EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Toepasbaarheid en uitbreidingen van het real-time expertsysteem SIMPLEXYS

de Vries, A.A.

Award date:
1993

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

7041

FACULTEIT DER ELEKTROTECHNIEK
TECHNISCHE UNIVERSITEIT
EINDHOVEN
VAKGROEP MEDISCHE ELEKTROTECHNIEK

Toepasbaarheid en uitbreidingen van het real-time expertsysteem SIMPLEXYS

door: A.A. de Vries

Rapport van het afstudeerwerk
uitgevoerd van aug 1992 tot aug 1993
in opdracht van prof. dr. ir. J.E.W. Beneken
onder leiding van dr. ir. J.A. Blom

DE FACULTEIT DER ELEKTROTECHNIEK VAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN AANVAARDT GEEN AANSPRAKELIJKHEID VOOR DE INHOUD VAN STAGE- EN AFSTUDEERVERSLAGEN.

Abstract

This paper describes suggestions for extensions to the real-time expert system SIMPLEXYS. SIMPLEXYS expert system applications are diagnostics and control. SIMPLEXYS is not suitable for construction, design or planning tasks.

Unique in SIMPLEXYS is the checking that is performed on the rule-base. These checks however, are limited to small expert system applications only.
Splitting a large knowledge base in separate rather autonomous modules, allows separate checking of those modules and thus allows larger (verified) SIMPLEXYS applications.

An object oriented implementation of SIMPLEXYS is described in this report. This new implementation matches the existing version of SIMPLEXYS in terms of speed of execution, compactness and readability of code, and processes existing rule bases without any problems. I expect that the implementation of knowledge modules in Objective SIMPLEXYS is easier than in the previous version. Large global datastructures (arrays) in Objective SIMPLEXYS are avoided. Instead many smaller (dynamic) objects are used to hold the compiled rule base.

Samenvatting

In dit verslag worden uitbreidingen aangedragen voor het real-time expertsysteem SIMPLEXYS. SIMPLEXYS expertsysteem-applicaties zijn te karakteriseren als diagnostiserende-/regel-expertsystemen. SIMPLEXYS is niet geschikt voor constructie, ontwerp of planning.

Uitgebreide controles kunnen uitgevoerd worden op een SIMPLEXYS rule-base om de correctheid van de rule-base te testen. De tools die deze controles uitvoeren kunnen slechts kleine rule-bases controleren.
Het idee is nu een grote rule-base te splitsen in kleinere min of meer zelfstandige modulen, die onafhankelijk door de controle-tools geverifiëerd kunnen worden. De modulen gezamelijk vormen zodoende een groter (gecontroleerd) expertsysteem dan dat voorheen mogelijk was.

Een objectgeoriënteerde implementatie van SIMPLEXYS is onderzocht en wordt in dit verslag beschreven. De objectgeoriënteerde implementatie is equivalent qua executiesnelheid, compactheid en leesbaarheid van de code en is in staat bestaande rule-bases te verwerken zonder dat modifiacties van de rule-bases noodzakelijk zijn.

Mijn verwachting is dat kennismodulen in Objective SIMPLEXYS eenvoudiger te implementeren zijn dan in de bestaande versie, zonder dat snel tegen maximale array-grootten aangelopen wordt.

Inhoud

Figuren ix

Tabellen xi

Listings xiii

1 Inleiding 1

1.1 Doel 1
1.2 Werkmethode 1
1.3 Verrichtingen 2
1.4 Uitgangspunten 3

2 Toepassingsgebied SIMPLEXYS 5

2.1 Classificatie expertsystemen 5
2.2 Elektrotechnische toepassingen van expertsystemen 6
2.3 Inventarisatie SIMPLEXYS 7

3 Beperkingen van SIMPLEXYS 9

3.1 Algemene beperkingen 9
3.1.1 Modularisatie 10
3.1.2 Database-interfaces 12
3.2 Beperkingen door SYMPLEXYS implementatie 12
3.3 Inventarisatie SYMPLEXYS codering 14
3.3.1 De SIMPLEXYS tools 14
3.3.2 C/Pascal 16
3.3.3 Beveiliging 17
3.3.4 Leesbaarheid van de code 17
3.3.5 Snelheidsaspecten 17
3.3.6 Initialisatie code 19
3.3.7 Debugging opties 19
3.3.8 Optimalisatie redundancies. 20
3.3.9 Redeneren met tijd 20
3.4 Codering andere expertsystemen 21

4 Objective SIMPLEXYS 23

4.1 De objectstructuur 24
4.2 Datastructuren 25
4.2.1 On-Statements 26
4.2.2 Rules 27
4.2.3 Datastructuren rule 27
4.2.4 Do-, Tes-t en History-code 30
4.3 Werking 32
4.4 XLAT 36
4.5 Objective Rule Compiler 37
4.6 Integratie controle-tools in Rule Compiler 37

5 Objective SIMPLEXYS vs. SIMPLEXYS 39

5.1 Complexiteit codering 39
5.2 Verschillen in snelheid 40
5.3 Verschillen in geheugengebruik 42
5.3.1 Code-grootte 42
5.3.2 Geheugengebruik datastructuren 42
5.4 Verruiming limieten 44

6 Conclusies en Aanbevelingen 47

6.1 Conclusies 47
6.2 Aanbevelingen 47

Bijlage 1 Objectgeoriënteerd programmeren 49

Bijlage 2 DLLs 61

Bijlage 3 Borland Pascal 7.0 67

Bijlage 4 Inheritance diagrammen 71

Bijlage 5 CHK41 77

Bijlage 6 PET41 81

Bijlage 7 Vergelijkingstests 87

Bijlage 8 Foutjes in implementatie SIMPLEXYS . 97

Referenties 101

Index 103

Figuren

Figuur:

1 Classificatie expertsystemen 5
2 Structuur van een kennisdomein. 10
3 Objectklassenhiërarchie van te evalueren objecten. 24
4 Objectklassenhiërarchie van thelse-objecten. 25
5 Datastructuren On-statements. 27
6 Datastructuren regel. 30
7 Een hiërarchie van objectklassen 53
8 Inheritance-diagram van TStringCollection. 55
9 Voorbeeld inheritance diagram. 71
10 Enkelvoudige inheritance diagrammen 72
11 Inheritance diagram OThelse. 73
12 Inheritance diagram O2Evaluate. 74
13 Inheritance diagram ORuleCollection. 75

Tabellen

Tabel:

1 Categorieën. 6
2 Beperkingen SIMPLEXYS. 13
3 Geheugengebruik regel. 43
4 Geheugengebruik On-statements. 44
5 Setoperatoren Pascal. 82
6 Run-tijden (alle primitieven FA). 89
7 Run-tijden DPM (alle primitieven FA). 90
8 Run-tijden (alle primitieven TR). 90
9 Run-tijden DPM (alle primitieven TR). 91
10 Run-tijden verschillend aantal regels. 92
11 Redeneersnelheid. 92
12 Code-grootten. 94
13 Code-grootten in DPM. 95

Listings

Listing:

1 Type declaraties voor on-statements. 26
2 Type declaraties globale collections. 27
3 Superklasse regeltypen. 28
4 Rule-typen. 29
5 Procedure _FDos. 30
6 Unit met Do-code. 31
7 Recursieve evaluatie. 33
8 Bestaande implementatie thelses 34
9 Object georiënteerde implementatie thelses. 35
10 Objectdeclaratie TCollection. 50
11 Objectdeclaratie TSortedCollection. 53
12 Typedeclaratie dynamische objecten. 56
13 Voorbeeld DLL. 62
14 DLL met slechts index referenties. 63
15 Expliciet gebruik DLL. 63
16 Dynamisch gebruik DLLs. 64
17 Functie om in een N * N open array een cel te adresseren... 79
18 Type declaratie dynamische sets. 81
19 InitSetPtr. 82
20 Een set-compare functie. 84
21 Regelbestand voor snelheidstest 88
22 Regelbestand om code-grootte te testen. 93

1 Inleiding

1.1 Doel

In de vakgroep Medische Elektrotechniek is een real-time expertsysteem ontwikkeld genaamd SIMPLEXYS. Snelheid van executie, compactheid (het expertsysteem moet op PCs werken), de correctheid en de aanwezigheid van een interface met een standaard programmeertaal hebben voorop gestaan tijdens de ontwikkeling van SIMPLEXYS.

