alternatieve blok voor de assembly set van de actieve transactie, wordt de actieve transactie in de slapende toestand gebracht zonder voorwaarden te specifiëren voor een eventueel ontwaken. De transactie wordt dan op de matching chain geplaatst die hoort bij het actieve blok en de assembly set waartoe de transactie behoort.

MSavevalueClass

Parameters

- `operation_type operation`: Geeft aan of het geselecteerde element van de matrix savevalue met een waarde moet geïncrementeerd of gedecrementeerd worden of door de waarde moet vervangen worden.
- `parameter_type row_nbr`: Nummer van de rij binnen de matrix savevalue waarop het element zich bevindt.
- `parameter_type column_nbr`: Nummer van de kolom binnen de matrix savevalue waarop het element zich bevindt.
- `parameter_type value`: Waarde die in de modificatie-operatie moet worden gebruikt.
- `halfword_option_type halfword_option []`: Heeft in de huidige implementatie geen effect.

Beschikbaarheid Steeds beschikbaar.

Acties De operatie wordt uitgevoerd op het element van de matrix savevalue.

_OutputClass

Parameters `number_type output_nbr`: Nummer van de output van het model waarmee het blok in verband staat.

Beschikbaarheid Steeds beschikbaar.

Acties Het ten opzichte van het actieve model bovenliggende model wordt als actief model geïnstalleerd. Het attribuut van de actieve transactie dat het model aangeeft waarin de transactie zich bevindt, wordt op het nieuwe actieve model ingesteld. De attributen van de actieve transactie die het huidige blok en het volgende blok op het pad van de transactie aangeven, worden respectievelijk ingesteld op `NUMBER_NONE` en het identificatienummer van het blok waar de transactie in het nieuwe actieve model treedt.

PreemptClass

Parameters

- `parameter_type facility_nbr`: Identificatienummer van de facility die moet gepreempt worden als aan bepaalde condities voldaan is.
- `priority_option_type priority_option []`: Als deze optie ingesteld wordt en de facility bezet is bij aankomst van een transactie, zal de facility enkel door de transactie worden gepreempt als de prioriteit van de preempting transactie groter is dan die van de preempted transactie.

- `parameter_type branch_block_nbr []`: Heeft in de huidige implementatie geen effect.
- `parameter_type parameter_nbr []`: Heeft in de huidige implementatie geen effect.
- `remove_option_type remove_option []`: Heeft in de huidige implementatie geen effect.

Beschikbaarheid

- Als er zich geen transactie in de facility bevindt, kan de actieve transactie zonder aan andere condities te moeten beantwoorden, het blok binnentreden. Alhoewel er geen transactie verdrongen wordt, komt de facility toch in de verdringingstoestand.
- Als er zich wel een transactie in de facility bevindt zijn er twee mogelijkheden: ofwel bevindt de facility zich reeds in de verdringingstoestand, ofwel is dit niet het geval.
  - Als de facility zich reeds in de verdringingstoestand bevindt, wordt nagegaan of de prioriteitsoptie opgegeven is. Als dit zo is wordt de prioriteit van de transactie die zich reeds in de facility bevindt vergeleken met de prioriteit van de actieve transactie. Als de prioriteit van de actieve transactie groter is, kan de actieve transactie het blok betreden. In het andere geval wordt de actieve transactie in de slapende toestand gebracht. Ze zal ontwaken als de facility niet meer bezet is. Als de prioriteitsoptie niet is opgegeven, wordt de actieve transactie in de slapende toestand gebracht en zal ze ontwaken als de facility zich niet meer in de verdringingstoestand bevindt.
  - Als de facility zich nog niet in de verdringingstoestand bevindt, wordt eveneens gecontroleerd of de prioriteitsoptie opgegeven is. Als dit zo is, wordt hetzelfde scenario gevolgd als wanneer de facility zich wel in de verdringingstoestand bevindt. In het andere geval kan de actieve transactie zonder verdere voorwaarden het blok betreden.

Acties De gewenste facility wordt geselecteerd en door de actieve transactie gepreempt. Een rescan van de current event chain wordt geïnitieerd.

PrintClass

Parameters

- `parameter_type lower_limit []`: Laagste van de identificatienummers van de entiteiten waarover informatie moet afgedrukt worden.
- `parameter_type upper_limit []`: Hoogste van de identificatienummers van de entiteiten waarover informatie moet afgedrukt worden.
- `print_type print_kind []`: Duidt de klasse aan van de entiteiten waarover informatie moet afgedrukt worden.
- `ofstream *_output_file []`: Wijzer naar het bestand waarin de informatie moet terecht komen.

Beschikbaarheid Steeds beschikbaar.

Acties De geselecteerde informatie wordt afgedrukt. Als er geen bestandswijzer werd gespecifieerd, wordt de informatie naar het door de processor bijgehouden uitvoerbestand geleid. Als er geen onderste en bovenste limietwaarden werden gespecifieerd, wordt over alle tot de geselecteerde klasse behorende entiteiten informatie afgedrukt. Als enkel de onderste limiet niet werd opgegeven, wordt hiervoor het identificatienummer één verondersteld. Als tenslotte enkel de bovenste limiet niet werd opgegeven wordt deze gelijk gesteld aan de onderste limiet.

PriorityClass

Parameters

- `parameter_type new_priority_value`: Nieuwe aan de actieve transactie toe te kennen prioriteit.
- `buffer_option_type buffer_option []`: Als deze optie wordt opgegeven, wordt niet alleen een rescan van de current event chain geïnitieerd, maar wordt die ook onmiddellijk uitgevoerd.

Beschikbarheid Steeds beschikbaar.

Acties De nieuwe prioriteitswaarde wordt toegekend aan de actieve transactie. Het event horende bij deze transactie wordt opnieuw gescheduled in de current event chain om de plaats van het event in deze keten te laten overeenstemmen met de veranderde prioriteit. Een rescan van de current event chain wordt geïnitieerd. Als de buffer-optie is opgegeven wordt deze rescan onmiddellijk uitgevoerd door de actieve transactie te desactiveren.

_QueueClass

Parameters

- `parameter_type queue_nbr`: Identificatienummer van de queue waarin de actieve transactie moet opgenomen worden.
- `parameter_type nbr_of_units [P(PARAMETER_VALUE,1)]`: Aantal eenheden waarmee de lengte van de queue moet aangepast worden.

Beschikbaarheid Steeds beschikbaar.

Acties De gewenste queue wordt geselecteerd en een vertegenwoordiger van de actieve transactie wordt op de queue geplaatst.

ReleaseClass

Parameters `parameter_type facilty_nbr`: Identificatienummer van de facility die moet vrijgegeven worden.

Beschikbaarheid Steeds beschikbaar.

Acties De gewenste facility wordt geselecteerd en vrijgegeven. Een rescan van de current event chain wordt geïnitieerd.

ReturnClass

Parameters `parameter_type facility_nbr`: Identificatienummer van de facility die moet vrijgegeven worden door de actieve transactie en die tevens een preempting transactie moet zijn.

Beschikbaarheid Steeds beschikbaar.

Acties De gewenste facility wordt selecteerd en vrijgegeven. Als er op de interrupt chain horende bij de facility transacties staan, wordt de eerste van deze transacties terug tot de facility toegelaten.

SavevalueClass

Parameters

- `operation_type operation`: Geeft aan of de inhoud van de savevalue met een waarde moet vermeerderd of verminderd worden, of door een waarde moet vervangen worden.
- `parameter_type value`: De waarde gebruikt in de operatie op de savevalue.
- `halfword_option_type halfword_option` []: Heeft in de huidige implementatie geen effect.

Beschikbaarheid Steeds beschikbaar.

Acties De gewenste operatie wordt uitgevoerd op de savevalue.

SeizeClass

Parameters `parameter_type facility_nbr`: Identificatienummer van de facility opge-ëist door de actieve transactie.

Beschikbaarheid Als de facility niet toegekend is, kan de actieve transactie het blok betreden. In het andere geval wordt de transactie in de slapende toestand gebracht en zal ze slechts ontwaken als de facility vrijgegeven wordt.

Acties De gewenste facility wordt geselecteerd en aan de actieve transactie toegekend.

SelectClass

Parameters

- `select_type select_kind`: Aard aan van de selectie-operatie die moet worden uitgevoerd.
- `parameter_type parameter_nbr`: Nummer van de parameter van de actieve transactie waarin het identificatienummer van de eerste transactie die aan de gestelde conditie voldoet, moet gekopieerd worden.
- `parameter_type lower_limit`: Laagste van de identificatienummers van de entiteiten die moeten onderzocht worden.

- `parameter_type upper_limit`: Hoogste van de identificatienummers van de entiteiten die moeten onderzocht worden.
- `parameter_type comparison_value []`: Waarde waarmee het entiteitsattribuut moet vergeleken worden.
- `SNA_type attribute_to_examine []`: Entiteitsattribuut dat moet worden onderzocht.
- `parameter_type branch_block_nbr []`: Identificatienummer van het alternatieve blok. Dit is het blok waar de actieve transactie naartoe wordt geleid als geen enkele van de onderzochte entiteiten aan het gestelde selectiecriterium voldoet.

Beschikbaarheid Steeds beschikbaar.

Acties De selectie wordt uitgevoerd rekening houdend met het type van de operatie, de te onderzoeken entiteitsklasse, het entiteitsattribuut en de onderste en bovenste limiet van de identificatienummers. Als er geen enkele entiteit gevonden wordt die aan het criterium voldoet zijn er twee mogelijkheden:

- Als er een alternatief blok is opgegeven, wordt de actieve transactie naar dit blok geleid.
- In het andere geval vervolgt de actieve transactie haar weg zonder meer naar het sequentiële blok.

SplitClass

Parameters

- `parameter_type nbr_of_offspring`: Aantal van de actieve transactie af te leiden afstammelingen.
- `parameter_type branch_block_nbr`: Identificatienummer van het blok waar de afstammelingen moeten naartoe worden geleid.
- `parameter_type serialisation_parameter []`: Nummer van een parameter van de actieve transactie en haar afstammelingen. Deze parameter moet gebruikt worden als serialisatie-parameter.
- `parameter_type nbr_of_parameters []`: Aantal parameters waarmee de afstammelingen moeten worden uitgerust.

Beschikbaarheid Steeds beschikbaar.

Acties Het gewenste aantal afstammelingen wordt gecreëerd. Alle afstammelingen zijn identiek aan de actieve transactie voor wat betreft assembly set, prioriteit, parameters en stand van de absolute klok bij generatie. Als expliciet het aantal parameters waarmee de afstammelingen moeten worden uitgerust, werd opgegeven, mag dit niet kleiner zijn dan het aantal parameters waarover de actieve transactie beschikt. De eventuele extra parameters waarover de afstammelingen beschikken worden met `VALUE_NONE` geïnitialiseerd. Voor alle afstammelingen wordt een event gescheduled en worden de total en current count van het blok met één geïncrementeerd. De afstammelingen worden dan naar het alternatieve blok geleid terwijl de actieve transactie haar weg naar het sequentiële blok vervolgt. Als echter een nummer van een parameter te gebruiken als serialisatie-parameter werd opgegeven, wordt

eerst nog de volgende actie ondernomen: de actieve transactie en haar afstammelingen worden opeenvolgend genummerd door de inhoud van de respectievelijke parameters aangeduid als serialisatie-parameter op te hogen met waarden gaande van één voor de actieve transactie, tot het aantal afstammelingen plus één voor de laatste afstammeling.

TabulateClass

Parameters

- `parameter_type table_nbr`: Identificatienummer van de table waarin informatie moet worden opgenomen.
- `parameter_type weighting_factor [P(PARAMETER_VALUE,1)]`: Gewichtsfactor te gebruiken bij het opnemen van de informatie in de table. Deze factor geeft het aantal maal aan dat de informatie moet worden opgenomen.

Beschikbaarheid Steeds beschikbaar.

Acties De gewenste table wordt geselecteerd en de informatie wordt erin opgenomen.

TerminateClass

Parameters

`parameter_type termination_counter_decrement [P(PARAMETER_VALUE,0)]:`

Waarde waarmee de beëindigingsteller moet worden gedecrementeerd.

Beschikbaarheid Steeds beschikbaar.

Acties De beëindigingsteller wordt met de opgegeven waarde gedecrementeerd. De current count van het blok wordt met één verminderd. De actieve transactie wordt gedesactiveerd en daarna uit het geheugen verwijderd. Het aan de transactie gekoppelde event wordt uit de current event chain gehaald en eveneens uit het geheugen verwijderd.

TestClass

Parameters

- `test_type test_kind`: Aard van de uit te voeren vergelijking.
- `parameter_type first_value`: Eerste van de te vergelijken waarden.
- `parameter_type second_value`: Tweede van de te vergelijken waarden.
- `parameter_type branch_block_nbr []`: Identificatienummer van het alternatieve blok. Als het resultaat van de relationele bewerking vals is, wordt de actieve transactie naar dit blok geleid.

Beschikbaarheid Als er geen alternatief blok werd gespecifieerd, wordt de relationele bewerking uitgevoerd. Als het resultaat van de bewerking vals is, wordt de actieve transactie in de inactieve toestand gebracht. In alle andere gevallen wordt de actieve transactie tot het blok toegelaten.

Acties Als er een alternatief blok werd opgegeven, wordt de relationele bewerking uitgevoerd. Als het resultaat van de bewerking vals is, wordt de actieve transactie naar het alternatieve blok geleid. In de andere gevallen vervolgt de actieve transactie haar weg naar het sequentiële blok.

TransferClass

Parameters

- `parameter_type selection_mode []`: Deze parameter kan op een aantal manieren geïnterpreteerd worden. In het algemeen komt het erop neer dat deze parameter zal uitmaken welk van de twee mogelijke paden verder zal gevolgd worden door de actieve transactie.
- `parameter_type first_block_nbr []`: Identificatienummer van het eerste blok op het eerste mogelijke pad.
- `parameter_type second_block_nbr []`: Identificatienummer van het eerste blok op het tweede mogelijke pad.

Beschikbaarheid Steeds beschikbaar.

Acties

- Als er geen parameter opgegeven werd om de wijze van selectie aan te geven, wordt de actieve transactie onvoorwaardelijk naar het eerste blok op het eerste pad geleid.
- Als er wel een parameter opgegeven werd om de wijze van selecteren aan te geven zijn er twee mogelijkheden.
  - Als deze parameter gelijk is aan `P(PARAMETER_BOTH)`, wordt nagegaan of het eerste blok op het eerste pad in staat is om de actieve transactie te ontvangen. Deze test wordt op dezelfde wijze uitgevoerd als door de processor. Als het blok vrij is, wordt de actieve transactie er naartoe geleid. Als het blok niet vrij is, wordt nagegaan of het eerste blok op het tweede pad vrij is. Als dit het geval is, wordt de actieve transactie er naartoe geleid. Als het blok niet vrij is of als geen tweede pad werd gespecifieerd, wordt de actieve transactie gedesactiveerd, worden haar attributen die het identificatienummer van het huidige en het volgende blok op het pad aangeven respectievelijk ingesteld op `NUMBER_NONE` en het identificatienummer van het actieve blok en wordt de current count van het actieve blok met één verminderd.
  - Als er een parameter die de wijze van selecteren aangeeft opgeven werd, maar niet gelijk is aan `P(PARAMETER_BOTH)`, wordt deze geëvalueerd en vergeleken met een sample van een random number generator. Deze random number generator levert waarden tussen nul en één, uniform verdeeld. Als de sample groter of gelijk aan de geëvalueerde parameter is, wordt de actieve transactie naar het eerste blok van het eerste pad geleid. In het andere geval wordt ze naar het eerste blok van het tweede pad geleid.

UnlinkClass

Parameters

- `parameter_type user_chain_nbr`: Identificatienummer van de user chain waarvan de transacties moeten gehaald worden.

- `parameter_type unlinked_branch_block_nbr`: Identificatienummer van het blok waar de van de user chain gehaalde transacties naartoe moeten geleid worden.
- `parameter_type unlink_count`: Het aantal van de user chain te halen transacties.
- `parameter_type parameter_nbr []`: Deze parameter wordt doorgespeeld aan de user chain en geeft aan welke transactie van de keten moet gehaald worden.
- `parameter_type match_argument []`: Deze parameter wordt samen met de vorige doorgespeeld aan de user chain en geeft aan welke transactie van de keten moet worden gehaald.
- `parameter_type branch_block_nbr []`: Identificatienummer van het alternatieve blok. Dit is het blok waar de actieve transactie naartoe moet worden geleid als er geen enkele transactie van de user chain kon worden gehaald.

Beschikbaarheid Steeds beschikbaar.

Acties Als de user chain leeg is, wordt de link indicator gedesactiveerd. Als `unlink_count` gelijk is aan `P(PARAMETER_ALL)`, worden alle transacties van de user chain gehaald die voldoen aan de voorwaarde opgelegd door `parameter_nbr` en `match_argument`. In het andere geval worden zoveel aan de voorwaarde beantwoordende transacties van de user chain gehaald als `unlink_count` aangeeft. De operatie wordt gestopt als geen enkele transactie meer aan de voorwaarde voldoet zonder dat `unlink_count` bereikt is. De in beide gevallen van de keten gehaalde transacties worden in de inactieve toestand gebracht en naar het blok aangeduid door `unlinked_branch_block_nbr` geleid. Als geen enkele transactie van de keten kon gehaald worden, wordt de actieve transactie naar het alternatieve blok overgebracht.

6.3 Gebruikersfuncties

De gebruikersfuncties zijn die functies die over het algemeen in rechtstreeks verband staan met de HGPSS-statements. De namen van deze functies zijn - waar mogelijk - dezelfde als de HGPSS statementsleutelwoorden en bestaan ook uit niets dan hoofdletters. Er kan onderscheid gemaakt worden tussen vier groepen gebruikersfuncties.