De tools die SIMPLEXYS opspannen voldoen dan ook aan deze uitgangspunten: het systeem redeneert ongekend snel, is compact en om de veiligheid van een systeem te kunnen garanderen, zijn veel inspanningen verricht om de correctheid van het expertsysteem te controleren.

De vraag was nu of het systeem voldoende constructies bezit om allerlei elektrotechnische toepassingen in SIMPLEXYS te kunnen realiseren. Bestudering van dergelijke toepassingen zouden mogelijke uitbreidingen voor SIMPLEXYS aan het licht kunnen brengen.
Het hoofddoel van dit afstudeerwerk was dan ook te komen tot voorstellen voor mogelijke aanvullingen op SIMPLEXYS.

1.2 Werkmethode

Een vak als "kennissystemen" is geen verplicht onderdeel van de studie Informatie Techniek. Om in de materie van kennissystemen thuis te raken, ben ik begonnen met het volgen van dit vak. Een tweede stap die ik tegelijkertijd heb ingezet, vormde de bestudering van SIMPLEXYS voornamelijk door het zorgvuldig lezen van [BLO90]. Tijdens het lezen van dit proefschrift over SIMPLEXYS, heb ik reeds oppervlakkig naar de codering van de SIMPLEXYS tools gekeken.
Vervolgens ben ik gaan zoeken in de literatuur. Over expertsystemen in het algemeen zijn rekken vol geschreven. Diepgang in de boeken/artikelen was echter ver te zoeken. Een oorzaak hiervoor is dat wijdverspreide standaarden op het gebied van expertsystemen in de wereld ontbreken. Wat dit betreft zijn bijvoorbeeld conventionele programmeertalen (als Pascal of C) beter af, omdat boeken op ieder niveau over een programmeertaal te krijgen zijn. Het onderzoek naar expertsystemen wordt daardoor pragmatisch benaderd: op vele plaatsen vindt onderzoek plaats aan overal verschillende systemen. SIMPLEXYS vormt hierop geen uitzondering.
Veel is geschreven over beslissing-ondersteunende expertsystemen die in de algemene maatschappij worden toegepast (management-ondersteuning, planning, juridisch e.d.). Wanneer we als ingang in Vubis (het elektronische bibliotheekcatalogussysteem op de TUE)

een trefwoord (of woord uit titel) als 'expertsystemen' genomen wordt, dan worden vele honderden titels gevonden. Bij beperking met bijvoorbeeld "real-time" resteerden dan nog slechts zes titels.
Een andere weg voor literatuuronderzoek vormde het "Jaarboek Kennissystemen 1991". Naast een lijst van boeken over dit onderwerp, waren ook tijdschriftartikelen uit 119 tijdschriften in een overzicht opgenomen in dit jaarboek. Al hoewel bijna 70% van deze tijdschriften aanwezig waren op de universitaire bibliotheken, bevatten het geringe aantal artikelen, die handelden op het vlak waarop SIMPLEXYS ook opereert weinig concrete informatie.
Tijdens het lezen over expertsystemen en de algemene principes van kennissystemen die behandeld werden tijdens het college, ben ik voldoende stof tegengekomen op basis waarvan SIMPLEXYS nuttig aan kracht kan winnen. De vier uitgangspunten aan de hand waarvan SIMPLEXYS ontwikkeld is, zijn met succes gerealiseerd.
De huidige implementatie heeft echter ook zijn beperkingen. De voornaamste beperking is dat in SIMPLEXYS sSlechts kleine expertsystemen gerealiseerd kunnen worden (zie §3.2).
Erg opvallend bij het lezen over expertsystemen was, dat deze gerealiseerd zijn in talen als Prolog en LISP, of dat voor een objectgeoriënteerde taal gekozen was. Om andere expertsystemen die in een objectgeoriënteerde taal geprogrammeerd waren te kunnen bestuderen en omdat ik de indruk kreeg dat deze programmeertechniek de bovenstaande beperkingen zou kunnen verlichten, heb ik objectgeoriënteerd leren programmeren.

1.3 Verrichtingen

Het vermoeden of een objectgeoriënteerde implementatie inderdaad tot verbeteringen (snellere, compactere en meer modulaire code) zou leiden, heb ik onderzocht door het redeneermechanisme objectgeoriënteerd te implementeren. Deze beperkte stap (dus zonder de overige tools te wijzigen) moest voldoende inzicht verschaffen om te kunnen beoordelen of SIMPLEXYS inderdaad gebaat is bij objectgeoriënteerde implementatie.

De bestaande implementatie van het redeneermechanisme heb ik vertaald in een objectgeoriënteerde implementatie. Deze vertaling heeft niet één op één plaatsgevonden. De doelstellingen bij deze vertaling waren:

- Upward compatibiliteit van regelbestanden; bestaande regelbestanden moeten zonder meer door het objectgeoriënteerde redeneermechanisme verwerkt kunnen worden.
- De redeneersnelheid moest behouden blijven.
- De eenvoud van de codering mocht niet verdwijnen.
- Objective SIMPLEXYS moet een meer lokale codering hebben. Toekomstige wijzigingen in de code zijn dan eenvoudiger aan te brengen.

Het redeneermechanisme is gerealiseerd in objectgeoriënteerde vorm. Om deze implementatie te kunnen testen is eveneens een translator gemaakt, die het gecompileerde regelbestand afkomstig van de bestaande Rule Compiler om te zetten in een objectgeoriënteerde beschrijving, waarmee het nieuwe redeneermechanisme kan werken.

Om in SIMPLEXYS grotere kennissystemen te kunnen implementeren, wordt gesuggereerd een regelbestand in modulen te kunnen opsplitsen. De Rule Compiler zal daartoe het meest veranderd moeten worden. De datastructuren van Objective SIMPLEXYS hoeven nauwelijks aangepast te worden, wanneer kennismodulen deel uit maken van de SIMPLEXYS syntaxis. Beperkingde limieten als maximale array-grootten doemen niet snel op in Objective SIMPLEXYS.

1.4 Uitgangspunten

Dit verslag is geschreven voor een lezer met voorkennis van expertsystemen[1] en SIMPLEXYS[2] in het bijzonder. Dit verslag is vooral geschreven voor mijn begeleider en aan toekomstige stagiaires en afstudeerders die SIMPLEXYS verder ontwikkelen.

1. Het niveau dat bereikt wordt in het college "kennissystemen" dat gegeven wordt aan de hand van het boek [JAC90].

2. Zie proefschrift [BLO90]. Ook kennis van de sources van de SIMPLEXYS tools is gemakkelijk om dit verslag te kunnen lezen.

2 Toepassingsgebied SIMPLEXYS

2.1 Classificatie expertsystemen

Expertsystemen zijn op verschillende manieren te classificeren. Ik maak hier een zeer grove tweesplitsing in diagnostiserende en oplossing-genererende expertsystemen:

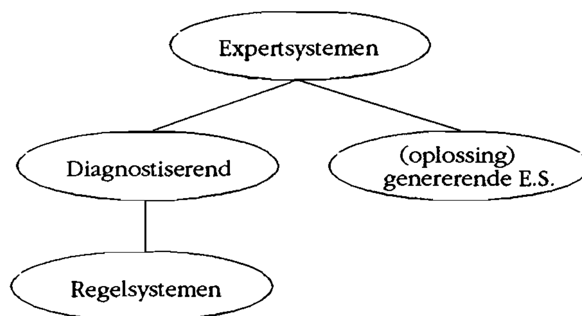

Fig. 1: Classificatie expertsystemen

De diagnostiserende expertsystemen analyseren een (klein) domein gebied. Het aantal mogelijke toestanden is aftelbaar. Wanneer bijvoorbeeld een proces of patient is geanalyseerd, is eventueel automatische (real-time) regeling mogelijk. Deze expertsystemen die eigenlijk regelsystemen zijn vormen een subklasse van de diagnostiserende expertsystemen.