- De *blokdeclaratie-functies* staan in voor de dynamische creatie van blokken behorende tot een bepaalde klasse en de registratie van die blokken in de blok-keten van het actief model. Deze functies staan in rechtstreeks verband met de HGPSS blokdeclaratie-statements. De parameterlijst van de blokdeclaratie-functies reflecteert de parameters die kunnen gebruikt worden bij de HGPSS blokdeclaratie-statements. In vele gevallen kunnen parameters weggelaten worden. De blokdeclaratie-functies zorgen ervoor dat toch een parameter doorgegeven wordt ter vervanging van de weggelaten parameter. Deze vervangingsparameter is niet noodzakelijk dezelfde als de parameter die bij verstek zal gebruikt worden binnen het blok. De vervangingsparameter laat het blok enkel weten dat een bepaalde parameter niet werd opgegeven. Elk blok zal hier dan de noodzakelijke conclusies uit trekken.
- De *entiteitsdeclaratie-functies* staan in voor de dynamische creatie en registratie van de entiteiten die verplicht moeten gedeclareerd worden. De functies staan in rechtsteeks verband met de HGPSS entiteitsdeclaratie-statements. Ook hier kunnen soms parameters weggelaten worden.

- De *commando-functies* staan in verband met de HGPSS commando-statements. In elke functies wordt de geschikte processorfunctie aangeroepen. In bepaalde gevallen kunnen ook parameters weggelaten worden.

- Een laatste groep zijn de *additionele functies.* Zij staan niet in verband met een HGPSS-statement maar zijn wel noodzakelijk bij het opbouwen van een model in HGPSS++. Alle additionele functies worden gebruikt om informatie op te vragen van entiteiten binnen het model.

Alle gebruikersfuncties worden hieronder opgesomd. Gezien hun sterke relatie met blokken of processorfuncties is het onnodig om verdere informatie over elke gebruikersfunctie afzonderlijk te verschaffen.

6.3.1 Blokdeclaratie-functies

- ```
  void ADVANCE(number_type    _block_nbr,
               number_type    _next_block_nbr,
               parameter_type _mean=P(),
               parameter_type _spread=P());
  ```

- ```
  void ASSEMBLE(number_type    _block_nbr,
                number_type    _next_block_nbr,
                parameter_type _assembly_count);
  ```

- ```
  void ASSIGN(number_type    _block_nbr,
              number_type    _next_block_nbr,
              parameter_type _parameter_nbr,
              operation_type _operation,
              parameter_type _value,
              parameter_type _function_nbr=P());
  ```

- ```
  void BUFFER(number_type    _block_nbr,
              number_type    _next_block_nbr);
  ```

- ```
  void DEPART(number_type    _block_nbr,
              number_type    _next_block_nbr,
              parameter_type _queue_nbr,
              parameter_type _nbr_of_units=P());
  ```

- ```
  void ENTER(number_type    _block_nbr,
             number_type    _next_block_nbr,
             parameter_type _storage_nbr,
             parameter_type _nbr_of_units=P());
  ```

- ```
  void ENTERMODEL(number_type    _block_nbr,
                  number_type    _next_block_nbr,
                  number_type    _model_nbr,
                  number_type    _input_nbr);
  ```

- ```
  void EXTERN(number_type _block_nbr,
              number_type _next_block_nbr,
              void      (*_body_arrival)(void),
              void      (*_body_poll)(void));
  ```

- ```
  void GATE(number_type    _block_nbr,
            number_type    _next_block_nbr,
            gate_type      _gate_kind,
            parameter_type _entity_nbr,
            parameter_type _branch_block_nbr=P());
  ```

- ```
  void GATHER(number_type    _block_nbr,
              number_type    _next_block_nbr,
              parameter_type _gather_count);
  ```

```
• void GENERATE(number_type          _block_nbr,
                number_type          _next_block_nbr,
                parameter_type       _mean=P(),
                parameter_type       _spread=P(),
                parameter_type       _offset_interval=P(),
                parameter_type       _limit_count=P(),
                parameter_type       _priority_level=P(),
                parameter_type       _nbr_of_parameters=P(),
                fullword_option_type _fullword_option=FULLWORD_OPTION_NONE);

• void INPUT(number_type _block_nbr,
             number_type _next_block_nbr,
             number_type _input_nbr);

• void INTERN(number_type _block_nbr,
              number_type _next_block_nbr,
              void      (*_body)(void));

• void LEAVE(number_type    _block_nbr,
             number_type    _next_block_nbr,
             parameter_type _storage_nbr,
             parameter_type _nbr_of_units=P());

• void LEAVEMODEL(number_type _block_nbr,
                  number_type _next_block_nbr,
                  number_type _model_nbr,
                  number_type _output_nbr);

• void LINK(number_type    _block_nbr,
            number_type    _next_block_nbr,
            parameter_type _user_chain_nbr,
            link_type      _link_kind,
            parameter_type _branch_block_nbr=P());

• void LOGIC(number_type    _block_nbr,
             number_type    _next_block_nbr,
             logic_type     _logic_kind,
             parameter_type _logic_switch_nbr);

• void LOOP(number_type    _block_nbr,
            number_type    _next_block_nbr,
            parameter_type _parameter_nbr,
            parameter_type _branch_block_nbr);

• void MARK(number_type    _block_nbr,
            number_type    _next_block_nbr,
            parameter_type _parameter_nbr=P());

• void MATCH(number_type    _block_nbr,
             number_type    _next_block_nbr,
             parameter_type _match_block_nbr);

• void MSAVEVALUE(number_type          _block_nbr,
                  number_type          _next_block_nbr,
                  parameter_type       _m_savevalue_nbr,
                  operation_type       _operation,
                  parameter_type       _row_nbr,
                  parameter_type       _column_nbr,
                  parameter_type       _value,
                  halfword_option_type _halfword_option=HALFWORD_OPTION_NONE);

• void OUTPUT(number_type _block_nbr,
              number_type _next_block_nbr,
              number_type _output_nbr);

• void PREEMPT(number_type          _block_nbr,
               number_type          _next_block_nbr,
               parameter_type       _facility_nbr,
```

```
              priority_option_type _priority_option=PRIORITY_OPTION_NONE,
              parameter_type       _branch_block_nbr=P(),
              parameter_type       _parameter_nbr=P(),
              remove_option_type   _remove_option=REMOVE_OPTION_NONE);
```

- ```
  void PRINT(number_type    _block_nbr,
             number_type    _next_block_nbr,
             parameter_type _lower_limit=P(),
             parameter_type _upper_limit=P(),
             print_type     _print_kind=PRINT_NONE,
             ofstream      *_output_file=NULL);
  ```

- ```
  void PRIORITY(number_type        _block_nbr,
                number_type        _next_block_nbr,
                parameter_type     _new_priority_value,
                buffer_option_type _buffer_option=BUFFER_OPTION_NONE);
  ```

- ```
  void QUEUE(number_type    _block_nbr,
             number_type    _next_block_nbr,
             parameter_type _queue_nbr,
             parameter_type _nbr_of_units=P());
  ```

- ```
  void RELEASE(number_type    _block_nbr,
               number_type    _next_block_nbr,
               parameter_type _facility_nbr);
  ```

- ```
  void RETURN(number_type    _block_nbr,
              number_type    _next_block_nbr,
              parameter_type _facility_nbr);
  ```

- ```
  void SAVEVALUE(number_type          _block_nbr,
                 number_type          _next_block_nbr,
                 parameter_type       _savevalue_nbr,
                 operation_type       _operation,
                 parameter_type       _value,
                 halfword_option_type _halfword_option=HALFWORD_OPTION_NONE);
  ```

- ```
  void SEIZE(number_type    _block_nbr,
             number_type    _next_block_nbr,
             parameter_type _facility_nbr);
  ```

- ```
  void SELECT(number_type    _block_nbr,
              number_type    _next_block_nbr,
              select_type    _select_kind,
              parameter_type _parameter_nbr,
              parameter_type _lower_limit,
              parameter_type _upper_limit,
              parameter_type _comparison_value=P(),
              SNA_type       _attribute_to_examine=SNA_NONE,
              parameter_type _branch_block_nbr=P());
  ```

- ```
  void SPLIT(number_type    _block_nbr,
             number_type    _next_block_nbr,
             parameter_type _nbr_of_offspring,
             parameter_type _branch_block_nbr,
             parameter_type _serialisation_parameter=P(),
             parameter_type _nbr_of_parameters=P());
  ```

- ```
  void TABULATE(number_type    _block_nbr,
                number_type    _next_block_nbr,
                parameter_type _table_nbr,
                parameter_type _weighting_factor=P());
  ```

- ```
  void TERMINATE(number_type    _block_nbr,
                 number_type    _next_block_nbr,
                 parameter_type _termination_counter_decrement=P());
  ```
  ```

- ```
  void TEST(number_type    _block_nbr,
            number_type    _next_block_nbr,
            test_type      _test_kind,
            parameter_type _first_value,
            parameter_type _second_value,
            parameter_type _branch_block_nbr=P());
  ```

- ```
  void TRANSFER(number_type    _block_nbr,
                number_type    _next_block_nbr,
                parameter_type _selection_mode,
                parameter_type _first_block_nbr=P(),
                parameter_type _second_block_nbr=P());
  ```

- ```
  void UNLINK(number_type    _block_nbr,
              number_type    _next_block_nbr,
              parameter_type _user_chain_nbr,
              parameter_type _unlinked_branch_block_nbr,
              parameter_type _unlink_count,
              parameter_type _parameter_nbr=P(),
              parameter_type _match_argument=P(),
              parameter_type _branch_block_nbr=P());
  ```

6.3.2 Entiteitsdeclaratie-functies

- ```
  void BVARIABLE(number_type _nbr,
                 boolean_type (*_boolean_variable)(void));
  ```

- ```
  void FUNCTION(number_type _function_nbr,
                value_type (*_function)(void));
  ```

- ```
  void FUNCTION(number_type _function_nbr,
                parameter_type _argument,
                function_type _function_kind,
                count_type _nbr_of_points,
                ...);
  ```

- ```
  void MATRIX(number_type _m_savevalue_nbr,
              word_type   _word_kind,
              number_type _nbr_of_rows,
              number_type _nbr_of_columns);
  ```

- ```
  void SUBMODEL(number_type _submodel_nbr,
                ModelClass *_submodel);
  ```

- ```
  void QTABLE(number_type _table_nbr,
              number_type _queue_nbr,
              value_type  _start,
              value_type  _width,
              count_type  _count);
  ```

- ```
  void STORAGE(number_type _storage_nbr,
               count_type  _capacity);
  ```

- ```
  void TABLE(number_type          _table_nbr,
             parameter_type       _arguments,
             value_type           _start,
             value_type           _width,
             count_type           _count,
             weighted_option_type _weighted_option=WEIGHTED_OPTION_NONE,
             time_type            _time_interval=TIME_NONE);
  ```

- ```
  void VARIABLE(number_type _variable_nbr,
                value_type (*_variable)(void));
  ```

- ```
  void FVARIABLE(number_type _variable_nbr,
                 value_type (*_variable)(void));
  ```
  ```

6.3.3 Commando-functies

- `void CLEAR(void);`
- `void DOWN(number_type _model_nbr);`
- `void END(void);`
- `void INITIAL(initial_type _initial,`
  `value_type _value=NUMBER_NONE);`
- `void JOB(void);`
- `void PRINT(parameter_type _lower_limit=P(),`
  `parameter_type _upper_limit=P(),`
  `print_type _print_kind=PRINT_NONE,`
  `ofstream *_output_file=NULL);`
- `void RESET(void);`
- `void SIMULATE(ModelClass *_model,`
  `char _output_file_name[]="hgpsspp.out");`
- `void START(count_type _termination_count);`
- `void UP(void);`

6.3.4 Additionele functies

- `value_type EVALUATE(parameter_type _parameter);`
- `void *INFORMATION(information_type _information_kind,`
  `number_type _entity_nbr);`
- `boolean_type FU(logical_type _logical_kind,`
  `number_type _nbr);`
- `boolean_type FNU(logical_type _logical_kind,`
  `number_type _nbr);`
- `boolean_type FI(logical_type _logical_kind,`
  `number_type _nbr);`
- `boolean_type FNI(logical_type _logical_kind,`
  `number_type _nbr);`
- `boolean_type LS(logical_type _logical_kind,`
  `number_type _nbr);`
- `boolean_type LR(logical_type _logical_kind,`
  `number_type _nbr);`
- `boolean_type SE(logical_type _logical_kind,`
  `number_type _nbr);`
- `boolean_type SNE(logical_type _logical_kind,`
  `number_type _nbr);`
- `boolean_type SF(logical_type _logical_kind,`
  `number_type _nbr);`
- `boolean_type SNF(logical_type _logical_kind,`
  `number_type _nbr);`

6.4 Voorbeeld

Alhoewel een simulatie over het algemeen niet volledig in HGPSS++ zal uitgewerkt worden, wordt in Figuur 6.19 toch geïllustreerd hoe een systeem kan gemodelleerd en gesimuleerd worden in HGPSS++. Om de HGPSS++-kernel in een HGPSS++-programma te kunnen aanspreken, moet een *header file* worden geïncludeerd. Ten einde de meest kwetsbare delen van de kernel af te schermen voor minder ervaren gebruikers, worden twee versies van de header file ter beschikking gesteld. De *user* header file verleent enkel toegang tot de meest courante onderdelen van de kernel. Door gebruik te maken van de *advanced user* header file komt daarentegen de hele functionaliteit van de kernel ter beschikking. In Appendix B werd een afdruk van de user header file opgenomen.

Ter illustratie van een HGPSS++-simulatie werd opnieuw het warenhuis-systeem uit Hoofdstuk 1 gekozen. Het HGPSS++-programma uit Figuur 6.19 vertoont zeer sterke gelijkenissen met de HGPSS-variant uit Figuur 5.1. Het model wordt opgebouwd door een klasse met als naam `WarenhuisClass` te creëren, afgeleid van `ModelClass`. Binnen de constructor van deze klasse worden tussen `StartConstruct` en `EndConstruct` de HGPSS++-statements opgenomen die overeenstemmen met de HGPSS-statements die in de modelsectie terechtkomen (lijnen (13)-(20)). De grootste verschillen met de HGPSS blokdeclaratie-statements zijn:

- Het expliciet voorkomen van het identificatienummer van het blok en het sequentiële blok als respectievelijk eerste en tweede parameter.
- De afwijkende vorm waarin de andere parameters moeten gespecifieerd worden.
- Het ontbreken van voorzieningen voor het gebruik van labels. Alle entiteiten moeten via een indentificatienummer aangesproken worden.

De HGPSS++-statements die corresponderen met de HGPSS-statements die in de commandosectie terechtkomen, worden in de functie `main` geplaatst. Een opvallend verschil is hier de expliciete creatie van een instantie van `WarenhuisClass` door de operator `new` in het SIMULATE-statement. De blokdeclaratie-statements in de constructor van `WarenhuisClass` zullen worden uitgevoerd bij het creëren van een instantie van de klasse.

Figuur 6.20 en Figuur 6.21 tonen de uitvoer gegenereerd tijdens de afhandeling van het programma.

```
 (1) #include "hgpsspp.h"   // User header file
 (2)
 (3) class WarenhuisClass : public ModelClass
 (4)    {
 (5)      public:
 (6)       WarenhuisClass(void);
 (7)    };
 (8)
 (9) WarenhuisClass::WarenhuisClass(void)
(10)    {
(11)      StartConstruct();
(12)
(13)      GENERATE  (1,2,P(PARAMETER_VALUE,10),P(PARAMETER_VALUE,7));
(14)      ADVANCE   (2,3,P(PARAMETER_VALUE,12),P(PARAMETER_VALUE,5));
(15)      QUEUE     (3,4,P(PARAMETER_VALUE,1));
(16)      SEIZE     (4,5,P(PARAMETER_VALUE,1));
(17)      DEPART    (5,6,P(PARAMETER_VALUE,1));
(18)      ADVANCE   (6,7,P(PARAMETER_VALUE,3),P(PARAMETER_VALUE,1));
(19)      RELEASE   (7,8,P(PARAMETER_VALUE,1));
(20)      TERMINATE (8,9,P(PARAMETER_VALUE,1));
(21)
(22)      EndConstruct();
(23)    }
(24)
(25) void main(void)
(26)    {
(27)      SIMULATE        (new WarenhuisClass);
(28)      START           (500);
(29)      PRINT           ();
(30)      END             ();
(31)    }
```

Figure 6.19: HGPSS++-programma voor simulatie warenhuis-systeem

```
*******************************************************************************

   Description: HGPSS++ output file
   Name: hgpsspp.out
   Date: Tue Apr 07 16:13:17 1992

*******************************************************************************


SIMULATE - Tue Apr 07 16:13:17 1992

START: Simulation begins - Tue Apr 07 16:13:17 1992

       Simulation stops - Tue Apr 07 16:13:26 1992

PRINT - Tue Apr 07 16:13:26 1992

   Absolute clock                      : 5012.746416

   Relative clock                      : 5012.746416

   Termination count                   : 0

   Entity: BLOCK

      Number                           : 1
      Name                             : GENERATE
      Total                            : 502
      Current                          : 0

      Number                           : 2
      Name                             : ADVANCE
      Total                            : 502
      Current                          : 2

      Number                           : 3
      Name                             : QUEUE
      Total                            : 500
      Current                          : 0

      Number                           : 4
      Name                             : SEIZE
      Total                            : 500
      Current                          : 0

      Number                           : 5
      Name                             : DEPART
      Total                            : 500
      Current                          : 0

      Number                           : 6
      Name                             : ADVANCE
      Total                            : 500
      Current                          : 0

      Number                           : 7
      Name                             : RELEASE
      Total                            : 500
      Current                          : 0

      Number                           : 8
      Name                             : TERMINATE
      Total                            : 500
      Current                          : 0
```