Oplossing genererende expertsystemen worden gebruikt voor constructie, ontwerp of planning. De kenmerken van deze systemen zijn dat de systemen niet real-time zijn, en beschikken over een werkgeheugen waarin dynamisch één of meerdere (deel-)oplossingen tot stand komen. De systeemeisen voor dergelijke systemen zijn groot (snelheid en hoeveelheid werkgeheugen waarover de computer beschikt).

Uit paragraaf 2.3 en hoofdstuk 3 zal blijken dat SIMPLEXYS onder de diagnostiserende expertsystemen valt.

2.2 Elektrotechnische toepassingen van expertsystemen

Het literatuuronderzoek naar elektrotechnische toepassingen van expertsystemen is voornamelijk verricht aan de hand van INSPEC op CD ROM. Uit dit medium heb ik meer dan 1000 titels van artikelen gefilterd en uitgesplitst naar categorie.

Tabel 1: Categorieën.

Categorie	Aantal titels
CIM	41
Chips+	52
Computer+	12
Control	195
Design	38
Diagn+	96
Elektro+	55
Medisch+	10
Netwerk+	9
Object+	22
PCBs+	10
Planning	42
Power+	118
Real-Time+	9
Robot+	55
Rule+	3
Software+	13
Speech	4
Vision	19
Unknown	29
-	251

29 artikelen kan ik aan de hand van de titel niet classificeren. 251 titels vond ik bij voorbaat niet interessant in relatie tot SIMPLEXYS. De onderwerpen die gemarkeerd zijn met een +, zijn elektrotechnische toepassingen van expertsystemen. Helaas zijn veel van deze categorieën niet bruikbaar voor het vinden van toepassingen van elektrotechnisch gebruik van SIMPLEXYS omdat het veelal constructie, ontwerp of plannende expertsystemen zijn (PCB

ontwerp, ontwerp van schakelingen (al dan niet digitaal). Soms zijn de zoekruimten bijzonder groot wanneer zelfs van een diagnosisch expertsysteem gesproken kan worden (foutzoeken in schakelingen, digitale dan wel analoge, of foutzoeken op PCBs).

In de vetgedrukte categorieën in tabel 1 worden veel meer applicatities gevonden die met SIMPLEXYS gerealiseerd zouden kunnen worden. Bij het foutzoeken (diagnose) in schakelstations van distributienetwerken (categorie Power) is van een beperkte zoekruimte sprake. Het aantal fouten dat kan optreden in een schakelstation is aftelbaar klein.
In de robotica kan een SIMPLEXYS expertsysteem wellicht zijn diensten bewijzen, zolang er geen sprake is van Vision. Het grootste toepassingsgebied vormt de categorie control. De artikelen uit deze categorie handelen bijna alle over toepassingen in de (proces-)chemie.

Uitdraaien van de titels van de artikelen gebundeld naar categorie zijn in handen van mijn begeleider. De complete referenties kunnen teruggevonden worden op een begeleidende diskette die hoort bij de uitdraaien.

2.3 Inventarisatie SIMPLEXYS

Indien je meer en meer leest omtrent expertsystemen en ook leest wat de verschillende expertsysteem shells allemaal kunnen, dan vinden we een aantal van deze kenmerken direct in SIMPLEXYS terug:

- Het systeem is op regels gebaseerd (veel expertsystemen hebben een rule-base, al dan niet gecombineerd met bijvoorbeeld frames). In SIMPLEXYS kan er zowel **achterwaarts** (goal directed) alsook **voorwaarts** geredeneerd worden.
- Een regel kan als conclusie slechts waar, onwaar dan wel onbekend krijgen. Er is dus gekozen niet te werken met een waarschijnlijkheidsinterval (bijvoorbeeld waarden tussen 0 en 1). De achtergrond hiervan is dat een (real-time) expertsysteem altijd een besluit dient te nemen (wel of geen actie). Is de beschikbare hoeveelheid informatie niet voldoende voor een regel om tot een bindende conclusie te komen, dan wordt dit kenbaar gemaakt door de waarde PO (possible) als conclusie aan de regel toe te kennen. Andere regels in het kennisbestand moeten dan geraadpleegd worden om toch te kunnen besluiten wel of niet tot actie over te gaan. Het PO worden van een regel kan ook direct tot actie leiden.
- Voor een expertsysteem redeneert SIMPLEXYS ongekend snel. Daar SIMPLEXYS is ontworpen als een real-time expertsysteem, is deze eigenschap niet echt verwonderlijk. De snelheid van redeneren wordt bereikt door de rule-base te compileren[1].
- SIMPLEXYS heeft een interface naar de onderliggende programmeertaal (Pascal c.q. C). Via deze interface komt de binding tot stand met de real-time wereld, waarin het SIMPLEXYS expertsysteem opereert.

---

1. Andere real-time expertsystemen die ik heb aangetroffen die eveneens gecompileerd worden zijn: Tirs, Level 5 en Egeria [STA91] en STRESS 2 [SLU92].

Een dergelijke interface naar conventionele programmeertalen wordt veelal gewenst bij een expertsysteem, zodat het expertsysteem geïntegreerd kan worden met (de massa) software die reeds in een geautomatiseerde omgeving aanwezig is.

Eigenschappen die in verband staan met real-time aspecten van expertsystemen:

- Een schatting van de tijd die maximaal nodig is om het regelbestand te evalueren kan worden gemaakt. Dit is mogelijk doordat SIMPLEXYS niet in een multitasked operating systeem draait en garbage collection niet voorkomt in SIMPLEXYS.
- In SIMPLEXYS is het mogelijk te redeneren met tijd.

Wanneer je over een expertsysteem leest, dan wordt altijd uitvoerig gerept over de mogelijkheden hoe kennis in het expertsysteem kan worden ondergebracht, met kennis geredeneerd wordt, met onzekerheid wordt omgesprongen. Voor een op regels gebaseerd expertsysteem worden de volgende voordelen aangedragen:

- De kennis die is ondergebracht in de afzonderlijke regels, is goed leesbaar en tevens is de juistheid van elke regel te controleren.
- De veelal duidelijke scheiding tussen de kennis die is ondergebracht in de regels en de wijze waarop het redeneermechanisme redeneert met die regels.

Het belangrijkste negatieve neveneffect van een op regels gebaseerd expertsysteem, is het consistent houden van het regelbestand wanneer dit ontwikkeld/uitgebreid wordt, zeker wanneer het een groot regelbestand betreft. Nadat de syntactische correctheid van de regels is geverifieerd, controleert SIMPLEXYS het regelbestand (zo goed mogelijk) op mogelijke logische fouten. Dit is een welkome ondersteuning, wanneer een werkend regelbestand gewijzigd wordt. Dat aan logische controle in SIMPLEXYS zeer veel aandacht is besteed, is af te meten aan de verhouding van de hoeveelheid programmaregels tussen bijvoorbeeld het totale redeneermechanisme en de hulpmiddelen die de logische controles uitvoeren (±700 regels versus ±2200 regels Pascal code). Dit laatste getal is de hoeveelheid regels zonder de syntaxis controle (dus alleen CHK41.PAS en PET41.PAS).

Een ander groot voordeel wanneer je kennismaakt met SIMPLEXYS is, dat het eenvoudig is geprogrammeerd:

- Er zijn geen complexe datastructuren (de regels worden bijvoorbeeld gecompileerd naar een tabel (array), zodat een regel door een simpele 'table-lookup' is terug te vinden, zonder dat tijd verloren gaat aan zoeken).
- De programmamodules zijn klein.
- Commentaar is niet in die mate toegevoegd dat de op zichzelf heel leesbare code erin verzuipt.