Figure 6.20: Uitvoer van HGPSS++-programma

```
   Entity: BOOLEAN VARIABLE

   Entity: EXTERN

   Entity: FACILITY

      Number                          : 1
      Full                            : FALSE
      Preempted                       : FALSE
      Total                           : 500
      Non zero total                  : 500
      Average time                    : 2.946455
      Average non zero time           : 2.946455
      Average utilisation             : 0.295681

   Entity: FUNCTION

   Entity: INPUT

   Entity: LOGIC SWITCH

   Entity: MATRIX SAVEVALUE

   Entity: OUTPUT

   Entity: QUEUE

      Number                          : 1
      Table number                    : 0
      Total                           : 500
      Non zero total                  : 68
      Content                         : 0
      Average content                 : 0.021211
      Average non zero content        : 1.010887
      Maximum content                 : 2
      Average time                    : 0.211369
      Average non zero time           : 1.554187

   Entity: RANDOM NUMBER GENERATOR

   Entity: SAVEVALUE

   Entity: STORAGE

   Entity: SUBMODEL

   Entity: TABLE

   Entity: USER CHAIN

   Entity: VARIABLE

END - Tue Apr 07 16:13:26 1992

*******************************************************************************
```

Figure 6.21: Uitvoer van HGPSS++-programma (vervolg)

Chapter 7

De HGPSS naar HGPSS++ compiler

7.1 Inleiding

In Hoofdstuk 5 werden de HGPSS-taal en de verschillen tussen deze taal en GPSS globaal besproken. De implementatie van de kernel die uiteindelijk voor de uitvoering van een simulatie moet instaan en de manieren waarop deze kan worden aangesproken, werden in Hoofdstuk 6 behandeld. Vooraleer de kernel in actie kan treden voor de afhandeling van een HGPSS-simulatie, moet een HGPSS-programma echter eerst naar een C++-programma worden omgezet. In dit C++-programma zal de kernel worden aangeroepen door middel van ingebedde HGPSS++-statements. Voor de omzetting van een HGPSS- naar een C++-programma[1] werd een precompiler ontwikkeld. Deze luistert naar de naam *HGPSS naar HGPSS++ compiler*. De voornaamste taken van de precompiler zijn:

- Het genereren van een C++-programmaframe waarbinnen de naar HGPSS++ vertaalde HGPSS-statements kunnen geplaatst worden.
- Het vertalen van de HGPSS blokdeclaratie-, entiteitsdeclaratie en commando-statements.
- Het wegwerken van de in het HGPSS-programma gebruikte labels door deze om te zetten naar unieke en geschikte identificatienummers.
- Het genereren van unieke identificatienummers voor de gebruikte blokken.
- Het vertalen van HGPSS-variables, -boolean variables en -fullword variables naar C++-functies. Deze C++-functies zullen dan als entiteit gebruikt worden.
- Het verwerken van commentaar en ingebedde C++-code.
- Het uitvoeren van een controle op de syntax van het HGPSS-programma.

7.2 LEX en YACC

Een precompiler die de geschetste taken vervult, werd ontwikkeld door gebruik te maken van de compiler construction tools *LEX* en *YACC*.

[1] In hetgeen volgt zal een C++-programma met ingebedde HGPSS++-instructies ook als HGPSS++-programma worden aangeduid.

LEX [Mason & Brown 1990, Forsyth 1982] is een *lexical analyser generator*. Hiermee wordt bedoeld dat LEX uitgaande van een high-level beschrijving van een aantal *reguliere expressies*, een *lexicale analysator* genereert [Aho, Sethi & Ullman 1986]. De analysator wordt gegenereerd in de vorm van een programma in een general-purpose programmeertaal. Deze taal wordt de gasttaal genoemd. Naast de beschrijving van de door de analysator te herkennen reguliere expressies worden in de aan LEX geleverde broncode ook de acties beschreven die moeten uitgevoerd worden als de analysator een bepaalde expressie herkend heeft. Deze acties worden genoteerd in de gasttaal en duidelijk gescheiden van de beschrijving van de reguliere expressies. Klassiek is het gebruik van C als gasttaal. Ook in het geval van de HGPSS-precompiler werd een versie van LEX gebruikt waarin C de gasttaal is.

Om tot een volledige compiler te komen wordt een lexicale analysator meestal gecombineerd met een *parser* [Aho, Sethi & Ullman 1986]. YACC (Yet Another Compiler-Compiler) [Mason & Brown 1990] is een *parser generator*. Uitgaande van een high-level beschrijving van een context-vrije grammatica wordt een parser gegenereerd in de vorm van een programma in een general-purpose programmeertaal, de gasttaal. YACC genereert tevens tabellen die de parser toelaten het *LR(1)* parsing-algoritme uit te voeren. Naast de beschrijving van de context-vrije grammatica worden in de aan YACC geleverde broncode ook de acties beschreven die moeten uitgevoerd worden gedurende de parsing-operatie. De acties worden genoteerd in de gasttaal en duidelijk gescheiden van de beschrijving van de grammatica. Zoals bij LEX is het gebruik van C als gasttaal klassiek. De versie van YACC aangewend in het kader van de HGPSS-precompiler gebruikt eveneens C als gasttaal.

De in de LEX- en YACC-beschrijving van een compiler te gebruiken syntax en de verdere details van deze programma's zullen hier niet beschreven worden. Hiervoor wordt verwezen naar [Mason & Brown 1990].

7.3 Implementatie

7.3.1 Algemeen

Aangezien gepoogd werd om de HGPSS++ kernel-interface zo nauw mogelijk te laten aanleunen bij de HGPSS-syntax, is het omzetten van een HGPSS-programma naar de HGPSS++-variant in principe vrij eenvoudig. Elk HGPSS-statement kan onmiddellijk naar het corresponderende HGPSS++-statement worden omgezet zonder dat eerst een parse-tree moet opgebouwd worden. Er zijn echter wel een aantal eigenschappen van de GPSS-syntax en bijgevolg ook van de HGPSS-syntax, die het compiler-constructieproces bemoeilijken.

Aangezien versies van LEX en YACC die C als gasttaal gebruiken ter beschikking staan voor de meeste UNIX- en MSDOS-machines, werd de LEX- en YACC-broncode van de precompiler zodanig opgebouwd dat deze portabel is, mits enkele onvermijdelijke aanpassingen uit te voeren. De broncode werd oorspronkelijk ontwikkeld voor gebruik bij een MSDOS-versie van LEX en YACC. Na het uitvoerig uittesten van de precompiler werd de broncode dan aangepast voor gebruik bij de op UNIX-systemen gebruikelijke versies van LEX en YACC, zodat de precompiler momenteel onder beide besturingssystemen beschikbaar is. Aangezien de broncode van de HGPSS++-kernel volledig portabel is, is het totale HGPSS-systeem beschikbaar op zowel MSDOS- als UNIX-systemen die over een C/C++-compiler beschikken.

In hetgeen volgt wordt de implementatie van de HGPSS-precompiler besproken. Hierbij zal niet in detail gegaan worden aangezien de kans vrij klein is dat een gebruiker genoodzaakt wordt om de precompiler aan te passen. De bespreking zal dus minder uitgebreid zijn dan die van de HGPSS++-kernel. De kans dat de gebruiker met deze kernel te maken krijgt is immers wel reëel. In de Engelstalige handleiding die zich in Appendix A bevindt, is een extended Backus Naur form-beschrijving van de HGPSS-taal te vinden. Deze leunt nauw aan bij de LEX- en YACC-beschrijving van de precompiler.

De volledige HGPSS-precompiler bestaat in de eerste instantie uit:

- De functie `yylex` gegenereerd door de LEX-compiler uitgaande van de LEX-beschrijving.
- De functie `yyparse` gegenereerd door de YACC-compiler uitgaande van de YACC-beschrijving.

Naast deze functies worden ook nog een aantal andere functies gebruikt.

- De functie `main` analyseert de parameters meegegeven op de commandolijn bij de invocatie van de precompiler en leidt hieruit de namen van de bestanden af die door de precompiler moeten gebruikt worden. De functie zorgt er ook voor dat de parser wordt aangeroepen. De bestanden die door de precompiler moeten gebruikt worden zijn:
  - Het *invoerbestand*: Dit bestand bevat het te vertalen HGPSS-programma.
  - Het *uitvoerbestand*: In dit bestand komt het vertaalde programma terecht.
  - Het *header-bestand*: In dit bestand komt voor elke modelsectie in het invoerbestand, een declaratie terecht van de na vertaling resulterende modelklasse. In dit bestand wordt voor elke modelsectie ook informatie opgenomen in verband met de gebruikte labels. Het header-bestand wordt in het uitvoerbestand geïncludeerd.
  - Het *variable-bestand*: Dit bestand zal vertaalde variables, boolean variables en fullword variables bevatten. In HGPSS kunnen de variable-entiteiten op twee manieren gedefinieerd worden: door een expressie op te geven of door een C++-functie te koppelen aan de variable. In HGPSS++ is enkel de tweede manier mogelijk. Bij HGPSS-variables gedefinieerd door een expressie, moet de expressie bij omzetting naar HGPSS++ dan ook vertaald worden naar een C++-functie. Deze functies komen in het variable-bestand terecht. Het variable-bestand wordt in het uitvoerbestand geïncludeerd.
- De functies `lexgetc` en `UnGetc` werken rechtstreeks in op het invoerbestand. `lexgetc` is een functie uit de LEX-bibliotheek die werd geherdefinieerd om naast het inlezen van een karakter ook een aantal andere operaties uit te voeren. `UnGetc` is een herdefinitie van de standaard C-functie met dezelfde naam. De functie laat toe om een karakter terug te plaatsen in het buffer geassocieerd met het invoerbestand. Buiten dit terugplaatsen voert de functie ook nog enkele andere acties uit.
- Een aantal functies worden gebruikt om gegevens vanuit `yylex` verder te verwerken en door te spelen naar `yyparse`. Deze functies zijn:
  - `ProcessInteger`
  - `ProcessReal`
  - `ProcessRelationalParameter`
  - `ProcessLogicalParameter`
  - `ProcessParameter`
  - `ProcessIdentifier`
- Tijdens de parsing-operatie wordt gebruik gemaakt van twee symbol tables, één voor het opslaan van labels en de andere voor het opslaan van de namen van de modelklassen waartoe de gebruikte submodellen behoren. Om de *label symbol table* te manipuleren worden volgende functies gebruikt:
  - `InstallLabel`

  - `PrintLabels`
  - `DeleteLabels`

  De *submodel symbol table* wordt door volgende functies gemanipuleerd:

  - `InstallSubmodel`
  - `RetrieveSubmodel`
  - `DeleteSubmodels`

- De functies `yyerror` en `ArgumentError` worden aangeroepen bij het optreden van een fout. `ArgumentError` staat in voor de afhandeling van een foutieve invocatie van de precompiler terwijl `yyerror` in alle andere gevallen gebruikt wordt. `yyerror` is de herdefinitie van een functie die deel uitmaakt van de LEX-bibliotheek.

- Een aantal functies worden door `yyparse` gebruikt om enkele minder belangrijke operaties uit te voeren.

  - `IsInteger` wordt gebruikt om na te gaan of een gespecifieerde karaktersliert een geheel getal voorstelt.
  - `IsExpression` wordt gebruikt om na te gaan of een gespecifieerde karaktersliert een tussen dubbele aanhalingtekens geplaatste C++-expressie bevat.
  - `IntegerToString` tenslotte zet een geheel getal om naar een karaktersliert.

- De functies

  - `PrintParameter`,
  - `PrintLogical` en
  - `PrintInitial`

  worden gebruikt om drie specifieke klassen HGPSS-parameters eerst te vertalen naar het HGPSS++-equivalent en dit daarna af te drukken in het uitvoer- of het variable-bestand. Voor de omzetting maakt elk van de drie functies gebruik van een eigen mini-compiler. Deze mini-compilers bestaan uit een zeer eenvoudige lexicale analysator en een parser.

In Figuur 7.1 is een overzicht gegeven van de functies waaruit de precompiler is opgebouwd en de manier waarop deze interageren.

7.3.2 Lexicale analysator

De lexicale analysator leest karakters in vanuit het invoerbestand via de functie `lexgetc`. In de sliert van ingelezen karakters wordt geprobeerd een deelsliert of *lexeme* te vinden dat in overeenstemming is met één van de patronen of *tokens*, gespecifieerd door de reguliere expressies. Als een dergelijke deelsliert gevonden is, wordt een code corresponderende met het token waarvoor de deelsliert voldoet aan het definiërende patroon, doorgestuurd naar de parser. Eventueel wordt de deelsliert zelf ook doorgestuurd. Het token wordt doorgestuurd door middel van een interne LEX-variabele die ook door YACC gekend is; de deelsliert door middel van een zelfgedefinieerde karaktersliert-variabele.

De door de lexicale analysator herkende patronen zijn:

- Het `\n`-karakter.

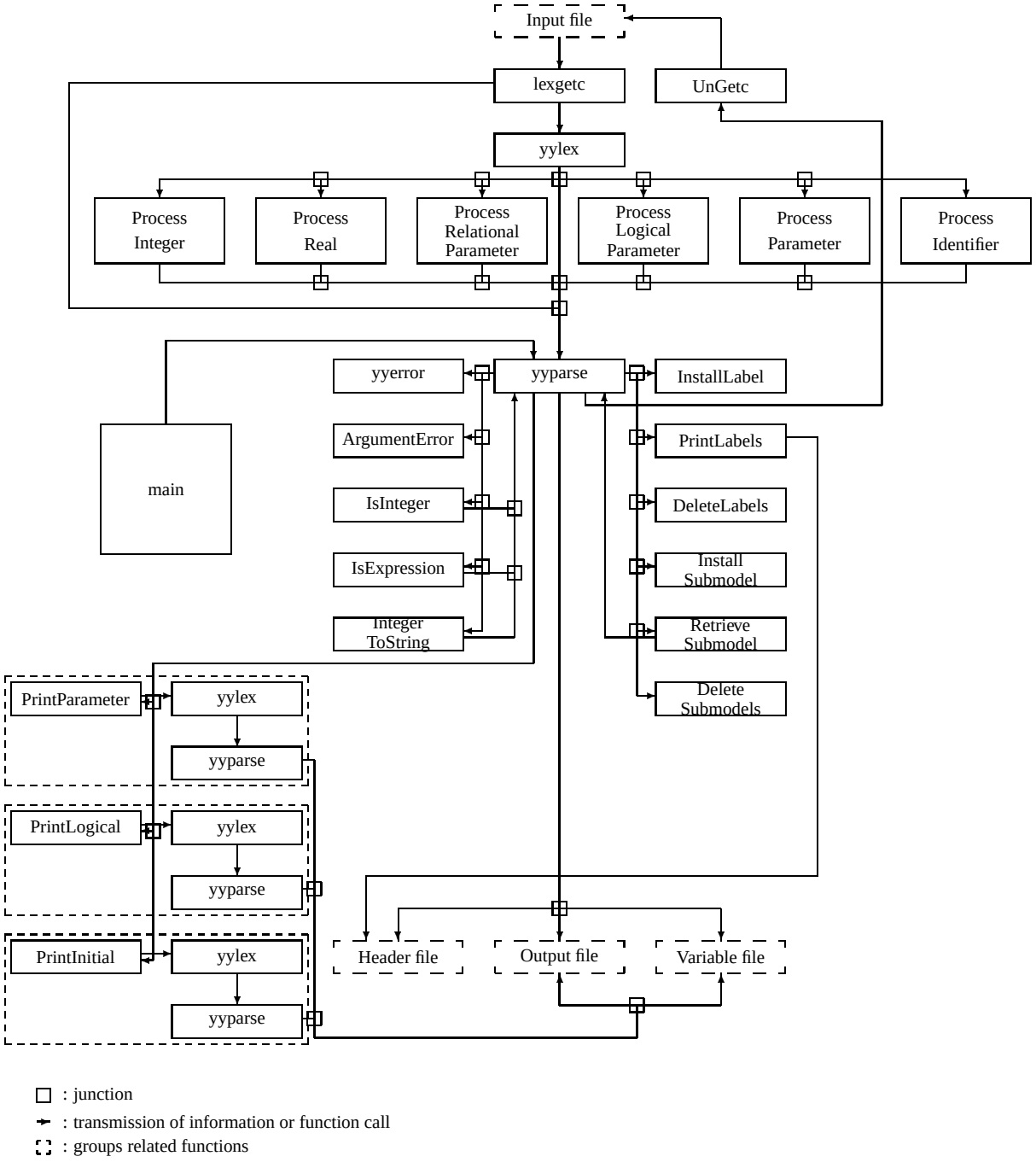

Figure 7.1: Proces-model van de HGPSS naar HGPSS++ compiler

,	/	-
+	*	@
(	)	"

Table 7.1: Symbolen

'E' 'L'	'G'	'GE' 'NE'
	'LE'	

Table 7.2: Relationele operatoren

FU	FNU	FI
FNI	LS	LR
SE	SNE	SF
SNF		

Table 7.3: Logische attributen

- De in Tabel 7.1 opgenomen *symbolen.*
- De sequentie `%{`.
- Een opeenvolging van één of meerdere spaties, `\t`- of `\r`-karakters, aangeduid door de benaming *white space.*
- Een *geheel getal* opgebouwd uit een opeenvolging van één of meerdere cijfers.
- Een *reëel getal* bestaande uit een mantisse, het exponentsymbool `e` of `E` en een exponent. Mantisse en exponent zijn optioneel. De mantisse mag een decimaal punt bevatten en moet verder uit niets dan cijfers bestaan. De exponent moet volledig uit cijfers opgebouwd zijn.
- De in Tabel 7.2 opgenomen *relationele operatoren.*
- *Logische parameters*, deze bestaan uit een *logisch attribuut* onmiddellijk gevolgd door één van de volgende mogelijkheden:
  - Een geheel getal of een *expressie.* Door expressie wordt een karaktersliert aangeduid waarin geen " voorkomt, begrensd door dubbele aanhalingstekens.
  - Een $-karakter gevolgd door een *identificator.* Een identificator is een sequentie van één of meerdere hoofdletters, kleine letters, cijfers, _ en $ waarbij het eerste karakter verplicht een letter, _ of $ moet zijn.
  - Een *-karakter gevolgd door een geheel getal of een expressie.