Door deze opmerkelijke eenvoud is een buitenstaander wellicht geneigd lichtzinnig over SIMPLEXYS te gaan denken. Naar mijn mening is het juist een kunst (en niet gemakkelijk) om software eenvoudig en onderhoudbaar te houden.

Nu de sterke punten van SIMPLEXYS geïnventariseerd zijn, wordt het tijd de beperkingen die aan SIMPLEXYS verbonden zijn te bekijken.

3 Beperkingen van SIMPLEXYS

Beperkingen die SIMPLEXYS met zich meedraagt, komen op twee manieren aan het licht:

- Door te lezen over andere (real-time) expertsystemen. Hierbij kom je aspecten van kennissystemen tegen die niet in SIMPLEXYS gerealiseerd zijn. Natuurlijk kunnen niet alle aspecten die aangetroffen worden bij expertsystemen binnen één expertsysteem gerealiseerd worden. Er zijn in SIMPLEXYS keuzen gemaakt (waaruit beperkingen voortvloeien), maar de mogelijkheden van SIMPLEXYS kunnen naar mijn mening zinvol uitgebreid worden. Vooral constructies die gebruikt worden bij expertsystemen die ontwerp- of planningtakken verrichten zijn niet zinvol in SIMPLEXYS (working memory, TMS *(Truth Maintenance Systems)* en het simultaan onderhouden van meerdere toestandsbeschrijvingen van een domeingebied).
- Door de codering (van vooral de datastructuren) van SIMPLEXYS nader onder de loep te nemen, doemen er verschillende grenzen op (denk hierbij bijvoorbeeld aan het maximale aantal regels dat SIMPLEXYS aan kan (zie §3.2)). Beperkingen worden deels veroorzaakt door het hardware platform (PC), waarop SIMPLEXYS draait. Ook de programmeertaal waarin SIMPLEXYS is geïmplementeerd (Turbo Pascal) legt beperkingen op.

In de volgende paragrafen worden de beperkingen afzonderlijk genoemd. Waar mogelijk wordt reeds (oppervlakkig) aangeduid hoe een dergelijke beperking binnen SIMPLEXYS opgelost zou kunnen worden.

3.1 Algemene beperkingen

Zoals reeds eerder is vermeld, kunnen alle technieken die we heden ten dage aantreffen bij expertsystemen niet zinvol ondergebracht worden in één expertsysteem. Voor SIMPLEXYS is bijvoorbeeld gekozen om kennis onder te brengen in regels, en wordt onzekerheid op basis van een driewaardige logica afgehandeld. Voor deze rule-base met driewaardige logica is een redeneermechanisme ontwikkeld, en zijn bijbehorende tools ontwikkeld die het regelbestand op mogelijke fouten controleren.
Ik ga hier geen mogelijke uitbreidingen opsommen die geïmplementeerd zouden kunnen worden in SIMPLEXYS die tornen aan deze basis, daar deze huidige basis door de eenvoud een zeer solide ondergrond is, waarop zinvolle uitbreidingen mogelijk zijn. Constructies nodig voor ontwerp- en planningtaken worden niet genoemd.

Ik draag mogelijke uitbreidingen aan, die in de toekomst gerealiseerd kunnen worden. Kennismodulen die in de volgende paragraaf beschreven worden, is de meest belangrijke uitbreiding voor SIMPLEXYS. In §3.3 worden enkele wijzigingen voorgesteld t.a.v. de implementatie van de SIMPLEXYS tools. Het redeneren met de tijd kan aan kracht winnen (zie §3.3.9).

3.1.1 Modularisatie

Kenniselicitatie is het gedeelte van het **kennisacquisitie** proces, waarbij kennis onttrokken wordt aan een expert en de structurering van die kennis. De gestructureerde kennis wordt ook wel **kennismodel** (conceptual model) genoemd.
In het algemeen zal het zo zijn dat verschillende delen van de kennis als in een soort boomstructuur met elkaar samenhangen. De root wordt gevormd door een algemene doelstelling, die is op te splitsen in meerdere sub goals. Deze sub goals zijn op hun beurt weer verder op te splitsen[1]. Denk bijvoorbeeld aan het monitoren en regelen van meerdere grootheden van een patiënt (bloeddruk, ademhaling ...). De main goal vormt het regelen van al deze grootheden. Een sub goal kan het regelen van de beademing zijn. Het regelen van de beademing op zich kan wellicht verder opgesplitst worden (volledige beademing, ondersteunende beademing enz.).

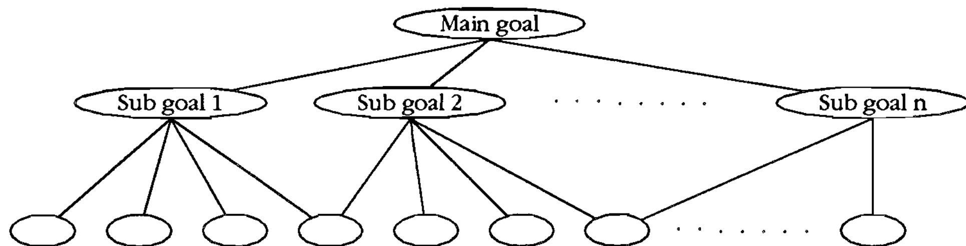

Fig. 2: Structuur van een kennisdomein.

Het zou logisch zijn deze kennis als dergelijke modulen te kunnen implementeren. Dergelijke modulen zijn helaas niet geïmplementeerd in SIMPLEXYS; de regels worden in één regelbestand aangeboden aan de Rule Compiler. Een voordeel van een dergelijke modulariteit is dat de logische controles die nu op het regelbestand worden uitgevoerd veel sneller klaar zijn. Deze controles hebben veelal een tijdcomplexiteit van $(\#regels)^3$. Indien dus een groot regelbestand opgesplitst wordt in kleinere modulen, dan is de tijdwinst aanzienlijk wanneer die modulen afzonderlijk gecontroleerd worden. Van zo'n module hoeven slechts de regels van die module plus de regels die uit andere modulen geïmporteerd worden, voor de logische controle in beschouwing te worden genomen.

[1]. Ook de source files van grote computerprogramma's zijn als een boom gestructureerd, denk maar aan de verschillende library files ("unit" files in Turbo Pascal) en de project file die de verschillende library files gebruikt ("program" file in Turbo Pascal).

De Pascal versie van SIMPLEXYS is nu niet in staat regelbestanden met meer dan 180 regels te controleren op logische fouten (zie §3.2). Indien een modulaire structuur van de regels ondersteund zou worden, kunnen veel grotere expertsystemen in SIMPLEXYS geïmplementeerd worden. Deze kunnen dan profiteren van de redeneersnelheid van SIMPLEXYS.

Kennismodulen kunnen als Turbo Pascal units onderling samenhangen. Een interface-sectie moet aan een SIMPLEXYS regelbestand toegevoegd worden, waarin regels opgesomd worden die in een bovenliggende kennismodulen gebruikt mogen worden. Dit komt overeen met procedure/functie headers in de interface-sectie van een Turbo Pascal unit.

In een SIMPLEXYS expertsysteem worden de regels soms ook gebruikt als variabelen in Pascal waaraan Thelses van andere regels waarden toekennen. Thelses aan regels in andere modulen zouden niet toegestaan mogen worden omdat:

- Dan nauwelijks nog gesproken kan worden van een modulaire implementatie van kennis. De modulen zouden dan slechts een opsplitsing van het huidige regelbestand in meerdere bestanden zijn.
- Thelses niet zijn toegestaan aan successors en predecessors van een regel in SIMPLEXYS.
- De Semantics Checker niet meer dan 180 regels kan controleren. Worden thelses toegestaan aan regels in andere modulen, dan kan de Semantics Checker niet meer per module te werk gaan.
  Wanneer de Semantics Checker per module de correctheid controleert, hoeven slechts de regels van één module plus de regels die de module importeert gecontroleerd te worden. De geïmporteerde regels kunnen als primitieven beschouwd worden.

Goal thelses naar regels in onderliggende modules zijn niet bezwaarlijk. Indien in een submodule een protocol afgewerkt moet worden (bijvoorbeeld in de module die de bloeddruk regelt), zal een constructie aan SIMPLEXYS moeten worden toegevoegd die een run van het protocol in die module activeerd. In de interfacesectie kan een aanduiding worden opgenomen, die aanduidt dat de betreffende submodule een protocol heeft.

De Rule Compiler moet zodanig worden uitgebreid, dat deze een automatische make-functie kent. Wanneer de Semantics Checker geïntegreerd is in de Rule Compiler, moet de Rule Compiler de regels van één module aan de Semantics Checker aanbieden. De regels die de betreffende module importeerd zijn als primitieven te beschouwen en hoeven daarom niet meegenomen te worden in de semantische controle.
Voor de modulen die een protocol in zich herbergen moet de Rule Compiler de Protocol Checker aanroepen. Het aanroepen van de controle-tools zou slechts mogen plaatsvinden indien de module daadwerkelijk gewijzigd is (karakteristiek van een goede make-implementatie), omdat de controle-tools vaak seconden tot minuten computertijd in beslag nemen.

In een objectgeoriënteerde implementatie van SIMPLEXYS is een kennismodule als één object (soort van record) te realiseren. Een dergelijk object beheert alle gegevens van een module (de regels, het protocol en de Pascal-code behorend bij die module).

3.1.2 Database-interfaces

Veel kennissystemen die ik ben tegen gekomen hebben interfaces naar databases of soms ook naar spreadsheets (SQL, DBase, Lotus, Excel e.d.), om te kunnen redeneren over gegevens die reeds in de vorm van computerbestanden bestaan[1]. Wanneer dit wenselijk wordt geacht, zouden deze interfaces via de Pascal/C interface in SIMPLEXYS gerealiseerd kunnen worden. Een database-interface is in een SIMPLEXYS applicatie reeds een keer ontwikkeld.

Indien echter een werkgeheugen met daarin data-objecten zou zijn toegevoegd aan SIMPLEXYS, zal de interface tussen het SIMPLEXYS kennisbestand en een dergelijke database/spreadsheet op een analoge wijze kunnen geschieden als tussen het kennisbestand en het werkgeheugen. Bijvoorbeeld een spreadsheet als geheel is namelijk ook een object, dat weer onder te verdelen is in objecten die binnen de spreadsheet aanwezig zijn (bijvoorbeeld zijn verschillende kolommen met gegevens ieder ook op te vatten als een object, evenals elke cel binnen een spreadsheet).

3.2 Beperkingen door SYMPLEXYS implementatie

SYMPLEXYS kent beperkingen die voortkomen uit:
- het hardware platform (PC) waarop SYMPLEXYS ontwikkeld is;
- de programmeertaal (Turbo Pascal/C);
- de wijze van codering (programmering).

De voornaamste beperkingen die de hardware veroorzaakt is dat de Intel 80X86 processor familie een gesegmenteerd geheugen model kent. Dit veroorzaakt dat zowel het code-segment als het data-segment van een programma (of unit) alsook het globale stack segment maximaal maar 64K groot kunnen zijn. Een gewoon DOS programma kan bovendien over maar maximaal ±600K werkgeheugen beschikken. Deze barrière kan doorbroken worden door een programma te compileren tot een **DPM** "*Dos Protected Mode*" programma zie bijlage: "Borland Pascal 7.0". Een DPM programma kan 16 Mb werkgeheugen van de PC adresseren. Borland Pascal 7.0 kan helaas geen 32-bit DPM programma's maken (die alleen op processoren vanaf een 80386 kunnen lopen). Alleen 16-bit offsets vanaf een segment-base adres worden ondersteund. Dit impliceert dat de segmenten in DPM nog altijd niet groter kunnen zijn dan 64K (32-bit offsets maken 4 GB segmenten mogelijk).

---

1. In [STA91] is een overzicht opgenomen van expertsysteem shells die verkocht worden op de Nederlandse markt. Beknopte beschrijvingen van de mogelijkheden die iedere shell biedt, zijn opgenomen in dit overzicht.

De programmeertaal Turbo Pascal heeft bovendien als nadeel dat sets niet meer dan 256 elementen kunnen bevatten. Ook is het niet mogelijk in Turbo Pascal te werken met datastructuren (bijvoorbeeld arrays) die groter zijn dan 64K. Het kost de programmeur enige moeite om om deze limieten heen te werken[1].

Het SYMPLEXYS regelbestand wordt gecompileerd tot arrays. In totaal kunnen binnen één unit/program sectie maar 64K aan variabelen (zoals arrays) worden gedefinieerd.
Deze barrière kan omzeild worden door i.p.v. statische declaratie (binnen var-sectie) geen arrays te definiëren, maar pointers naar dergelijke arrays. Een pointer kan wijzen naar een structuur die maximaal 64K groot kan zijn. Een dergelijke structuur kan tijdens het runnen van het programma worden gecreëerd, dan wel worden vernietigd (dynamische structuur). Een dynamische structuur wordt opgeslagen op de heap. De heap is de vrije geheugenruimte die in DOS overblijft nadat een programma geladen is. Zodoende bepaalt de totale hoeveelheid vrije geheugenruimte de grootte van de datastructuren en niet één enkel data-segment, waarbinnen alle arrays gedefinieerd worden.

Inventarisatie van de huidige implementatie van SIMPLEXYS levert de volgende essentiële beperkingen op:

Tabel 2: Beperkingen SIMPLEXYS.

<table>
<tr><th>max #</th><th>RUC41</th><th>CHK41</th><th colspan="3">PET41</th><th>SIM41</th></tr>
<tr><td>On-statements</td><td>100</td><td></td><td colspan="3"></td><td></td></tr>
<tr><td>On-list length</td><td>1000</td><td></td><td colspan="3"></td><td></td></tr>
<tr><td>rules</td><td>800</td><td>180</td><td colspan="3"></td><td>±3000</td></tr>
<tr><td>eval-items</td><td>8000</td><td></td><td colspan="3"></td><td></td></tr>
<tr><td>thelses</td><td>5000</td><td></td><td colspan="3"></td><td></td></tr>
<tr><td>thelsebys</td><td>2000</td><td></td><td colspan="3"></td><td></td></tr>
<tr><td>state rules
trigger rules</td><td></td><td></td><td>256
256</td><td>40
16</td><td>24
32</td><td></td></tr>
</table>

Toelichting bij tabel 1:
- De maxima die de Rule Compiler (RUC41) oplegt, zijn als constanten opgenomen in de code. In totaal wordt daar ±42K als arrays in het datasegment gedeclareerd.
- De Semantics Checker (CHK41) gebruikt een 2-dimensionaal array (#regels * #regels), waarbij elke cel twee bytes beslaat. De beperking is dus: $\sqrt{65.536/2} \approx 180$ regels (65.536 is de maximale grootte van een segment).

[1] De library 'SetLib' (zie bijlage 6) voorziet in routines die operaties op sets van meer dan 256 elementen mogelijk maken.

- De Protocol Checker (PET41) werkt intern met sets. Vandaar dat op het eerste gezicht maar 256 state rules en 256 trigger rules verwerkt kunnen worden. Echter blijkt dat deze niet onafhankelijk beschouwd mogen worden; de grootte van de benodigde datastructuren hangen af van het aantal state en trigger rules. Indien bijvoorbeeld 16 trigger rules aanwezig zijn, kunnen tegelijkertijd maar 40 state rules in het regelbestand opgenomen zijn. Neemt het aantal trigger rules toe tot 32, dan mogen nog slechts 24 State-rules gedeclareerd zijn.
- Het eigenlijke SYMPLEXYS expertsysteem (SIM41) kan maximaal ±3000 regels in de arrays onderbrengen.

Probemen ontstaan voornamelijk bij het controleren van grote knowledge bases. Afgezien van de hoeveelheid geheugen die de controle-tools nodig hebben, levert ook de run-tijd van de controle-tools problemen op wanneer de rule-bases groot worden.De tijdcomplexiteit van de controle-tools bedraagt soms $(\#regels)^3$.