  De mogelijke logische attributen zijn in Tabel 7.3 opgenomen.
- *Parameters*, deze kunnen de volgende vormen aannemen:
  - Eén van de mnemonics uit Tabel 7.4, *enkelvoudige SNA's*[2] genoemd omdat ze door geen andere karakters gevolgd worden.
  - Een *gewone SNA* gevolgd door

[2] Standard Numerical Attribute

C1	PR	M1

Table 7.4: Enkelvoudige SNA's

N	W	F	FC
FR	FT	FN	Q
QA	QC	QM	QT
QX	QZ	RN	X
XH	R	S	SA
SC	SR	SM	ST
TB	TC	TD	P
MP	CA	CC	CH
CM	CT	BV	V

Table 7.5: Gewone SNA's

MH	MX

Table 7.6: Matrix SNA's

  - Een geheel getal of een expressie.
  - Een $-karakter gevolgd door een identificator.
  - Een *-karakter gevolgd door een geheel getal of een expressie.

  De gewone SNA's zijn de in Tabel 7.5 opgenomen mnemonics.
- Een *matrix SNA* gevolgd door
  - een geheel getal of een expressie,
  - een $-karakter gevolgd door een identificator,
  - een *-karakter gevolgd door een geheel getal of een expressie

  en afgesloten door twee gehele getallen of expressies, gescheiden door een comma en ingesloten tussen (en). De matrix SNA's zijn opgenomen in Tabel 7.6.

- Een groot aantal *gereserveerde woorden.* Alle gereserveerde woorden werden in Tabel 7.7 opgenomen.

- Uiteindelijk worden ook nog identificatoren herkend. Deze bestaan zoals reeds vermeld uit een opeenvolging van één of meerdere hoofdletters, kleine letters, cijfers, _ en $ waarbij de eerste letter verplicht een letter, _ of $ moet zijn. Uiteraard vallen ook de gereserveerde woorden en nog enkele andere categorieën eerder beschreven patronen onder de definitie van een identificator. Als er echter *clashing* zou optreden wordt rekening gehouden met de prioriteit van de patronen. De in deze opsomming eerst beschreven patronen hebben de hoogste prioriteit.

In de lexicale analysator worden naast enkele eenvoudige ook enkele vrij ingewikkelde patronen herkend, zoals de logische parameters en de parameters. De reden hiervoor is dat de tabellen gegenereerd vanuit de gespecifieerde context-vrije grammatica in bepaalde versies van YACC, slechts een vrij beperkte omvang kunnen aannemen. Om deze tabellen in omvang te limiteren werd een deel van het uit te voeren werk overgeheveld naar de lexicale analysator.

Als een patroon of token herkend is, moet in het eenvoudigste geval een code overstemmend met het token naar de parser worden doorgestuurd. Als bij herkenning van een token enkel deze operatie moet uitgevoerd worden, werd in de LEX-broncode een corresponderende actie opgenomen ná de definitie van het token. Als er meerdere acties moeten ondernomen worden zoals het doorsturen van de code, het doorsturen van het lexeme en eventueel nog andere acties, werden deze acties gebundeld in een aparte C-functie. Een aanroep van deze functie werd in de LEX-broncode opgenomen ná de definitie van het token. De aldus gebruikte functies zijn:

ABSOLUTECLOCK	ADVANCE	ALL
ASSEMBLE	ASSIGN	BACK
BLOCK	BOTH	BUFFER
BVARIABLE	CLEAR	COMMAND
CURRENTEVENTCHAIN	DEPART	DOWN
E	END	ENDCOMMAND
ENDMODEL	ENTER	ENTERMODEL
EXTERN	F	FACILITY
FIFO	FUNCTION	FUTUREEVENTCHAIN
FVARIABLE	G	GATE
GATHER	GE	GENERATE
H	I	IA
INITIAL	INPUT	INTEGER
INTERN	JOB	L
LE	LEAVE	LEAVEMODEL
LIFO	LINK	LOGIC
LOGICSWITCH	LOOP	LR
LS	M	MARK
MATCH	MATCHINGCHAINCHAIN	MATRIX
MAX	MIN	MODEL
MSAVEVALUE	NE	NI
NM	NU	OUTPUT
PR	PREEMPT	PRINT
PRIORITY	QTABLE	QUEUE
R	RANDOMNBRGENERATOR	RE
RELATIVECLOCK	RELEASE	RESET
RETURN	RT	S
SAVEVALUE	SE	SEIZE
SELECT	SF	SIMULATE
SNE	SNF	SPLIT
START	STORAGE	SUBMODEL
TABLE	TABULATE	TERMINATE
TERMINATIONCOUNTER	TEST	TRANSACTION
TRANSFER	U	UNLINK
UP	USERCHAIN	VARIABLE
X		

Table 7.7: Gereserveerde woorden in HGPSS

- `ProcessInteger`
- `ProcessReal`
- `ProcessRelationalParameter`
- `ProcessLogicalParameter`
- `ProcessParameter`
- `ProcessIdentifier`

De syntax waarin de LEX-broncode moet gespecifieerd worden, vertoont een aantal verschillen naargelang de gebruikte LEX-implementatie. Concreet vertoont de syntax vereist door de MSDOS LEX-implementatie deel uitmakend van het *DECUS C Language System*, waarin de lexicale analysator initieel ontwikkeld werd, een aantal verschillen met de versies van LEX die op UNIX-systemen beschikbaar zijn. Bij het overdragen van de LEX-broncode van MSDOS naar UNIX waren dan ook enkele wijzigingen aan de syntax noodzakelijk. Semantisch is er echter geen verschil tussen de MSDOS- en de UNIX-versie van de lexicale analysator.

7.3.3 Parser

De parser probeert de door de lexicale analysator geleverde tokens te combineren tot regels uit de grammatica. Tijdens dit proces wordt uitvoer gegenereerd naar het uitvoerbestand, het header-bestand en het variable-bestand.

Er werd veel aandacht besteed aan het uitzicht van de drie uitvoerbestanden. Er werd immers van het principe uitgegaan dat het uit het HGPSS-programma resulterende HGPSS++-programma niet enkel voor de C++-compiler maar ook voor de gebruiker leesbaar moet blijven. Om dit doel te bereiken werd gebruik gemaakt van een aantal technieken.

- Bij het genereren van het HGPSS++-programma wordt consistent gebruik gemaakt van intanding en alignering om de structuur van het programma te accentueren.
- Om de HGPSS++-code resulterend uit de HGPSS-secties duidelijk van elkaar te scheiden worden lege regels gebruikt.
- Alle gegenereerde bestanden worden voorzien van een hoofding en een aantal afsluitende regels.
- Alle in het HGPSS-programma op de toegelaten manieren opgenomen commentaar wordt in zijn totaliteit overbracht naar het HGPSS++-programma.
- Om de in het HGPSS-programma gebruikte labels ook in het HGPSS++-programma te laten voorkomen, wordt gebruik gemaakt van een speciale techniek. Hierbij wordt voor elke voorkomende modelklasse een structuur geconstrueerd met constanten als velden. De velden zijn genoemd naar de in het HGPSS-programma gebruikte labels en de inhoud van de velden zijn de door de precompiler aan de labels geassocieerde identificatienummers.

Globaal gezien tracht de parser vijf soorten statements op te sporen in het invoerbestand:

- *sectie markering*-statements
- *entiteitsdeclaratie*-statements

- *blokdeclaratie*-statements
- *commando*-statements
- *tekst*

Met tekst worden eigenlijk geen statements bedoeld maar stukken uit het invoerbestand die door de precompiler gewoon naar het uitvoerbestand moeten gekopieerd worden. Onder deze categorie vallen

- regels met commentaar aangeduid door een `*`,
- regels met C++-code aangeduid door een `-`,
- stukken uit het invoerbestand bestaande uit één of meerdere regels C++-code omgeven door `%{` en `%}`,
- lege regels.

In hetgeen volgt wordt kort de manier besproken waarop deze categorieën statements worden behandeld.

Sectie markering-statements

Sectie markering-statements of *directives* duiden het begin en het einde aan van een model- of commandosectie. Bij het vertalen van een dergelijk statement worden de delen van het programma-frame gegenereerd waarbinnen de vertaalde entiteitsdeclaratie-, blokdeclaratie- en commando-statements zullen geplaatst worden. Een sectie markering-statement ziet er syntactisch als volgt uit:

```
[WHITE_SPACE] directive [WHITE_SPACE comment] NEW_LINE
```

Bij de interpretatie van deze en volgende syntax-specificaties moeten volgende regels in acht genomen worden:

- Een meta-variabele bestaande uit hoofdletters duidt een *terminal* aan.
- Een meta-variabele bestaande uit kleine letters duidt een *non-terminal* aan.
- `[]` wijst op een optioneel deel.
- `()` wordt gebruikt voor groepering.
- `|` scheidt alternatieven.
- Karakters die tussen enkele aanhalingsteken werden geplaatst moeten letterlijk worden geïnterpreteerd.

De meta-variable `directive` slaat op één van volgende mogelijkheden:

- `COMMAND`: Duidt het begin van de commandosectie aan. Naar het uitvoerbestand wordt volgende C++-code gezonden:

  ```
  void main(void)
     {
  ```

- `ENDCOMMAND`: Duidt het einde van de commandosectie aan. Naar het uitvoerbestand wordt volgende C++-code gezonden:

```
}
```

- `MODEL WHITE_SPACE model_name ['('parameter_list')']`: Duidt het begin van een modelsectie aan. Als er geen parameter-lijst gespecifieerd is, wordt naar het uitvoerbestand volgende C++-code gezonden:

  ```
  <model_name>Class::<model_name>Class(void)
    {
      StartConstruct();
  ```

  In het andere geval wordt dit:

  ```
  <model_name>Class::<model_name>Class(<parameter_list>)
    {
      StartConstruct();
  ```

  Hierbij worden de meta-variabelen die tussen < en > geplaatst werden, vervangen door de actuele bij de meta-variable horende karaktersliert. In het header-bestand komt als er geen parameter-lijst opgegeven werd, het volgende terecht:

  ```
  class <model_name> : public ModelClass
    {
      public:
       <model_name>Class(void);
    }
  ```

  Als er wel een parameter-lijst opgegeven werd, wordt dit:

  ```
  class <model_name> : public ModelClass
    {
      public:
       <model_name>Class(<parameter_list>);
    }
  ```

  De parameter-lijst wordt vanuit het invoerbestand letterlijk gekopieerd naar het uitvoer- en het header-bestand.

- `ENDMODEL`: Duidt het einde van een modelsectie aan. Naar het uitvoerbestand wordt volgende C++-code gezonden:

  ```
      EndConstruct();
  }
  ```

De op het einde van een statement opgenomen commentaar, aangeduid door `comment`, wordt vanuit het invoerbestand letterlijk gekopieerd naar het uitvoerbestand en tussen `/*` en `*/` geplaatst.

BVARIABLE	FUNCTION	MATRIX
SUBMODEL	QTABLE	STORAGE
TABLE	VARIABLE	FVARIABLE

Table 7.8: Entiteitsdeclaratie-statements

Entiteitsdeclaratie-statements

Een entiteitsdeclaratie-statement ziet er syntactisch als volgt uit:

```
[WHITE_SPACE] ( INTEGER | IDENTIFIER | '"'expression'"' ) WHITE_SPACE
entity_declaration_body [WHITE_SPACE comment] NEW_LINE
```

Met `entity_declaration_body` wordt zowel het sleutelwoord van het statement bedoeld als de eventuele door comma's gescheiden parameters. De parameters zijn van het sleutelwoord gescheiden door white space. De sleutelwoorden van de entiteitsdeclaratie-statements zijn in Tabel 7.8 opgenomen. Voor elk statement is in de grammatica een afzonderlijke regel opgenomen die de vorm van het statement en vooral de parameter-lijst, gedetailleerd weergeeft. Door zo weinig mogelijk variatie toe te laten kunnen heel wat fouten opgespoord worden. Een HGPSS-programma dat zonder fouten door de precompiler kan verwerkt worden, zal een HGPSS++-programma leveren dat met grote zekerheid zonder verdere problemen kan compileerd worden door een C++-compiler.

Een entiteit kan gelabeled worden door een nummer, een karaktersliert of een tussen dubbele aanhalingstekens geplaatste C++-expressie. Een nummer wordt gewoon overgenomen in het resulterende HGPSS++-statement, als identificatienummer. Ook een expressie wordt gewoon overgenomen, ontdaan van de dubbele aanhalingstekens. Als de entiteit voorzien is van een karaktersliert als label, wordt aan dit label door de precompiler automatisch een identificatienummer geassocieerd. Dit identificatienummer is afkomstig van een teller die bij het begin van elke modelsectie op een bepaalde en steeds dezelfde waarde wordt geïnitialiseerd. Als een identificatienummer met een label moet geassocieerd worden, wordt de inhoud van de teller genomen, waarna de teller met één wordt geïncrementeerd. Alle door karakterslierten benoemde entiteiten zullen dus consecutief genummerd worden, beginnend vanaf een bepaalde waarde. Bij het benoemen van entiteiten door een nummer of expressie moet er veiligheidshalve voor gezorgd worden dat de aldus toegekende identificatienummers kleiner zijn dan het nummer waarmee de teller geïnitialiseerd wordt. Label en bijhorend identificatienummer worden opgenomen in de label symbol table. Deze table is geïmplementeerd als een enkelvoudig geketende lineaire lijst. De elementen van de keten zijn structuren. Elke structuur heeft als velden

- het label zelf, met andere woorden de karaktersliert,
- het aan het label geassocieerde identificatienummer en
- een wijzer naar het volgende element.

In het resulterende HGPSS++-statement wordt het aan de karaktersliert geassocieerde identificatienummer niet rechtstreeks opgenomen, maar wel in de vorm van een constante die er als volgt uitziet:

```
<model_name>Struct.<label>
```

Hierbij is `model_name` de naam van de modelklasse waarvan de precompiler de bijbehorende modelsectie aan het verwerken is en `label` de karaktersliert gebruikt als label. De gebruikte structuur is er één die door de precompiler zelf gegenereerd wordt en in het header-bestand geplaatst. Dit gebeurt op het einde van de verwerking van een modelsectie, uitgaande van de informatie opgeslagen in de label symbol table.

De gegenereerde structuur krijgt als naam `<model_name>Struct` en is opgebouwd uit velden die alle constant zijn. De velden zijn genoemd naar de in de symbol table opgeslagen label-karakterslierten. Aan de velden worden de in de symbol table opgeslagen identificatienummers toegekend. Aangezien de aldus verkregen structuur in het header-bestand wordt opgenomen en het header-bestand in het uitvoerbestand wordt geïncludeerd, kunnen de velden van de structuur gebruikt worden in HGPSS++-statements. Door het toepassen van deze werkwijze worden de in het HGPSS-programma gebruikte labels gereflecteerd in het HGPSS++-programma. Aangezien alle constanten horende bij een model in een specifieke structuur worden gegroepeerd, wordt eveneens de lokaliteit van de labels gewaarborgd.

Als in een HGPSS-statement in één van de parameters gerefereerd wordt naar een entiteit door middel van een karaktersliert-label, wordt nagegaan of dit label zich in de symbol table bevindt. Als dit niet het geval is, wordt de symbol table uitgebreid met een element. In dit element wordt het label opgeslagen maar aan dat label wordt nog geen identificatienummer geassocieerd. De associatie gebeurt ofwel bij de verwerking van het entiteitsdeclaratie-statement corresponderende met de entiteit of bij het afsluiten van de verwerking van een modelsectie. Als bij de verwerking van een entiteitsdeclaratie-statement geconstateerd wordt dat voor het gebruikte karaktersliert-label reeds een element in de symbol table is opgenomen, wijst dit er op dat reeds naar de entiteit gerefereerd is in een ander statement. Aan het label wordt een identificatienummer geassocieerd door gebruik te maken van de eerder beschreven teller. Sommige entiteiten hoeven niet door een entiteitsdeclaratie-statement gedeclareerd te worden. Aan een label horende bij een dergelijke entiteit kan dan ook geen identificatienummer geassocieerd worden op de zoëven beschreven manier. Bij het afsluiten van de verwerking van een modelsectie wordt de symbol table echter overlopen en aan alle labels waaraan nog geen identificatienummer werd toegekend, wordt een nummer geassocieerd uitgaande van de teller. De label symbol table wordt uit het geheugen verwijderd bij het afsluiten van de verwerking van een modelsectie.

Het SUBMODEL-statement maakt niet enkel gebruik van de label symbol table, maar ook van de submodel symbol table. De submodel symbol table is een enkelvoudig geketende lineaire lijst, bestaande uit elementen in de vorm van een structuur. Elke structuur bestaat uit

- een label in de vorm van een nummer of een karaktersliert,
- de naam van de modelklasse waarvan het submodel aangeduid door het label, een instantie is,
- een wijzer naar het volgende element.

De submodel symbol table is van belang bij de ENTERMODEL- en LEAVEMODEL-blokdeclaraties. Bij SUBMODEL-statements gebruik makend van een expressie als label wordt geen element in de submodel symbol table opgenomen. De submodel symbol table wordt bij de beëindiging van de verwerking van een modelsectie uit het geheugen verwijderd.