Oplossingen:

- Door de rules in dynamische arrays op te slaan worden de beperkende limieten voor zowel RUC41 als SIM41 sterk verruimd.
  De grootste beperking: het maximum van 180 regels dat door CHK41 wordt opgelegd kan niet worden verholpen d.m.v. een pointer naar een array. Indien deze limiet omhoog getild dient te worden, dan zal een andere oplossing dan het huidige 2-dimensionale array noodzakelijk zijn. Een dergelijke wijziging zal waarschijnlijk duur betaald moeten worden in termen van executietijd en programma complexiteit.
  Wanneer de verschillende arrays van de Protocol Checker dynamisch gemaakt worden, verruimen de limieten enigzins. Wanneer een andere datastructuur in de Protocol Checker genomen wordt, dan is 256 als maximaal aantal state en trigger rules haalbaar (zie bijlage 6).
- In combinatie met bovenstaande verbeteringen kunnen kennismodulen (zie §3.1.1) tot grotere (controleerbare) regelbestanden, wanneer de kennismodulen afzonderlijk gecontroleerd kunnen worden.

3.3 Inventarisatie SYMPLEXYS codering

Een gedetailleerde bestudering van een programma gaat vooraf aan de fase waarin je tot oplossingen van beperkingen van het programma kunt komen. Afgezien van de beperkingen uit de vorige paragraaf kan bij inventarisatie van de source op een aantal andere aspecten van de codering worden gelet. Deze verschillende aspecten worden in de volgende paragrafen afzonderlijk toegelicht.

3.3.1 De SIMPLEXYS tools

Een SIMPLEXYS regelbestand wordt door een aantal SIMPLEXYS tools bewerkt. Het door de Rule Compiler gecompileerde regelbestand wordt gecompileerd tezamen met de Inference Engine, zodat het uiteindelijke expertsysteem ontstaat.

De verschillende tools met hun functie zijn:

- Rule Compiler.
  Dit programma heeft als invoer een regelbestand. Het regelbestand wordt door de Rule Compiler gecontroleerd op syntactische correctheid. Wanneer aan alle syntaxisvoorwaarden is voldaan, worden door de Rule Compiler verschillende uitvoerbestanden aangemaakt:
  - RUSES.QQQ bevat de declaraties uit de USES sectie van het regelbestand.
  - RINEX.QQQ bevat de procedures _InitG, _ExitG, _InitR, _ExitR.
  - RDODO.QQQ bevat de procedure _FDos waarin alle Do-code wordt opgenomen.
  - RTEST.QQQ bevat de functie _FTest met daarin alle tests die door Pascal uitgevoerd moeten worden.
  - RHIST.QQQ bevat alle history tests die zijn opgenomen in _FHist.
  - RINFO.QQQ bevat de gecompileerde regel- en protocolbeschrijving (in array-vorm).

  De file RInfo.QQQ is de eigenlijke beschrijving van het expertsysteem (zoals een gecompileerde beschrijving van de regels en de On-statements). Aan de hand van deze file wordt het regelbestand gecontroleerd door de Semantics en Protocol Checkers.
- Semantics Checker.
  RInfo.QQQ wordt met de Semantics Checker gecompileerd, zodat de Semantics Checker het regelbestand op logische fouten kan controleren.
- Protocol Checker.
  Ook de Protocol Checker wordt gecompileerd met RINFO.QQQ en controleert het protocol (de toestandsveranderingen) van het regelbestand.
- de Options Builder. Met behulp van dit kleine programma zijn de debugging opties van de Inference Engine in te stellen. Het programma genereert een optiebestand "OPTIONS.QQQ" die de Inference Engine wordt meegecompileerd.
- Inference Engine.
  Alle bestanden die door de Rule Compiler worden gegenereerd plus het optiebestand worden meegecompileerd met het programma van de Inference Engine zodat één executable ontstaat: het uiteindelijke expertsysteem.

De eerste drie programma's worden altijd na elkaar uitgevoerd. De Semantics en de Protocol Checker worden steeds opnieuw gecompileerd door een command-line compiler, omdat de file RINFO.QQQ meegecompileerd moet worden met deze tools. Dit is overbodig: de drie programma's zijn onder te brengen in één executable, door ze onder te brengen in verschillende units en delen van de datastructuren van deze programma's dynamisch te maken (zie §4.6).

Aan het telkens hercompileren van (delen van) de Inference Engine is echter niet te ontkomen. Immers kan in het regelbestand Pascal code opgenomen zijn, die door het uiteindelijke expertsysteem gebruikt wordt. Daar de Rule Compiler geen Pascal compiler in zich herbergt, moet de Pascal code, die de Rule Compiler uit het regelbestand verzamelt,

gecompileerd worden door een Pascal compiler. Deze code moet vervolgens gelinked worden aan de overige componenten van het expertsysteem: de gecompileerde regelbeschrijving, de Inference Engine en de debugging opties.

3.3.2 C/Pascal

De huidige situatie is dat twee van de SIMPLEXYS toolboxen ontwikkeld zijn: een C- en een Turbo Pascal versie.
De Pascal-versie van SIMPLEXYS bestaat voornamelijk omwille van de onderhoudbaarheid van het programma. Studenten aan de TUE krijgen Pascal als een verplicht onderdeel tijdens de studie, zodat het gemakkelijk is voor studenten om aan SIMPLEXYS te werken. De overige voordelen van Pascal zijn:
- compileert/linkt veel sneller dan C doordat Pascal een one-pass compiler is;
- strikte type-controle;
- minder haakjes en complexe constructies dan bij C binnen één programmaregel. De onderhoudbaarheid en leesbaarheid van een Pascal programma is dus meestal beter. Daarmee is niet gezegd dat in C géén leesbare programma's te schrijven zijn; dit vereist echter discipline van de programmeur.

De voordelen van C boven Pascal:
- genereert sneller code (±50% beter dan Turbo Pascal[1]);
- niet sterk platform gebonden.

De eerste drie tools die het aangeboden regelbestand parsen en controleren, zijn nu ook in beide programmeertalen geschreven. Ik ben van mening dat één versie kan volstaan: de Rule Compiler behoeft alleen die .QQQ files aan te maken voor de betreffende programmeertaal. De gecompileerde regelbeschrijving kan door één Rule Compiler voor zowel C als Pascal aangemaakt worden.

Dit ene programma (met daarin geïntegreerd de Rule Compiler, Semantics Checker en Protocol Checker) kan in of C of Pascal geprogrammeerd worden. Ondanks de voordelen van Pascal is waarschijnlijk C toch de betere keus, omdat C-code beter overdraagbaar is. In de bijlagen 5 en 6 wordt geschetst hoe de Semantics Checker en de Protocol Checker in de Rule Compiler ingebouwd moeten worden.

Ook de Inferencing Engine kan in slechts één programmeertaal geïmplementeerd worden, wanneer de code van een conventionele programmeertaal van een SIMPLEXYS expertsysteem gecompileerd kan worden tot een DLL file (zie bijlage 2).

---

[1] Er is echter een Turbo Pascal compatible compiler die snellere code genereert. De prijs is dat het compileren dan ongeveer acht keer langer duurt. In Turbo Pascal is gekozen voor snel compileren zonder zich al te druk te maken over het al dan niet optimaal zijn van de te genereren code. Snelle compilatie is zeer prettig voor programmeurs. Een optimaliserende compiler zou alleen voor eindprodukten gebruikt kunnen worden.

3.3.3 Beveiliging

Afgezien van de Rule Compiler is het noodzakelijk dat de sources van de overige tools meegeleverd moeten worden aan ontwikkelaars van op SIMPLEXYS gebaseerde expertsystemen. Aan het verspreiden van de source-code zijn nadelen verbonden:

- De ontwikkelaar moet altijd wachten op compilatie van de Semantics en Protocol Checker.
- Sources zijn niet te beveiligen tegen illegaal kopiëren; dit in tegenstelling tot .EXE files.
- De ontwikkelaars veranderen misschien de sources, zodat er meerdere versies van SIMPLEXYS in omloop zijn. Wanneer er een nieuwere versie van het origineel komt, moeten de ontwikkelaars eigen versies veranderen.