Blokdeclaratie-statements

Blokdeclaratie-statements zien er syntactisch als volgt uit:

```
[WHITE_SPACE] [IDENTIFIER WHITE_SPACE] block_declaration_body [WHITE_SPACE comment] NEW_LINE
```

`block_declaration_body` wijst zowel op het sleutelwoord van het statement als op de door comma's gescheiden parameters. Voor elk blokdeclaratie-statement is in de grammatica een afzonderlijke regel opgenomen die de vorm van een dergelijk statement en vooral de parameter-lijst, gedetailleerd weergeeft. Hierdoor kunnen heel wat fouten opgespoord worden, maar wordt de omvang van de grammatica wel vrij uitgebreid aangezien het aantal blokdeclaratie-statements groot is. De sleutelwoorden van de blokdeclaratie-statements zijn in Tabel 7.9 opgenomen.

ADVANCE	ASSEMBLE	ASSIGN
BUFFER	DEPART	ENTER
ENTERMODEL	EXTERN	GATE
GATHER	GENERATE	INPUT
INTERN	LEAVE	LEAVEMODEL
LINK	LOGIC	LOOP
MARK	MATCH	MSAVEVALUE
OUTPUT	PREEMPT	PRINT
PRIORITY	QUEUE	RELEASE
RETURN	SAVEVALUE	SEIZE
SELECT	SPLIT	TABULATE
TERMINATE	TEST	TRANSFER
UNLINK		

Table 7.9: Blokdeclaratie-statements

In een blokdeclaratie-statement kan optioneel een karaktersliert-label voorkomen. Dit label wordt opgeslagen in de label symbol table, waarvan ook de entiteitsdeclaratie-statements gebruik maken. Aan het label wordt een identificatienummer geassocieerd dat afkomstig is van een teller die bij het begin van de verwerking van een modelsectie op nul ingesteld wordt. Bij de vertaling van elk blokdeclaratie-statement wordt de teller met één geïncrementeerd. De teller geeft dus steeds het aantal reeds vertaalde blokdeclaratie-statements in de huidige modelsectie aan. In het HGPSS++-statement resulterend uit een HGPSS-blokdeclaratie moeten de identificatienummers van het blok en van het sequentiële blok opgenomen worden. Als een blok niet voorzien is van een label wordt hiervoor respectievelijk de inhoud van de teller en de inhoud van de teller vermeerderd met één ingevuld. In het andere geval wordt respectievelijk gebruik gemaakt van een constant veld van een structuur zoals bij de entiteitsdeclaraties, en van de inhoud van de teller vermeerderd met één.

De ENTERMODEL- en LEAVEMODEL-statements maken naast de label symbol table ook gebruik van de submodel symbol table. Deze beide statements hebben als parameters de naam van een submodel en de naam van een in- of output van het submodel. De naam van de in- of output kan een nummer of een karaktersliert zijn. Als de naam een nummer is, is er bij de vertaling geen probleem. Als de naam echter een karaktersliert is, kan het niet vertaald worden als

```
<model_name>Struct.<label>
```

met `model_name` de naam van de modelklasse die actueel verwerkt wordt. De in- of output maakt immers geen deel uit van de actueel behandelde modelklasse maar van de modelklasse aangeduid door de eerste parameter van het bijhorende SUBMODEL-statement. De vertaling moet daarom als volgt geschieden:

```
<submodel_name>Struct.<label>
```

Hierin is `submodel_name` de naam van de modelklasse waartoe het submodel behoort. Deze naam kan in de submodel symbol table opgespoord worden door gebruik te maken van de naam van het submodel als index. Deze naam zal slechts aanwezig zijn in de tabel als het submodel reeds gedeclareerd werd door middel van een SUBMODEL-statement. In een ENTERMODEL- en LEAVEMODEL-statement kan dus slechts gebruik gemaakt worden van de naam van een submodel als dit submodel vóór het ENTERMODEL- of LEAVEMODEL-statement werd gedeclareerd.

Commando-statements

De commando-statements zien er syntactisch vrij eenvoudg uit:

```
[WHITE_SPACE] command_body [WHITE_SPACE comment] NEW_LINE
```

CLEAR	DOWN	END
INITIAL	JOB	PRINT
RESET	SIMULATE	START
UP		

Table 7.10: Commando-statements

Hierbij staat `command_body` zowel voor het sleutelwoord van het statement als voor de parameters. De sleutelwoorden van de commando-statements zijn in Tabel 7.10 opgenomen. Als in een commando-statement via een karaktersliert-label gerefereerd wordt naar een entiteit kan dit pas gebeuren nadat een model als actief model is gedeclareerd via het SIMULATE-statement. Pas als dit gebeurt is, kan er een naam ingevuld worden als `model_name` in

```
<model_name>Struct.<label>
```

Tekst

De syntax waarin de aan de MSDOS-versie en de UNIX-versie van YACC geleverde parser-beschrijving moet gespecifieerd worden, is identiek. In principe bestaat er dan ook geen verschil tussen de YACC-beschrijving van de HGPSS-precompiler voor gebruik bij de MSDOS- en UNIX-versie van YACC. Aangezien echter bij de verwerking van commentaar en C++-code, delen van het invoerbestand rechtstreeks gekopieerd worden naar het uitvoerbestand, moet de parser op een low-level manier interageren met het invoerbestand. Hierbij komt de parser in het vaarwater van de lexicale analysator terecht. Gezien er verschillen bestaan in de interne werking van de door de MSDOS- en UNIX-versies van LEX gegenereerde functie `yylex`, zal het rechtstreeks kopiëren ook anders moeten verlopen. Op de plaatsen waar rechtstreeks moet gekopieerd worden bestaan er bijgevolg wel verschillen tussen de YACC-beschrijving voor MSDOS en UNIX. Globaal gezien komt het er op neer dat de MSDOS-versie van `yylex` steeds gebruik maakt van een *lookahead*-karakter [Aho, Sethi & Ullman 1986], terwijl de UNIX-versie dit niet altijd doet.

Het header-bestand

Zoals eerder vermeld komen in het header-bestand twee stukken C++-code terecht voor elk in het invoerbestand beschreven model.

- In eerste instantie wordt een declaratie opgenomen van de klasse die uit de HGPSS-beschrijving van het model wordt afgeleid. De definitie van de klasse is over twee locaties verspeid.
  - Aangezien alle modelklassen afgeleid worden van `ModelClass`, is het grootste deel van de definitie te vinden in de HGPSS++-kernel.
  - De eigenlijke beschrijving van het model bevindt zich in het uitvoerbestand, in de vorm van de definitie van de modelklasse-constructor. De naar HGPSS++ vertaalde, van het model deel uitmakende statements worden binnen deze constructor geplaatst.
- Naast de declaratie van de modelklasse komt in het header-bestand voor elk model ook de structuur terecht die alle in het model gebruikte labels als velden bevat.

De aanwending van een dergelijk header-bestand heeft drie voordelen.

- Ten eerste is het header-bestand erg nuttig bij het gebruik van voorgecompileerde modellen. Als in een programma gebruik gemaakt wordt van modellen die eerder naar object-code gecompileerd werden, is zowel een prototype van het model nodig als informatie omtrent de in het model gebruikte labels. Aangezien dit de informatie is die in het header-bestand opgenomen werd, kan in een programma gebruik gemaakt worden van een afzonderlijk gecompileerd model door het header-bestand horende bij dat model te includeren in het programma. Het gebruik van bibliotheken bestaande uit voorgecompileerde modellen wordt dan ook sterk vereenvoudigd. Het is voldoende om de header-bestanden van alle in de bibliotheek residerende modellen samen te brengen in één bestand en dit bestand te includeren als gebruik gemaakt wordt van de bibliotheek.

- Een tweede voordeel situeert zich in de context van de precompiler zelf. Het gebruik van een afzonderlijk bestand waarin de labels en de eraan geassocieerde identificatienummers worden opgenomen laat toe om de precompilatie in één doorgang te laten plaatsvinden. In principe zou een eerste doorgang vereist zijn om aan alle optredende labels een identificatienummer te associëren en een tweede om de eigenlijke uitvoercode te genereren, waarin van de eerder afgeleide identificatienummers gebruik gemaakt wordt. Door tijdens de eerste en enige doorgang labels en geassocieerde identificatienummers weg te schrijven naar een afzonderlijk bestand en dat bestand te includeren in het uitvoerbestand, vervalt de nood aan een tweede doorgang.

- De label-informatie opgenomen in het header-bestand is ook interessant bij het analyseren van de uitvoer van een simulatie. In deze uitvoer worden de entiteiten uitsluitend aangeduid door hun identificatienummer. Om deze entiteiten in verband te kunnen brengen met de in het HGPSS-programma door middel van een karaktersliert aangeduide entiteiten, is de *cross-reference* informatie uit het header-bestand noodzakelijk.

Het variable-bestand

Zoals eerder vermeld wordt het variable-bestand gebruikt om C++-functies in onder te brengen die gegenereerd werden bij de vertaling van een variable-, boolean variable- of fullword variable-entiteitsdeclaratie. Telkens een dergelijke entiteitsdeclaratie ontmoet wordt, waarin de variable bovendien gedefinieerd is door een arithmetische of booleaanse expressie, wordt deze expressie vertaald naar een C++-functie. De naam van de functie wordt afgeleid uit een teller. Deze teller wordt bij het begin van de verwerking van het invoerbestand op één geplaatst om daarna bij elke generatie van een C++-functie met één te worden geïncrementeerd. In het geval van een variable of fullword variable zal de gegenereerde C++-functie er als volgt uitzien:

```
static value_type Variable<number>(void)
   {
     ...
   }
```

waarbij `number` vervangen wordt door de actuele waarde van de teller. Een vertaalde boolean variable ziet er dan weer als volgt uit:

```
static boolean_type Variable<number>(void)
   {
     ...
   }
```

Het gebruik van een afzonderlijk variable-bestand tijdens de precompilatie vereenvoudigt deze precompilatie sterk. Er is echter geen echt voordeel verbonden aan het behouden van een afzonderlijk variable-bestand ná de precompilatie. Indien gewenst kan het variable-bestand dan ook fysisch opgenomen worden in het uitvoerbestand.

Eliminatie van conflictsituaties

De grammatica van de HGPSS-taal zoals die aanvankelijk werd opgesteld vertoonde twee conflicten of *ambiguïteiten.* In de grammatica gebruikt bij de constructie van de precompiler werden deze conflicten geëlimineerd door de grammatica lichtjes aan te passen.

- Het eerste conflict situeerde zich bij de declaratie van variable-, boolean variable- en fullword variable-entiteiten. In HGPSS kunnen deze op twee manieren gedefinieerd worden: door een expressie of door een C++-functie. In bepaalde gevallen kan de naam van een C++-functie niet worden onderscheiden van een expressie. Om hieraan een oplossing te bieden werd vooropgesteld dat de naam van een C++-functie voorafgegaan moet worden door een `/`-karakter. Aangezien ook functie-entiteiten door een C++-functie kunnen gedefinieerd worden, werd het gebruik van een `/`-karakter ook in dit geval verplicht gesteld, om de syntax min of meer consistent te houden.

- Het tweede conflict werd in de grammatica geïntroduceerd door de invoering van de vrijere codeervorm. Aangezien de commentaar op het einde van een statement niet meer op een vaste positie moet aanvangen, treedt er een probleem op bij statements waarvan alle parameters optioneel zijn. Er kan niet ondubbelzinnig uitgemaakt worden of de karakterstroom gescheiden van het sleutelwoord door white space, moet geïnterpreteerd worden als parameter-lijst of als commentaar. Om dit probleem te elimineren werd het verplicht gebruik ingevoerd van een `/`-karakter als alle parameters weggelaten worden.

Parameters

De parameters die in HGPSS-statements gebruikt worden, kunnen heel wat verschillende vormen aannemen. Bovendien gebruiken alle statements niet hetzelfde type parameters. De meeste parameters worden in hun totaliteit herkend door de lexicale analysator en als karaktersliert doorgestuurd naar de parser. Omdat het analyseren en het vertalen naar HGPSS++ van de meeste parameters vrij ingewikkeld is, werden voor deze taak drie afzonderlijke compilers geconstrueerd die ondergeschikt zijn aan de eigenlijke HGPSS-precompiler. Elke mini-compiler behandelt een verschillend type parameters. De minicompilers worden door de precompiler voorzien van karakterslierten die de parameters voorstellen. De parameters worden dan vertaald en afgedrukt in het uitvoer- of variable-bestand, naargelang gespecifieerd door de precompiler. De mini-compilers worden respectievelijk aangesproken door de functies `PrintParameter`, `PrintInitial` en `PrintLogical`. Alle mini-compilers werden beschreven in YACC en maken gebruik van een eigen functie `yylex` en `yyparse`. Deze functies bevinden zich in afzonderlijke modules en zijn afgeschermd van de andere modules. De functies `yylex` werden niet gegenereerd door gebruik te maken van LEX maar werden rechtstreeks in C geprogrammeerd, aangezien ze uiterst eenvoudig zijn.

7.3.4 Voorbeeld

Ter illustratie van de wijze waarop een HGPSS-programma naar een HGPSS++-programma wordt omgezet, werd hieronder een HGPSS-programma opgenomen evenals de door de precompiler gegenereerde HGPSS++-uitvoer. Als voorbeeld werd het via enkele kleine ingrepen naar HGPSS omgezette GPSS-programma

genomen uit [Gordon 1978], Figuur 12-4. In Figuur 7.2 is het invoerbestand opgenomen, in Figuur 7.3 en Figuur 7.4 het uitvoerbestand, in Figuur 7.5 het header-bestand en in Figuur 7.6 het variable-bestand. Bij het vergelijken van de HGPSS- en HGPSS++-versie van het programma valt op dat beide versies sterke gelijkenissen vertonen. Voorts blijft het HGPSS++-programma ook vrij goed leesbaar. In het voorbeeld is duidelijk te zien hoe de HGPSS-blokdeclaraties, -entiteitsdeclaraties, -commando's en -directives behandeld werden evenals de commentaar en de labels. In het bijzonder kan eenvoudig worden nagegaan hoe HGPSS-functies en -variables behandeld worden. Bovenaan het uitvoerbestand wordt naast het header- en het variable-bestand ook nog een derde bestand geïncludeerd. Dit bestand bevat een aantal definities en declaraties die noodzakelijk zijn om de HGPSS++-kernel te kunnen aanspreken.

```
*Project: HGPSS: Object-oriented process-interaction simulation

*Origin: System simulation, Figure 12-4
*Author: G. Gordon
*Description: Simulation of a supermarket


MODEL Gordon6

*
*        SIMULATION OF A SUPERMARKET
*
1        FUNCTION            RN1,C24             Function for i/a interval
         0.0,0.0/0.1,0.104/0.2,0.222/0.3,0.355/0.4,0.509/0.5,0.69-
         0.6,0.915/0.7,1.2/0.75,1.38/0.8,1.6/0.84,1.83/0.88,7.12-
         0.9,2.3/0.92,2.52/0.94,2.81/0.95,2.99/0.96,3.2/0.97,3.5-
         0.98,3.9/0.99,4.6/0.995,5.3/0.998,6.2/0.999,7/0.9997,8
*
         GENERATE            36,FN1,,,,1         Create shoppers with 1 parameter
         TRANSFER            BOTH,,AWAY          Check for available basket
         ENTER               BSKT                Get a basket
         ASSIGN              1,FN2               Determine no. of items
         ADVANCE             1,FN3               Shop
         QUEUE               WAIT                Wait for counter space
         ENTER               CKT                 Get counter space
         DEPART              WAIT                Leave queue
         ADVANCE             V1                  Check-out
         LEAVE               CKT                 Free counter space
         TABULATE            TRT                 Tabulate transit time
         TABULATE            ITM                 Tabulate no. of items
         LEAVE               BSKT                Return basket
         TERMINATE           1
*
AWAY     TERMINATE           /                   Lost customers
*
TRT      TABLE               M1,500,500,10       Transit time table
ITM      TABLE               P1,5,5,4            item count table
*
CKT      STORAGE             5                   Number of counters
BSKT     STORAGE             50                  Number of baskets
*
2        FUNCTION            RN1,D4              Distr. of no. of items
         .2,5/.5,.10/.9,15/1.0,20
*
3        FUNCTION            P1,C5               Shopping time distr.
         0,0/5,400/10,900/15,1500/20,2250
*
1        VARIABLE            P1*10+25            Check-out time

ENDMODEL
COMMAND

         SIMULATE            Gordon6             Simulate model
         START               50                  Initialise
         RESET                                   Wipe out statistics
         START               1000                Main run
         PRINT               /                   Print all information
         END                                     Remove model

ENDCOMMAND
```

Figure 7.2: Voorbeeld van een invoerbestand voor de HGPSS-precompiler

```
/***************************************************************************

   Description: HGPSS++ source file generated from HGPSS source file
   Name: gordon6.C
   Generated from: gordon6.gps
   Date: Sat Apr 25 17:48:57 1992

***************************************************************************/


#include "hgpsspp.h"
#include "gordon6.h"
#include "gordon6.var"

/* Project: HGPSS: Object-oriented process-interaction simulation */

/* Origin: System simulation, Figure 12-4 */
/* Author: G. Gordon */
/* Description: Simulation of a supermarket */


Gordon6Class::Gordon6Class(void)
   {
     StartConstruct();

/*  */
/*         SIMULATION OF A SUPERMARKET */
/*  */
     FUNCTION  (1,P(PARAMETER_SNA_DIRECT,1,SNA_RN),FUNCTION_C,24,   /* Function for i/a interval */
               (value_type) 0.0,(value_type) 0.0,
               (value_type) 0.1,(value_type) 0.104,
               (value_type) 0.2,(value_type) 0.222,
               (value_type) 0.3,(value_type) 0.355,
               (value_type) 0.4,(value_type) 0.509,
               (value_type) 0.5,(value_type) 0.69,
               (value_type) 0.6,(value_type) 0.915,
               (value_type) 0.7,(value_type) 1.2,
               (value_type) 0.75,(value_type) 1.38,
               (value_type) 0.8,(value_type) 1.6,
               (value_type) 0.84,(value_type) 1.83,
               (value_type) 0.88,(value_type) 7.12,
               (value_type) 0.9,(value_type) 2.3,
               (value_type) 0.92,(value_type) 2.52,
               (value_type) 0.94,(value_type) 2.81,
               (value_type) 0.95,(value_type) 2.99,
               (value_type) 0.96,(value_type) 3.2,
               (value_type) 0.97,(value_type) 3.5,
               (value_type) 0.98,(value_type) 3.9,
               (value_type) 0.99,(value_type) 4.6,
               (value_type) 0.995,(value_type) 5.3,
               (value_type) 0.998,(value_type) 6.2,
               (value_type) 0.999,(value_type) 7,
               (value_type) 0.9997,(value_type) 8);
/*  */
     GENERATE  (1,2,P(PARAMETER_VALUE,36),P(PARAMETER_SNA_DIRECT,1,SNA_FN),P(),P(),P(),P(PARAMETER_ ...
     TRANSFER  (2,3,P(PARAMETER_BOTH),P(),P(PARAMETER_VALUE,Gordon6Struct.AWAY));   /* Check for ... */
     ENTER     (3,4,P(PARAMETER_VALUE,Gordon6Struct.BSKT));   /* Get a basket */
     ASSIGN    (4,5,P(PARAMETER_VALUE,1),OPERATION_NONE,P(PARAMETER_SNA_DIRECT,2,SNA_FN));   /* ... */
     ADVANCE   (5,6,P(PARAMETER_VALUE,1),P(PARAMETER_SNA_DIRECT,3,SNA_FN));   /* Shop */
     QUEUE     (6,7,P(PARAMETER_VALUE,Gordon6Struct.WAIT));   /* Wait for counter space */
     ENTER     (7,8,P(PARAMETER_VALUE,Gordon6Struct.CKT));   /* Get counter space */
     DEPART    (8,9,P(PARAMETER_VALUE,Gordon6Struct.WAIT));   /* Leave queue */
```

Figure 7.3: Voorbeeld van een door de HGPSS-precompiler genereerd uitvoerbestand

```
     ADVANCE   (9,10,P(PARAMETER_SNA_DIRECT,1,SNA_V));   /* Check-out */
     LEAVE (10,11,P(PARAMETER_VALUE,Gordon6Struct.CKT)); /* Free counter space */
     TABULATE  (11,12,P(PARAMETER_VALUE,Gordon6Struct.TRT));   /* Tabulate transit time */
     TABULATE  (12,13,P(PARAMETER_VALUE,Gordon6Struct.ITM));   /* Tabulate no. of items */
     LEAVE     (13,14,P(PARAMETER_VALUE,Gordon6Struct.BSKT));   /* Return basket */
     TERMINATE (14,15,P(PARAMETER_VALUE,1));
/*  */
     TERMINATE (Gordon6Struct.AWAY,16,P());   /* Lost customers */
/*  */
     TABLE     (Gordon6Struct.TRT,P(PARAMETER_SNA_DIRECT,VALUE_NONE,SNA_M1),500,500,10);   /* ... */
     TABLE     (Gordon6Struct.ITM,P(PARAMETER_SNA_DIRECT,1,SNA_P),5,5,4);   /* item count table */
/*  */
     STORAGE   (Gordon6Struct.CKT,5);   /* Number of counters */
     STORAGE   (Gordon6Struct.BSKT,50);   /* Number of baskets */
/*  */
     FUNCTION  (2,P(PARAMETER_SNA_DIRECT,1,SNA_RN),FUNCTION_D,4,   /* Distr. of no. of items */
               (value_type) .2,(value_type) 5,
               (value_type) .5,(value_type) .10,
               (value_type) .9,(value_type) 15,
               (value_type) 1.0,(value_type) 20);
/*  */
     FUNCTION  (3,P(PARAMETER_SNA_DIRECT,1,SNA_P),FUNCTION_C,5,   /* Shopping time distr. */
               (value_type) 0,(value_type) 0,
               (value_type) 5,(value_type) 400,
               (value_type) 10,(value_type) 900,
               (value_type) 15,(value_type) 1500,
               (value_type) 20,(value_type) 2250);
/*  */
     VARIABLE  (1,Variable1);   /* Check-out time */

     EndConstruct();
   }

void main(void)
   {

     SIMULATE        (new Gordon6Class);   /* Simulate model */
     START           (50);   /* Initialise */
     RESET           ();   /* Wipe out statistics */
     START           (1000);   /* Main run */
     PRINT           (P());   /* Print all information */
     END             ();   /* Remove model */

   }


/*************************************************************************/
```

Figure 7.4: Voorbeeld van een door de HGPSS-precompiler gegenereerd uitvoerbestand (vervolg)

```
/*****************************************************************************

   Description: HGPSS++ header file generated from HGPSS source file
   Name: gordon6.h
   Generated from: gordon6.gps
   Date: Sat Apr 25 17:48:57 1992

*****************************************************************************/


class Gordon6Class : public ModelClass
   {
     public:
      Gordon6Class(void);
   };

static const struct
   {
     number_type AWAY;
     number_type BSKT;
     number_type WAIT;
     number_type CKT;
     number_type TRT;
     number_type ITM;
   } Gordon6Struct =

   {
     15,
     10003,
     10004,
     10002,
     10000,
     10001
   };


/*****************************************************************************/
```

Figure 7.5: Voorbeeld van een door de HGPSS-precompiler gegenereerd header-bestand

```
/*****************************************************************************

   Description: HGPSS++ variable file generated from HGPSS source file
   Name: gordon6.var
   Generated from: gordon6.gps
   Date: Sat Apr 25 17:48:57 1992

*****************************************************************************/


static value_type Variable1(void)
   {
     return( EVALUATE( P(PARAMETER_SNA_DIRECT,1,SNA_P) ) * 10 + 25 );
   }


/*****************************************************************************/
```

Figure 7.6: Voorbeeld van een door de HGPSS-precompiler gegenereerd variable-bestand

Chapter 8

Toepassing: logische simulatie

8.1 Inleiding

In dit hoofdstuk wordt door middel van een voorbeeld aangetoond hoe HGPSS kan ingezet worden bij de simulatie van systemen die inherent een sterk modulaire structuur bezitten, namelijk digitale elektronische netwerken. De bedoeling is om *logische* simulatie uit te voeren. Dit is de simulatie van een schakeling in het tijdsdomein op poortniveau. Hierbij wordt abstractie gemaakt van fysische grootheden als spanningen en stromen. Enkel het logische gedrag is van belang. Met een dergelijke simulatie kunnen meerdere objectieven nagestreefd worden.

- Het doel kan *functionele verificatie* zijn, waarbij wordt nagegaan of het netwerk de correcte vooropgestelde functionaliteit bezit.
- Bij een *timing-verificatie* wordt het netwerk onderzocht op eventuele pathologische timing-problemen als statische of dynamische hazards.
- Het doel kan ook de *analyse van de initialisatie* zijn. Hierbij wordt nagegaan of het netwerk bij het opstarten in een welbepaalde toestand komt en welke deze toestand is.
- Uiteindelijk kan het doel ook *foutsimulatie* zijn. Hierbij kan men geïnteresseerd zijn in de foutdekking of in het gedrag van het netwerk in de aanwezigheid van een fout.

Een digitaal netwerk is opgebouwd uit componenten. Deze componenten zijn instanties van types. Bij elk type hoort een bepaald functioneel- en timing-gedrag. Binnen een component voltrekken zich een aantal processen. Een netwerk kan voorts ook veelal opgedeeld worden in een aantal groepen sterk samenhangende componenten. Tussen de groepen onderling heerst een minder sterke binding. Deze groepen samenhangende componenten vormen deelnetwerken. Een netwerk is via een boomvormige hiërarchie uit deelnetwerken opgebouwd.

Vanuit het oogpunt van simulatie kunnen er twee niveaus van modulariteit worden onderscheiden: het component- en het deelnetwerk-niveau. Binnen HGPSS zullen zowel de componenten als de deelnetwerken door een submodel beschreven moeten worden. Bij de beschrijving van de componenten zal gebruik gemaakt worden van gewone HGPSS-blokken terwijl een deelnetwerk zal opgebouwd worden uit de interconnectie van een aantal dergelijke submodellen.

De belangrijkste fase in de ontwikkeling van een simulator voor logische simulatie van een netwerk in HGPSS, is het construeren van de met de gebruikte componenten overeenstemmende submodellen. Eens deze submodellen voorhanden zijn, kan het model voor het netwerk op een relatief eenvoudige wijze opgebouwd worden. Een component kan beschouwd worden als de combinatie van een aantal logische,

vertagingsloze operatoren en vertragingselementen. De wijze waarop de vertraging gemodelleerd wordt, kan verscheiden zijn. Een aantal veelgebruikte *vertragingsmodellen* zijn:

- *Nulvertraging*: Er wordt geen vertraging geïntroduceerd.
- *Eenheidsvertraging*: Alle componenten introduceren dezelfde vertraging.
- *Transportvertraging*: Elke component introduceert een vertraging eigen aan het type waartoe de component behoort.
- *Stijg- en valvertraging*: Een verschillende vertraging wordt geïntroduceerd naargelang de aard van de verandering van de uitgang van de component.
- *Ambigue vertraging*: De door de componenten geïntroduceerde vertraging is met een bepaalde probabiliteit gedistribueerd tussen een minimum- en maximum-waarde.
- *Belastingsvertaging*: De vertraging is niet enkel afhankelijk van de component zelf maar ook van de belasting aan de uitgang van deze component. De totale vertraging volgt uit de formule

$$\Delta_{totaal} = \Delta_0 + (C_{ingang} + C_{bedrading}) * \Delta_c$$

  waarbij
  Δ_{totaal} : totale vertraging
  Δ_0 : intrinsieke, door de component geïntroduceerde vertraging
  C_{ingang} : ingangscapaciteit van de aan de uitgang van de component in kwestie verbonden componenten
  $C_{bedrading}$: capaciteit van de bedrading
  Δ_c : vertraging geïntroduceerd per eenheid van capaciteit

- *Inertievertraging*: Bij gebruik van dit model zal een verandering aan de ingang van een poort slechts effect hebben als de tijdspanne gedurende dewelke deze verandering optreedt groot genoeg is.

Meestal wordt een combinatie van enkele van de geschetste vertragingsmodellen aangewend, of van varianten hierop.

Informatie wordt doorheen een netwerk getransporteerd door signalen die zich voorplanten over de verbindingen tussen de componenten. Deze verbindingen worden *netten* genoemd. In de context van logische simulatie wordt de informatie vervat in de signalen verondersteld deel uit te maken van een beperkte verzameling mogelijke waarden. Meestal worden slechts twee mogelijke toestanden gebruikt, voorgesteld door `0` en `1`. Deze toestanden zullen in werkelijkheid geassocieerd zijn aan een bepaald spanningsniveau, maar dit is bij logische simulatie irrelevant. Een logische simulator die echter enkel gebruik maakt van `0` en `1` heeft weinig nut. Minstens moet nog een toestand, aangeduid door bijvoorbeeld `X`, ingevoerd worden die aangeeft dat er onzekerheid heerst omtrent de ware toestand van een net[1]. Een `X` zal er met andere woorden op wijzen dat niet met zekerheid kan besloten worden tot een `0` of `1`. De reden voor deze onzekerheid kan verschillende oorzaken hebben. Zo zal bij het opstarten van een netwerk de toestand van de interne netten niet gekend zijn. Als de toestand van de signalen elementen zijn uit de verzameling `{X,0,1}` moeten de waarheidstabellen van de logische bewerkingen uitgebreid worden om met elementen uit deze uitgebreide verzameling te kunnen werken. Voor de bewerkingen `NOT`, `AND`, `OR` en `EXOR` wordt dit geïllustreerd in Tabel 8.1, 8.2, 8.3 en 8.4.

[1] De invoering van `X` werd voorgesteld door Eichelberger.

NOT	X	0	1
	X	1	0

Table 8.1: NOT-bewerking op {X,0,1}

AND	X	0	1
X	X	0	X
0	0	0	0
1	X	0	1

Table 8.2: AND-bewerking op {X,0,1}

8.2 Voorbeeld van modellering: AND-poort

Om aan te tonen hoe een digitale component voor logische simulatie kan gemodelleerd worden in HGPSS, wordt een AND-poort beschouwd. Een dergelijke poort is voorgesteld in Figuur 8.1. Als vertragingsmodel werd de transportvertraging gekozen. Door de manier waarop deze geïmplementeerd is, wordt echter ook een vorm van inertievertraging ingevoerd.

Aangezien transacties de entiteiten zijn die informatie doorheen een model transporteren, moeten de signalen die zich over de netten voortplanten, door transacties worden voorgesteld. Een transactieparameter kan gebruikt worden om de signaaltoestand aan te duiden. Om de signaaltoestanden eenvoudig te kunnen voorstellen kan hiervoor een specifiek C++-type gecreëerd worden zoals in Figuur 8.2.

Voor de implementatie van de AND-bewerking kan een C++-tabel worden aangemaakt waarin de uitgebreide waarheidstabel vervat zit, zoals in Figuur 8.3. Eventueel kan ook een functie gebruikt worden die toegang verleent tot de tabel.

In de eigenlijke HGPSS-beschrijving kan gebruik gemaakt worden van savevalues om de huidige toestand vast te houden van de netten verbonden met de ingangen en de uitgang van een poort. Als een transactie zich aandient aan een input van het model, wordt de inhoud van de parameter die de signaaltoestand aangeeft dan gekopieerd naar de met de input geassocieerde savevalue. De transactie wordt vervolgens op non-actief geplaatst gedurende een tijdspanne overeenstemmende met de transportvertraging. Na verloop van deze periode wordt de AND-bewerking uitgevoerd op de signaaltoestanden opgeslagen in de savevalues corresponderende met de inputs. Als het resultaat van deze bewerking verschillend is van de huidige toestand van het net verbonden met de uitgang, wordt de transactie die de nieuwe waarde van het uitgangsnet bevat, naar de output gestuurd. Als het resultaat van de bewerking gelijk is aan de huidige toestand van het uitgangsnet, is het nutteloos om een transactie een toestandsverandering te laten aangeven. Dit kan enkel de performantie van de resulterende simulator sterk negatief beïnvloeden. In Figuur 8.4 is de HGPSS-beschrijving van een mogelijk model van een AND-poort opgenomen. Het model wordt in Figuur 8.5 grafisch voorgesteld. De waarde van de door de poort geïntroduceerde vertraging moet via een parameter opgegeven worden bij instantiëring van het model. Eventueel kan aan deze parameter een waarde bij verstek worden toegekend. Andere types poorten zoals NAND, OR, NOR,

OR	X	0	1
X	X	X	1
0	X	0	1
1	1	1	1

Table 8.3: OR-bewerking op {X,0,1}

EXOR	X	0	1
X	X	X	X
0	X	0	1
1	X	1	0

Table 8.4: EXOR-bewerking op $\{X,0,1\}$

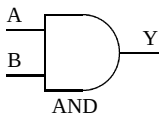

Figure 8.1: Symbool AND-poort

EXOR en NEXOR evenals buffers en invertoren, kunnen op een analoge manier beschreven worden.

Ter vergelijking werd in Figuur 8.6 de *HHDL*-beschrijving van een AND-poort opgenomen. HHDL staat voor *Hierarchical Hardware Description Language* en is een op PASCAL gebaseerde taal, specifiek voor de high-level beschrijving van digitale elektronische componenten. De taal vertoont sterke gelijkenissen met *VHDL* en wordt gebruikt binnen het *SL2010* CAEE[2]-pakket van de firma *Silvar Lisco*. Lijnen (1)-(17) zijn enkele eenvoudige PASCAL-statements die volkomen analoog zijn aan Figuur 8.2 en 8.3. Lijnen (19)-(34) vormen de eigenlijke beschrijving. Opvallend is hierbij dat de hele functionaliteit van de poort op één lijn kan worden neergeschreven (lijn (31)), terwijl dit in HGPSS meerdere lijnen in beslag neemt. Aangezien echter het HHDL-systeem specifiek voor logische simulatie werd ontwikkeld, zitten de voorzieningen voor vertraging in het systeem zelf ingebakken, zodat hieraan geen deel van de beschrijving moet worden gewijd.

8.3 Opbouwen van een netwerk

Een volledig netwerk kan gemodelleerd worden door naast de componenten, ook de wijze waarop de componenten verbonden zijn, of de *connectiviteit*, te beschrijven. Bij deze beschrijving kan eventueel gebruik gemaakt worden van een hiërarchie van deelmodellen. Het beschrijven van de connectiviteit van de componenten is in HGPSS niet moeilijk maar wel vrij omslachtig. De reden hiervoor ligt in het feit dat de vorm van een HGPSS-programma eigenlijk gericht is op het beschrijven van een eenvoudige rechtlijnige transactiestroom. Op plaatsen waar een transactiestroom zich splitst of meerdere stromen samenkomen, moeten speciale voorzieningen getroffen worden. In een netwerk is echter over het algemeen eenzelfde knoop, of meer specifiek eenzelfde net, met meerdere uitgangen en ingangen verbonden, zodat de stroom doorheen het systeem ver van rechtlijnig is. Beschouw bijvoorbeeld een algemeen geval van een net verbonden met *m* uitgangen en *n* ingangen, zoals voorgesteld in Figuur 8.7. Een dergelijke situatie moet in HGPSS beschreven worden als in Figuur 8.8 of via een grafische voorstelling als in Figuur 8.9.