Een voordeel van het verspreiden van de source-code is:

- Er zal feed back op de source-code van de gebruikers van de tools zijn, die direct aanleiding kunnen geven tot verbeteringen van die source-code.

In §3.3.1 kwam naar voren dat niet ontkomen kan worden aan het telkens compileren van de Inference Engine wanneer een expertsysteem gegeneerd wordt. Programmacode van de Inference Engine waarin geen conditionele compilatie voorkomt, behoeft niet als source verspreidt te worden. Deze code kan in gecompileerde vorm aan ontwikkelaars worden verstrekt (in '.TPU'-files).

3.3.4 Leesbaarheid van de code

De programmacode is goed leesbaar. Er is niet teveel commentaar opgenomen; de code is van zichzelf duidelijk. Appendix 4 "The Inference Engine's main data structures" uit [BLO90] is nodig naast de code om inzicht te krijgen hoe de Rule Compiler maar vooral de Inference Engine werkt.

Jammer is echter dat veel namen van variabelen erg cryptisch zijn. Enkele voorbeelden:

- _R (i.p.v. bijv. RuleValue)
- _F (i.p.v. NumberOfOnStatements)

De omschrijving achter het voorbeeld had beter gebruikt kunnen worden in de programma-code, zodat iemand sneller in de code thuis kan raken (de type-tijd maakt immers geen belangrijk deel uit van de ontwikkeltijd van een programma).

3.3.5 Snelheidsaspecten

Omdat een SIMPLEXYS programma in een real-time omgeving moet kunnen functioneren, kijk je tijdens het bestuderen van de code naar de efficiëntie van de codering. Hoewel in [BLO90] gesteld wordt, dat nog niet de moeite is genomen om de source te optimaliseren, zie ik maar een beperkt aantal mogelijke verbeteringen:

- Case-statements kunnen geoptimaliseerd worden. In een case-statement kunnen het beste de cases die veelvuldig voorkomen het eerst worden opgenomen in het case-statement. Een case-statement wordt nl. lineair van boven naar beneden afgelopen, totdat de juiste case gevonden is.

Het volgende programma fragment trof ik aan:

```
case _ruletype [rule] of
  _fact  : fatal_error ('trying to assign to a FACT rule');
  _state : fatal_error ('trying to assign to a STATE rule');
   .
   .
```

In bovenstaand case-statement staan allereerst twee items die in een correct functionerend expertsysteem nooit voor zullen komen. Optimaal zou zijn deze items als laatste te plaatsen in het case-statement.

- Avoid duplicate information. Deze basisregel van het programmeren resulteert meestal niet of nauwelijks in verbeteringen van de executietijd van een programma. Het is alleen een goede programmeerstijl ter voorkoming van fouten. Wordt iets veranderd, dan behoeft slechts één copy veranderd te worden. Bestaan er meerdere duplicaten, dan kunnen inconsequenties ontstaan, wanneer één copy bij verandering van het programma wordt overgeslagen. Duplicate code komt in SIMPLEXYS voor wanneer strings bijvoorbeeld naar het scherm geschreven worden, terwijl diezelfde string ook naar bijvoorbeeld een error-file geschreven moet worden[1].
- Het kopiëren van datastructuren van hetzelfde type die consecutief in het geheugen worden opgeslagen, kan aanzienlijk versneld worden. In plaats van element voor element kopiëren van bijvoorbeeld array elementen, kan dit in Turbo Pascal in één statement:

  ```
  move(Source,Dest,SizeOf(Source));
  ```

  Deze instructie voert een verplaatsing van gegevens in geheugen uit, middels één machinetaalinstructie (tijdens de verplaatsing is het niet nodig instructies uit het werkgeheugen op te halen). Bij het kopiëren van arrays levert dit een snelheidswinst van een factor 6 op[2].
- In Turbo Pascal is ook een instructie opgenomen om consecutieve datastructuren te initialiseren:

  ```
  fillchar(_Busy,SizeOf(_Busy),ord(false));
  ```

  De fillchar instructie plaatst sizeof(_Busy) keer false in het geheugen, beginnend op de eerste geheugenplaats die _Busy in beslag neemt. Ook de procedure fillchar wordt door één machinetaalinstructie uitgevoerd.

De bovengenoemde verbeteringen sorteren echter een gering effect (hooguit 5% snelheidswinst mag worden verwacht). Het optimaliseren van de case-statements (door de meest voorkomende cases bovenaan in een case-statement te plaatsen) is een geringe moeite

---

1. Het bestaan van C en Pascal versies van de Rule Compiler, Semantics Checker en Protocol Checker is ook een vorm van duplicate information (zie 3.3.2).
2. De snelheidswinst is als volgt getest:

```
uses USubs; { Supplies Cls and Timer routines }
var A, B : array[1..10000] of byte;
    I,J  : word;
begin
  Cls; Timer(true);
  for J:=1 to 100 do
  move(A,B,SizeOf(A)); { for I:=1 to 10000 do A[I]:=B[I]; }
  {fillchar(A,SizeOf(A),0); for I:=1 to 10000 do A[I]:=0; }
  Timer(false);
end.
```

en het voorkomen van (string-)duplicaten raad ik zeker aan. De laatste twee implementatieverbeteringen zijn Turbo Pascal afhankelijk (en kunnen daarom beter niet doorgevoerd worden).

SIMPLEXYS dankt zijn redeneersnelheid aan het gegeven dat niet naar regels gezocht hoeft te worden; de regels zijn immers gecompileerd en via een index te adresseren in een array [BLO90]. Toch is dit proces van compilatie van referenties niet tot het uiterste doorgevoerd. Vaak moet nog naar de juiste code voor het type Rule/Thelse of naar de juiste Test/Hist of Do gezocht worden. Essentiële procedures/functies die door het redeneermechanisme gebruikt worden waarin case-statements voor het adresseren van code gebruikt worden zijn:
- _FDos (#Dos);
- _FTest (#tests);
- _FHist (#history operators);
- _SetRule (6);
- _Thelse (15);
- _SkipExpr (11);
- _EvalExpr (11);
- _ThelseBy (6);
- _EvalRule (6).

Tussen haakjes is de lengte van het case-statement van de betreffende procedure/functie aangegeven. In [BLO90] §5.7 is gemeten aan een expertsysteem met Test-rules. De verwachting was dat de executietijd evenredig zou toenemen met het aantal Test-rules. Uit de verrichte meting bleek dat de executietijd iets meer dan evenredig toenam wanneer het aantal Test-rules groter werd. Dit is toe te schrijven aan de case die de test-code in _FTest adresseert.

3.3.6 Initialisatie code

Voorafgaand aan de inferencing worden verschillende initialisaties gedaan. Velden van regels (_History, _R, _S) worden geïnitialiseerd op basis van gecompileerde gegevens die de Rule Compiler uit het regelbestand heeft gehaald. Deze initialisaties hadden reeds verricht kunnen worden door de Rule Compiler. Het voordeel is dat deze initialisatie code niet hoeft te worden uitgevoerd door en opgenomen hoeft te zijn in de Inference Engine. Een voordeel van de huidige implementatie is wel dat alle initialisatiecode gebundeld bij elkaar is opgenomen in de Inference Engine.
Geheel overbodig is het array _IValue, dat de initiële waarden van de regels voorafgaand aan de inferencing bevat. Deze informatie had direct in _R opgenomen kunnen worden.

3.3.7 Debugging opties

De debugging opties die de Options Builder saved in OPTIONS.QQQ zijn DEFINE directives. Bepaalde debugcode wordt al dan niet in het expertsysteem meegecompileerd, afhankelijk van de defines die gevonden worden in OPTIONS.QQQ.