Aangezien de situatie bij elke knoop analoog zal zijn, kan eraan gedacht worden om een knoop ook als submodel te beschouwen. Een knoop kan als geparameteriseerd model worden opgevat waarbij één van de parameters het aantal uitgangen aangeeft waarmee de knoop verbonden is, en een andere parameter het aantal ingangen. Voor elke uitgang wordt binnen het model een input voorzien en voor elke ingang een output. Een dergelijk model kan moeilijk exclusief in HGPSS beschreven worden, in

[2]Computer Aided Electronic Engineering.

```
(1) enum level_enum {LOGIC_X,
(2)                  LOGIC_0,
(3)                  LOGIC_1};
(4)
(5) typedef level_enum level_type;
```

Figure 8.2: C++-type voor voorstelling signaaltoestanden

```
(1) #define NBR_OF_LEVELS                 3
(2)
(3) const level_type ANDTable[NBR_OF_LEVELS][NBR_OF_LEVELS]=
(4)   { { LOGIC_X,LOGIC_0,LOGIC_X },
(5)     { LOGIC_0,LOGIC_0,LOGIC_0 },
(6)     { LOGIC_X,LOGIC_0,LOGIC_1 } };
(7)
(8) level_type LOGIC_AND(level_type _in1,level_type _in2)
(9)   {
(10)    return(ANDTable[_in1][_in2]);
(11)  }
```

Figure 8.3: C++-tabel ter implementatie van de AND-bewerking

```
(1) -static value_type A_B_to_Y(void)
(2) -   {
(3) -     return( LOGIC_AND( (level_type) EVALUATE( P(PARAMETER_SNA_DIRECT,AND2Struct.S_A,SNA_X) ),
(4) -                        (level_type) EVALUATE( P(PARAMETER_SNA_DIRECT,AND2Struct.S_B,SNA_X) ) ) );
(5) -   }
(6)
(7) MODEL AND2(value_type _delay_A_B_to_Y=8)
(8)
(9)          INPUT               A
(10)         SAVEVALUE           S_A,P1
(11)         TRANSFER            ,N_1
(12)
(13)         INPUT               B
(14)         SAVEVALUE           S_B,P1
(15)
(16) N_1     ADVANCE             "_delay_A_B_to_Y"
(17)         ASSIGN              1,V1
(18)         TEST NE             S_Y,P1,N_2
(19)         SAVEVALUE           S_Y,P1
(20)         OUTPUT              Y
(21)
(22) N_2     TERMINATE           /
(23)
(24) 1       VARIABLE            /"A_B_to_Y"
(25)
(26) ENDMODEL
```

Figure 8.4: HGPSS-model AND-poort

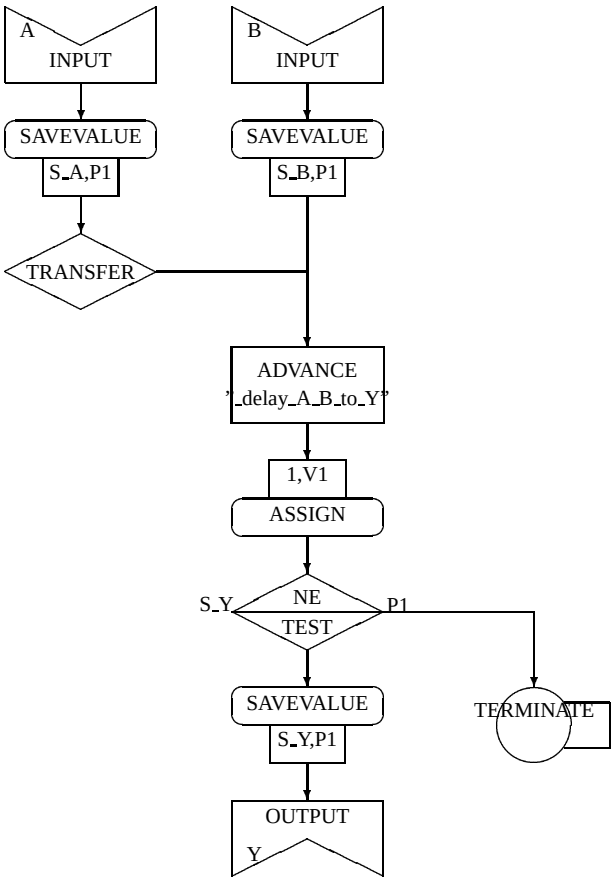

Figure 8.5: Grafische voorstelling HGPSS-model AND-poort

```
 (1) TYPE
 (2)
 (3)   level_type                          = (LOGIC_X,LOGIC_0,LOGIC_1);
 (4)
 (5) CONST
 (6)
 (7)   ANDTable                            : ARRAY[LOGIC_X..LOGIC_1,
 (8)                                                LOGIC_X..LOGIC_1]
 (9)                                         of level_type =
(10)                                         ( ( LOGIC_X,LOGIC_0,LOGIC_X ),
(11)                                           ( LOGIC_0,LOGIC_0,LOGIC_0 ),
(12)                                           ( LOGIC_X,LOGIC_0,LOGIC_1 ) );
(13)
(14) FUNCTION LOGIC_AND(_in1,_in2 : level_type) : level_type;
(15)    BEGIN
(16)       LOGIC_AND:=ANDTable[_in1,_in2]
(17)    END;
(18)
(19) COMPTYPE AND2(_delay_A_B_to_Y : REAL);
(20)
(21)    DEFAULT _delay_A_B_to_Y             = 8;
(22)
(23)    INWARD A,B                          : level_type;
(24)
(25)    OUTWARD Y                           : level_type;
(26)
(27)    SUBPROCESS
(28)
(29)       A_B_to_Y:
(30)
(31)          TRANSMIT LOGIC_AND[A,B] CHECK A,B TO Y DELAY _delay_A_B_to_Y;
(32)
(33)    BEGIN
(34)    END;
```

Figure 8.6: HHDL-model AND-poort

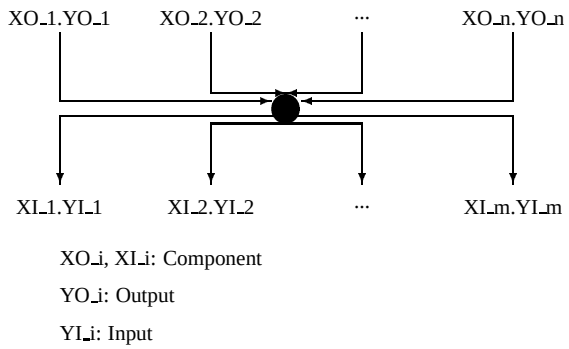

Figure 8.7: Algemene voorstelling net

```
         LEAVEMODEL          XO_1,YO_1
         TRANSFER            ,Node_1
         LEAVEMODEL          XO_2,YO_2
         TRANSFER            ,Node_1
         ...
         LEAVEMODEL          XO_m,YO_m
Node_1   SPLIT               1,Node_2
         ENTERMODEL          XI_1,YI_1
Node_2   SPLIT               1,Node_3
         ENTERMODEL          XI_2,YI_2
         ...
Node_n   ENTERMODEL          XI_n,YI_n
```

Figure 8.8: HGPSS-beschrijving connectiviteit

combinatie met HGPSS++ is dit eenvoudiger zoals geïllustreerd in Figuur 8.10 en 8.11. De beschrijving van de interconnecties uit Figuur 8.8 reduceert zich dan tot de beschrijving van Figuur 8.12. Alhoewel het aantal benodigde blokken in deze beschrijving ongeveer gelijk is als in Figuur 8.8 is de structuur toch eenvoudiger, wat een eventuele automatische vertaling van een *netlist* naar een HGPSS-programma zou toelaten.

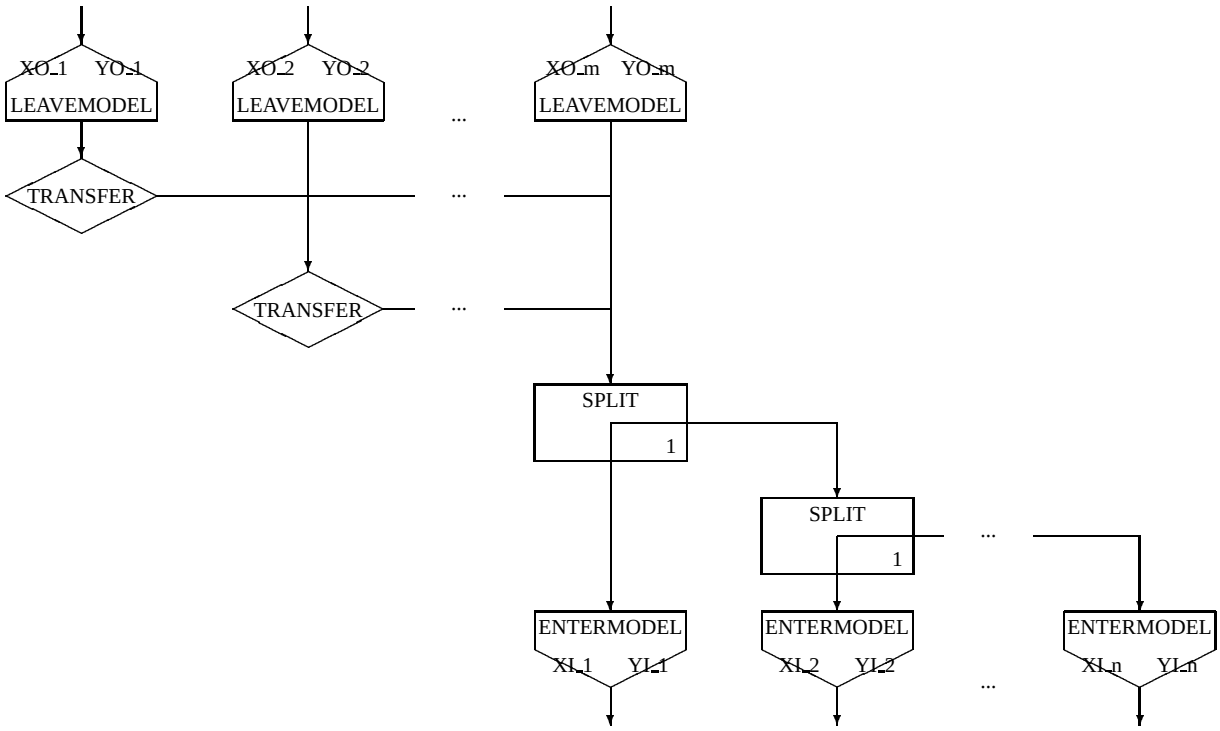

Figure 8.9: Grafische voorstelling HGPSS-beschrijving connectiviteit

```
MODEL Net(count_type _nbr_of_inputs,count_type _nbr_of_outputs)
%{
         for (number_type input_nbr=1; input_nbr<=_nbr_of_inputs; input_nbr++)
            INPUT(input_nbr,_nbr_of_inputs+1,input_nbr);
         for (number_type output_nbr=1; output_nbr<=_nbr_of_outputs-1; output_nbr++)
            {
              SPLIT(_nbr_of_inputs+output_nbr*2-1,_nbr_of_inputs+output_nbr*2,
                    P(PARAMETER_VALUE,1),P(PARAMETER_VALUE,_nbr_of_inputs+output_nbr*2+1));
              OUTPUT(_nbr_of_inputs+output_nbr*2,NUMBER_NONE,output_nbr);
            }
         OUTPUT(_nbr_of_inputs+_nbr_of_outputs*2-1,NUMBER_NONE,_nbr_of_outputs);
%}
ENDMODEL
```

Figure 8.10: HGPSS-beschrijving net-submodel

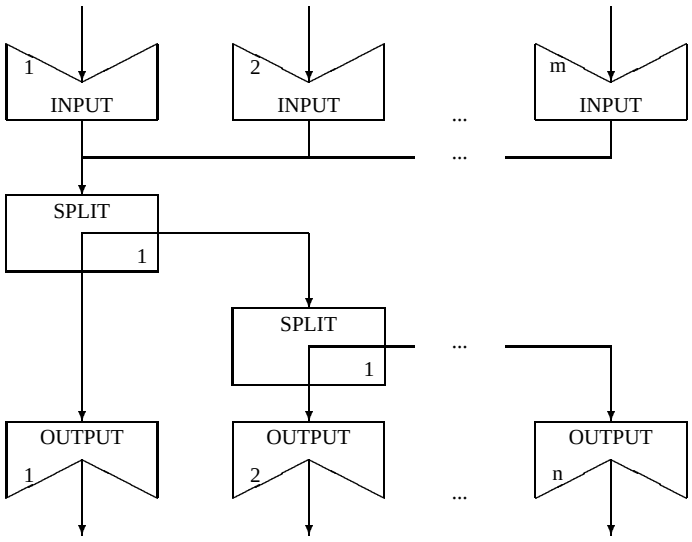

Figure 8.11: Grafische voorstelling HGPSS-beschrijving net-submodel

```
Node     SUBMODEL Net(m,n)

         LEAVEMODEL          XI_1,YI_1
         ENTERMODEL          Node,1
         LEAVEMODEL          XI_2,YI_2
         ENTERMODEL          Node,2
         ...
         LEAVEMODEL          XI_m,YI_m
         ENTERMODEL          Node,m
         LEAVEMODEL          Node,1
         ENTERMODEL          XU_1,YU_1
         LEAVEMODEL          Node,2
         ENTERMODEL          XU_2,YU_2
         ...
         LEAVEMODEL          Node,n
         ENTERMODEL          XU_n,YU_n
```

Figure 8.12: HGPSS-beschrijving connectiviteit gebruik makend van net-submodel

Chapter 9

Besluit

9.1 Evaluatie

9.1.1 Voor- en nadelen

Het belangrijkste item bij de evaluatie van het HGPSS-systeem is de mate waarin het systeem tegemoet komt aan de in Hoofdstuk 1 gestelde doelstellingen.

- De eerste doelstelling, het bieden van een mogelijkheid tot koppeling met een continue simulatie, werd verwezenlijkt door het toevoegen van twee additionele blokken aan de GPSS-taal: het INTERN- en het EXTERN-blok.
- De INTERN- en EXTERN-blokken dragen ook bij tot de realisatie van de tweede doelstelling: het leveren van mogelijkheden tot koppeling met uitwendige software. De mogelijkheden tot inbedding van gasttaal-statements in een HGPSS-programma en het gebruik van externe software-componenten als HGPSS-entiteiten, dragen eveneens bij tot het realiseren van deze doelstelling. Eigenlijk zit de eerste doelstelling vervat in de tweede doelstelling. Een continue simulatie kan ook eenvoudig als een externe software-component beschouwd worden. Omwille van het belang van de koppeling met continue simulatie werd deze doelstelling echter apart beschouwd.
- De derde doelstelling, het introduceren van object-oriëntatie in zowel de implementatie van de ondersteunende kernel als in de modellering zelf, werd uitgewerkt door de kernel te implementeren in de object-georiënteerde programmeertaal C++ en door het invoeren van hiërarchisch modelleren als uitbreiding op GPSS.
- De laatste doelstelling, het toelaten van interactie met de low-level aspecten van de simulatie, werd verwezenlijkt door de kernel rechtstreeks en op een aantal verschillende niveaus aanspreekbaar te maken. De kernel werd bovendien ontwikkeld als een open systeem dat steeds aanpasbaar en uitbreidbaar is, om aan veranderende eisen tegemoet te komen.

Naast de voordelen geïntroduceerd door de realisatie van de doelstellingen biedt het HGPSS-systeem nog een aantal positieve aspecten die reeds in GPSS aanwezig waren.

- Er wordt gebruik gemaakt van de process-interaction approach, die voor sterk interagerende systemen algemeen als de beste world view beschouwd wordt.
- HGPSS is een uitbreiding op GPSS, een sterk verspreide simulatie-taal met een groot volume reeds ontwikkelde software.

- HGPSS heeft als taal een high-level karakter. Krachtige constructies laten toe om een systeem op een eenvoudige en snelle manier te modelleren. HGPSS is zoals GPSS een volledige programmeertaal, waarin zowel sequentie, iteratie als selectie aanwezig zijn.

- Aangezien de processor tot nader order nog steeds alle low-level aspecten van de simulatie afhandelt, blijft het programmeergemak behouden.

- Gegenereerde data wordt automatisch gecollecteerd en uitgevoerd.

De meeste punten waarop GPSS zwak scoort werden geëlimineerd zodat deze bij de evaluatie van HGPSS niet meer kunnen aangehaald worden als nadelen.

- De erg strikte GPSS-codeervorm werd verlaten voor een vrijere vorm.

- De integer-based simulatie-klok werd vervangen door een real-based klok. Een integer-based klok kan echter indien gewenst nog steeds gebruikt worden door de herformulering van één enkele type-definitie in de broncode van de kernel.

- Buiten de processor worden alle objecten dynamisch gecreëerd. De statische geheugenallocatie behoort dus tot het verleden.

- Een gevolg van de indeling in model- en commandosecties, is de scheiding van model- en environmental frame.

- De manier waarop uitvoer gegenereerd wordt en de aard van de uitgevoerde informatie kan door de gebruiker worden gespecifieerd.

- De verzameling toegelaten identifiers is op een significante manier uitgebreid.

Een erg belangrijk voordeel van het HGPSS-systeem is de portabiliteit. Zowel de HGPSS-precompiler als de HGPSS++-kernel zijn beschikbaar voor MSDOS- en UNIX-systemen. In principe zijn beide software-componenten beschikbaar voor elk systeem dat beschikt over een C/C++-compiler. De C++-broncode van de kernel is immers volledig compatibel met de *AT&T C++ 2.0*-norm. De LEX- en YACC-broncode van de precompiler is mits enkele aanpassingen overdraagbaar tussen de verschillende bestaande versies van LEX en YACC. Indien LEX en YACC op een bepaald systeem niet voorhanden zijn, kan gepoogd worden om de broncode van de precompiler te compileren tot C-routines op een systeem waar LEX en YACC wel beschikbaar zijn en vervolgens deze C-routines te compileren en te linken op het target system.

Het HGPSS-systeem bezit in haar huidige vorm naast heel wat voordelen toch ook een aantal nadelen.

- Een eerste nadeel is inherent aan het gebruik van GPSS als basis voor de HGPSS-taal. GPSS maakt veelvuldig gebruik van moeilijk te memoreren mnemonics en de syntax bevat een aantal inconsistenties. Deze problemen hadden tijdens de ontwikkeling van HGPSS kunnen worden geëlimineerd, dit echter ten koste van de compatibiliteit met GPSS en de grote hoeveelheid reeds bestaande GPSS-software.

- Aangezien bij de ontwikkeling van de HGPSS++-kernel de aandacht op de eenvoud, duidelijkheid en doorzichtigheid werd toegespitst, bereikt het resulterende systeem niet de hoogst mogelijke performantie. Het performantieverlies ten opzichte van commerciële implementaties van GPSS is vrij aanzienlijk.

- Zowel bij de implementatie van de HGPSS-precompiler als de HGPSS++-kernel werd weinig aandacht besteed aan de afhandeling van fouten. In het geval van de precompiler wordt bij het constateren van een syntax-fout in het invoerbestand enkel gerapporteerd op welke regel de fout zich manifesteert, waarna de uitvoering van het programma wordt afgebroken. In het geval van de kernel wordt wel informatie afgedrukt over de aard van de fout vóór de beëindiging van de simulatie. Deze informatie is echter te summier om een eenvoudige remediëring toe te laten.

9.1.2 Vergelijking met HSL

Het HGPSS-systeem kan niet enkel geëvalueerd worden door de absolute voor- en nadelen te beschouwen, maar ook door het systeem te vergelijken met een ander systeem dat dezelfde objectieven nastreeft. In hetgeen volgt wordt HGPSS vergeleken met *HSL* (Hierarchical Simulation Language) [Sanderson 1991]. HSL werd ontwikkeld aan de *University of Pittsburgh* met het oog op het leveren van een veelzijdig stuk gereedschap voor object-georiënteerde process-interaction simulatie. De vergelijking wordt doorgevoerd aan de hand van een aantal van de eigenschappen en karakteristieken van HSL zoals deze beschreven worden in [Sanderson 1991]. Er wordt nagegaan of de elementen waaruit HSL bestaat ook in HGPSS aanwezig zijn. Eventuele additionele voordelen van HGPSS ten opzichte van HSL worden niet ten berde gebracht. Globaal gezien blijkt uit de vergelijking dat HGPSS op de meeste punten niet moet onderdoen voor HSL.

World view

Zowel HSL als HGPSS maken gebruik van de process-interaction world view waarbij een model wordt samengesteld uitgaande van interagerende processen.

Concept

HSL is een *procedurale taal* terwijl HGPSS in wezen een zogenaamde *scenario-taal* is. Een scenario-, *transaction flow-* of *transaction-oriented* taal laat toe om een systeem te modelleren via het specifiëren van scenarios die de levensloop van de transacties beschrijven. Veelal gebeurt dit door blokdiagrammen. Een procedurale simulatie-taal combineert general-purpose constructies met constructies specifiek voor simulatie. In het kader van simulatie worden ze in vergelijking met de scenario-talen als low-level beschouwd. Door het gebruik van general-purpose constructies zijn ze echter krachtiger en flexibeler dan scenario-talen. Alhoewel HGPSS in de grond ook een scenario-taal is, zijn de nadelen hieraan verbonden geneutraliseerd. Reeds in GPSS zorgen blokken als LOOP en TEST ervoor dat iteratie en selectie mogelijk zijn, waardoor meer kracht en flexibiteit worden geïntroduceerd. Door de uitgebreide mogelijkheden tot het weven van C++-code tussen de simulatie-statements, bevindt HGPSS zich eigenlijk halverwege tussen de scenario- en procedurale talen. De procedurale talen blijven als voordeel bezitten dat de syntax van de general-purpose- en de simulatie-constructies analoog is, terwijl tussen de syntax van deze constructies een groot verschil bestaat in het geval van HGPSS en in mindere mate HGPSS++.

Doelgroep

Door een gewone general-purpose programmeertaal ter beschikking te stellen voor de implementatie van een model, worden programmeurs bevoordeeld ten opzichte van modelbouwers. Bij gebruik van een taal specifiek gericht op het beschrijven van een model, is de situatie omgekeerd. Om beide groepen tevreden te stellen kan een nieuwe taal ontwikkeld worden of kan een bestaande taal aangepast worden. Zowel HSL als HGPSS hebben programmeurs én modelbouwers als doelgroep. Om deze beide groepen

tevreden te stellen werd in het geval van HSL geopteerd voor de ontwikkeling van een volledig nieuwe taal, terwijl HGPSS een uitbreiding vormt op GPSS.

Implementatie

HSL werd geïmplementeerd in de vorm van een interpreter terwijl HGPSS gecompileerd wordt. Het voordeel van interpretatie is de eenvoudige modifieerbaarheid terwijl compilatie een hogere performatie levert. Als modifieerbaarheid nagestreefd wordt, bestaat er echter geen enkele belemmering om HGPSS als interpreter te implementeren.

Vorm van een programma

In zowel HSL als HGPSS worden model en environmental frame gescheiden gehouden. Terwijl deze secties in HGPSS respectievelijk model- en commandosectie worden genoemd, worden in HSL de termen *simulator*- en *environment*-sectie gebruikt.

Hiërarchisch modelleren

Zowel HSL als HGPSS laten toe om een model hiërarchisch op te bouwen uit submodellen. In HSL worden twee verschillende soorten verfijningen syntactisch onderscheiden: gewone hiërarchische verfijning en laterale verfijning. In het eerste geval worden vanuit een proces een aantal deelprocessen opgestart die ná elkaar worden afgehandeld terwijl de deelprocessen in het tweede geval concurrent verlopen. In HGPSS zijn beide gevallen mogelijk zonder dat er evenwel een syntactisch verschil tussen bestaat.

Object-georiënteerd modelleren

In HSL zijn de meeste object-georiënteerde technieken van toepassing bij de beschrijving van entiteiten. Er kan gebruik gemaakt worden van klassen, inkapseling en overerving. De wijze waarop methodes aan een klasse worden toegekend is echter voor discussie vatbaar. Op het gebied van object-oriëntatie in verband met entiteiten scoort HGPSS zwak. Van voorgedefinieerde klassen kunnen instanties worden gecreëerd, maar de verzameling entiteiten kan niet rechtstreeks worden uitgebreid. Dit kan wel op het niveau van C++ gebeuren, mits ook de verzameling blokken uit te breiden zodat de nieuwe entiteiten kunnen worden gemanipuleerd.

Beschikbare entiteiten

In HSL zijn de volgende entiteiten beschikbaar:

- transacties (deze bezitten een identificatienummer en een prioriteit),
- statistieken,
- queues (LIFO, priority, random order),
- resources (first-come, first-served; first-fit; preemptive).

Voor al deze entiteiten bestaat in HGPSS een equivalent:

- transacties (deze bezitten onder andere een identificatienummer en een prioriteit),
- tables en andere objecten waarin data verzameld wordt,

- queues (LIFO, FIFO, volgens de inhoud van een parameter),
- facilities, storages.

Modelleren van stochastische processen

In HSL zijn met het oog op het modelleren van stochastische processen volgende hulpmiddelen voorhanden:

- random number generators die werken volgens de congruentiële methode,
- probabiliteitsdistributies,
- systeem- en user-defined statistieken in de vorm van objecten.

In HGPSS zijn analoge hulpmiddelen voorhanden.

Scheduling

Zowel in HSL als HGPSS wordt gebruik gemaakt van event lists die geordend zijn volgens tijd en prioriteit en die aangeven wanneer bepaalde processen moeten hervat worden. *Suspend* en *awaken* zijn acties die in HSL expliciet in een programma moeten opgenomen worden terwijl deze in HGPSS impliciet in bepaalde blokken aanwezig zijn.

Geheugenbeheer

In HSL moeten de te gebruiken objecten in een programma expliciet gecreëerd en verwijderd worden. In HGPSS gebeurt dit automatisch door declaratie of bij het eerste gebruik. Bij het beëindigen van een simulatie of bij het uitvoeren van bepaalde commando's worden objecten automatisch uit het geheugen verwijderd.

9.2 Mogelijke uitbreidingen en verbeteringen

Tot slot worden enkele uitbreidingen en verbeteringen besproken die mogelijk aan het HGPSS-systeem zouden kunnen aangebracht worden. Aangezien het ontwikkelen van een open systeem één van de doelstellingen van het project was, ligt de mogelijkheid tot uitbreiden steeds open. Ook het aanbrengen van verbeteringen moet eenvoudig zijn, gezien de kernel binnen een object-georiënteerde programmeertaal ontwikkeld werd.

De meest relevante uitbreidingen kunnen gevat worden in het concept van een *integrated development environment* voor de ontwikkeling van proces-georiënteerde discrete-event simulaties. Een dergelijke omgeving zou moeten toelaten om simulatie-programma's op een eenvoudige manier te ontwikkelen, te testen, uit te voeren en de resultaten te analyseren. De onderdelen waaruit de omgeving zou kunnen bestaan zijn:

- Een overkoepelende *program manager* die toelaat om de verschillende programma's die deel uitmaken van de omgeving te selecteren en op te starten.
- Een *teksteditor* voor het ontwikkelen en verbeteren van HGPSS-, HGPSS++- en C++-programma's.

- Een *grafische editor* voor het op een grafische wijze specifiëren van een model. Deze editor zal gebruik maken van een bibliotheek waarin de grafische voorstelling van de HGPSS-blokken en van eventueel zelf aangemaakte blokken is opgeslagen.

- De reeds bestaande *HGPSS naar HGPSS++ compiler* die een HGPSS-programma omzet naar het HGPSS++-equivalent.

- Een *C++-compiler* voor het omzetten van HGPSS++-modellen of -programma's naar object-code. De gecompileerde modellen kunnen dan eventueel in een model-bibliotheek opgenomen worden.

- Een *linker* voor het combineren van een simulatie-programma met eventuele in de model-bibliotheek residerende submodellen. Ook de reeds bestaande HGPSS++-kernel zal in het linking-proces worden gebruikt.

- Een *run-time executive* en *debugger* voor het uitvoeren en eventueel opsporen van fouten in een simulatie.

- Het hele precompilatie-, compilatie, linking- en debugging-proces kan ook worden vervangen door het invoeren van een *interpreter* die een simulatie direct uitvoert uitgaande van ingevoerde HGPSS-statements.

- Voor het *analyseren* van de door een simulatie geleverde resultaten kan ook een stuk software worden ingevoerd. De resultaten kunnen hiermee bijvoorbeeld in grafiek worden gezet of gebruikt als data voor statistische operaties.

- Een belangrijk deel van de omgeving zal ongetwijfeld de *database manager* zijn. Deze moet ervoor zorgen dat verschillende soorten gegevens op een efficiënte manier kunnen opgeslagen en opgevraagd worden. De te behandelen informatie kan bijvoorbeeld in verband staan met

  - de grafische voorstelling van symbolen,
  - standaard HGPSS- en zelfgedefinieerde blokken,
  - submodellen,
  - andere C++-routines voor gebruik in blokken of submodellen.

Andere uitbreidingen zouden het ontwikkelen van een *dedicated simulation environment* kunnen inhouden. Hierbij wordt het bestaande HGPSS-systeem uitgebreid met een aantal voorzieningen voor het op een eenvoudige manier simuleren van systemen uit een specifiek domein. Als voorbeeld kan logische simulatie beschouwd worden. Het HGPSS-systeem zou dan kunnen uitgebreid worden met een bibliotheek bestaande uit modellen voor veelgebruikte componenten. Naast deze bibliotheek zou ook een *netlist transformer* nuttig zijn. Een dergelijk programma vormt een *netlist* van een elektronisch netwerk om naar een HGPSS-beschrijving in termen van submodellen[1].

Naast het uitbreiden van het HGPSS-systeem kan het aanbrengen van enkele verbeteringen aan de huidige implementatie ook een direct nut hebben.

- In de HGPSS++-kernel kan de foutafhandeling sterk verbeterd worden. In de kernel wordt de uitvoering van een actie typisch geïnitieerd vanuit de hogere software-lagen en doorgespeeld naar de onderste lagen. Eventuele fouten zullen in de onderste lagen geconstateerd worden en vandaaruit gerapporteerd worden. Bij de rapportering van de fout wordt geen verband gelegd naar

[1] Nog interessanter zou een HGPSS++-beschrijving zijn.

de context waarin de actie oorspronkelijk geïnitieerd werd. Als bijvoorbeeld een entiteit wordt gedeclareerd met een reeds bestaand identificatienummer wordt dit gerapporteerd met de melding `Cannot register duplicate entity`. Hierbij wordt echter niet aangegeven om welke entiteitsklasse het hier gaat. Ook het identificatienummer dat aanleiding gaf tot de fout wordt niet gerapporteerd.

De foutrapportering bij de HGPSS-precompiler kan eveneens verbeterd worden, alhoewel de noodzaak hiertoe kleiner is.

- In de HGPSS++-kernel worden entiteiten gestockeerd op ketens. Deze ketens zijn geïmplementeerd als dubbel geketende lineaire lijsten. Deze voorstelling is efficiënt als het aantal entiteiten op de keten klein is. Bij de meeste ketens zal het aantal entiteiten typisch vrij klein zijn. Een submodel met bijvoorbeeld meer dan een tiental tables, variables of storages is uitzonderlijk. Het aantal blokken in een submodel kan echter wel groot worden, evenals het aantal transacties in het totale model en het aantal events op de event lists. Voor grote simulaties zal het gebruik van dubbel geketende lineaire lijsten nadelig worden. Het gebruik van één of andere boomvormige voorstelling zal dan betere resultaten leveren, alhoewel de overhead wat zal toenemen. Een belangrijke verbetering zou dus het wijzigen van de implementatie van de entiteitsketens kunnen zijn. De reden waarom initieel toch dubbel gekentende lijsten gebruikt werden is, naast het feit dat deze interessant zijn bij kleine lijsten, de eenvoud. Bij de implementatie van de kernel werd niet in de eerste plaats efficiëntie nagestreefd, maar eenvoud en doorzichtigheid.

Bibliography

[Aho, Sethi & Ullman 1986] Aho, Alfred V.
Sethi, Ravi
Ullman, Jeffrey D.
Compilers: Principles, Techniques and Tools
Addison-Wesley
1986

[Banks] Banks, Jerry
Carson, John S.
Getting started with GPSS/H
Wolverine Software Corporation

[Boehm & Jacopini 1966] Boehm, Corrado
Jacopini, Giuseppi
Flow diagrams, Turing machines, and languages with only two formation rules
Communications of the ACM, vol. 9, no. 5
1966

[Date 1990] Date, C.J.
An introduction to database systems
Addison-Wesley
1990

[Forsyth 1982] Forsyth, Charles H.
LEX: A lexical analyser generator
DECUS C Language System
University of Waterloo
Waterloo, Ontario, Canada 1982

[Fujimoto 1990] Fujimoto, Richard M.
Optimistic Approaches to Parallel Discrete Event Simulation
Transactions of the society of computer simulation, vol. 7, no. 2, pp. 153-191
1990

[Gordon 1978] Gordon, G.
System simulation
Prentice-Hall
Englewood Cliffs, New Jersey, U.S.A 1978

[Kreutzer 1986] Kreutzer, Wolfgang
System simulation programming styles and languages
International computer science series, Addison-Wesley
Canterbury, New-Zealand 1986

[Mason & Brown 1990] Mason, Tony
Brown, Doug
LEX & YACC
O'Reilly & Associates, Inc.
Sebastopol, California, U.S.A. 1990

[Meyer 1988] Meyer, Bertrand
Object-oriented software construction
Prentice-Hall
New York, New Jersey, U.S.A. 1988

[Neelamkavil 1987] Neelamkavil, Francis
Computer simulation and modelling
John Wiley & Sons
1987

[Pooley 1991] Pooley, R.J.
Hughes, P.H.
Towards a standard for hierarchical process oriented discrete event simulation diagrams
Transactions of the Society for Computer Simulation, vol. 8, no. 1
1991

[Sanderson 1991] Sanderson, D.P.
Sharma, R.
Rozin, R.
Treu, S.
The Hierarchical Simulation Language HSL: A Versatile Tool for Process-Oriented Simulation
ACM Transactions on Modeling and Computer Simulation, vol. 1, no. 2, pp. 113-153
1991

[Schildt 1990] Schildt, Herbert
Turbo C/C++ - The complete reference
Borland-Osborne/McGraw-Hill
Berkeley, California, U.S.A. 1990

[Schriber 1974] Schriber, Thomas J.
Simulation Using GPSS
John Wiley & Sons
New York, New Jersey, U.S.A. 1974

[Siemens 1979] *GPSS (BS1000, BS2000)*
Siemens-System 7-700
Siemens-System 4004
Anwendungs- und Programmbeschreibung, 1. Ausgabe
Siemens Aktiengesellschaft
Deutschland 1979

[Sim++ 1989] *Sim++ - A discrete-event simulation language*, Release 2.3 (Beta)
Jade Simulations International Corporation
Calgary, Alberta, Canada 1989

[Stroustrup 1987] Stroustrup, Bjarne
The C++ Programming Language
Addison-Wesley
1987

[Spriet & Vansteenkiste 1982] Spriet, Jan A.
Vansteenkiste, Ghislain C.
Computer-aided modelling and simulation
Academic Press
1982

[Vanwijnsberghe 1992] Vanwijnsberghe, Guido
Een object-georiënteerde kernel continue simulatie in µCSL
Afstudeerwerk ingediend tot het behalen van de graad van Licentiaat in de Informatica
Universiteit Gent
1992

[Zeigler 1976] Zeigler, B.P.
Theory of Modelling and Simulation
John Wiley & Sons
New York, New Jersey, U.S.A 1976