De debugging opties zijn dus niet tijdens de executie van het expertsysteem te veranderen. Worden de debugging options veranderd dan moet de Inference Engine met de regelbeschrijving en de verandere debugging opties opnieuw gecompileerd worden.
Een marginale opoffering van executietijd maakt het mogelijk dat debugging opties tijdens executie van een expertsysteem veranderd kunnen worden.
Vervang bijvoorbeeld:

```
{$IFDEF DEBUG2}
 _show_applied ('NOT', u);
{$ENDIF}
```

Door:

```
if Debug2 then ShowApplied('NOT',u);
```

De programmafragmenten die conditioneel gecompileerd werden, zijn vervangen door if-statements. De executietijd verslechtert nauwelijks door het uitvoeren van eenvoudige if-statements. De code-size neemt wel toe, daar alle debugcode altijd meegecompileerd wordt. Via hot-keys en Do-code zouden debug-variabelen tijdens executie van het expertsysteem kunnen veranderen.

3.3.8 Optimalisatie redundancies.

In hoofdstuk 6 van [BLO90] wordt de Semantics Checker beschreven. Verschillende redundancies, die in een SIMPLEXYS regelbestand op kunnen treden, worden gesignaleerd. Mogelijke optimalisaties worden niet teruggevoerd in het uiteindelijke expertsysteem. Deze zijn:

- partiële tautologieën
- eliminatie van Fact rules
- equivalente rules kunnen verdwijnen

Wanneer de Semantics Checker in de Rule Compiler geïntegreerd zou zijn, kan de Semantics Checker deze optimalisaties uitvoeren door de objectstructuur van de regels te optimaliseren voordat het regelbestand in gecompileerde vorm weggeschreven wordt naar disk. Indien in de bestaande versie van SIMPLEXYS dergelijke optimalisaties zouden worden doorgevoerd, dan zal de Semantics Checker de gecompileerde regelbeschrijving met de wijzigingen moeten kunnen saven op disk.

3.3.9 Redeneren met tijd

SIMPLEXYS in de huidige vorm heeft slechts beperkte mogelijkheden om te redeneren met tijd. Van een regel wordt slechts bijgehouden hoe lang een regel reeds TR is. Is een regel niet meer TR, dan is niet meer in relatie tot de tijd met die regel te redeneren. Niet achterhaald kan worden hoe lang een regel reeds onbekend (PO) is. Hoe lang een regel reeds FA is, is te bepalen door een extra regel toe te voegen, die door thelses invers wordt gehouden aan de regel waarvan we willen weten hoe lang die regel reeds FA is. Dit toevoegen van een extra regel is weinig elegant, en vooroorzaakt een toenemend geheugengebruik en een toenemende run-tijd van de Semantics Checker.

Het redeneren met tijd kan krachtiger door een aantal variabelen aan een regel toe te voegen i.p.v. de History counter die nu in gebruik is:

```
var
   TimeTR : longint;  {update when Value=TR}
   TimePO : longint;  {update when Value=PO}
   TimeFA : longint;  {update when Value=FA}
```

Een vraag of regel R1 reeds langer dan 30 seconden FA is kan als volgt in de syntaxis worden opgenomen:

```
not(R1) > 30
```

Het antwoord op een dergelijke vraag kan zijn:

- FA, wanneer Time-TimeTR kleiner of gelijk aan 30 seconden.
- PO, wanneer Time-TimeTR groter dan 30 seconden is en Time-TimePO kleiner of gelijk aan 30 seconden is.
- TR, wanneer Time-TimeTR en Time-TimePO groter dan 30 seconden zijn.

De relaties om te bepalen of een regel TR of PO is gedurende het afgelopen tijdsinterval, zijn op soortgelijke wijze te implementeren. De interpretatie van Poss(R1) > 30 is: Is regel R1 gedurende minstens 30 seconden onbekend?

Bovengenoemde suggestie is nog niet geïmplementeerd. Wanneer dit gedaan wordt moet gelet worden op initialisatie en het updaten (tussen twee runs) van de drie tijdvariabelen. Wanneer na een run een regel UD *(undefined)* blijkt te zijn, dan moeten de drie tijdvariabelen alle gezet worden op de huidige tijd. Ditzelfde geldt voor een regel die initieel geen waarde heeft (de huidige tijd is dan 0).
Wanneer een regel wel een beginwaarde heeft (d.m.v. een INITIALLY statement of een regel met een default-beginwaarde), dan moet de tijdvariabele die hoort bij de desbetreffende waarde van een regel geïnitialiseerd worden op -1. De overige twee tijdvariabelen moeten een tijdstip verder terug in de tijd aanduiden (-2).
Fact regels vormen de uitzondering op de tijd-initialisatie zoals die hier beschreven is. De tijdvariabele die behoort bij de waarde van een Fact regel, wordt geïnitialiseerd op -1000000. Dit duidt aan dat de Fact regel de desbetreffende waarde altijd heeft gehad. De overige twee tijdvariabelen moeten de waarde -1000001 krijgen (een tijdstip vroeger in het verleden).

3.4 Codering andere expertsystemen

Tijdens het bestuderen van andere expertsystemen viel mij op, dat naast expertsystemen die in LISP danwel Prolog geprogrammeerd zijn, ook veel expertsystemen in een object georiënteerde programmeertaal geïmplementeerd zijn. Bijvoorbeeld frame based expertsystemen zijn een soort van objectgeoriënteerde omgevingen. Ook bij blackboard systemen is er sprake van objecten die op het blackboard staan.
In SIMPLEXYS zijn de regels en bijvoorbeeld de On-statements te interpreteren als objecten.

Om over objectgeoriënteerde expertsystemen te kunnen lezen heb ik mij objectgeoriënteerd programmeren eigen gemaakt[1]. Dat expertsystemen veelal object georiënteerd geprogrammeerd zijn komt, doordat voordelen als inheritance en polymorfisme[2] van objectgeoriënteerd programmeren binnen expertsystemen veelvuldig benut kunnen worden. Wanneer een expertsysteem objectgeoriënteerd geprogrammeerd is, ontstaat veelal een overzichtelijke data- en programmastructuur wanneer de (interne) objecten cognitief goed aansluiten bij fysieke objecten die voor te stellen zijn (regels, On-statements). Doordat methods van objecten slechts een lokaal karakter hebben en weergegeven kunnen worden in inheritance diagrammen (zie bijlage 4), wordt een overzichtelijke programmastructuur mogelijk.

Een interessante vraag is of deze voordelen ook voor SIMPLEXYS gelden, wanneer dit objectgeoriënteerd geprogrammeerd wordt. Het converteren van de huidige implementatie van SIMPLEXYS naar een objectgeoriënteerde implementatie is alleen zinvol als er duidelijke voordelen zijn, zonder dat de voordelen van de huidige implementatie verloren gaan:

- Verlies van redeneersnelheid is zeker ongewenst voor een real-time systeem.
- De eenvoud van coderen zou behouden moeten blijven.
- De zelfde syntaxis (met bijbehorende semantiek) van de regelbestanden moet door de objectgeoriënteerde implementatie verwerkt kunnen worden.

Een ander voordeel dat geëist moeten worden om te kiezen voor een objectgeoriënteerde implementatie van SYMPLEXYS is, dat de aangedragen beperkingen uit de vorige paragrafen eenvoudiger te verhelpen zijn indien SIMPLEXYS objectgeoriënteerd geprogrammeerd zou zijn.

Het volgende hoofdstuk beschrijft de objectgeoriënteerde implementatie van SYMPLEXYS. In hoofdstuk 5 wordt de objectgeoriënteerde implementatie vergeleken met de huidige implementatie.

---

1. Objectgeoriënteerd programmeren behoort naar mijn inzicht opgenomen te zijn in de studies Informatica en Informatie Techniek. Afgestudeerden van deze richtingen zullen naar alle waarschijnlijkheid in hun latere werk zeker te maken krijgen met objectgeoriënteerd programmeren. Het mag de TUE aangerekend worden dat dit nog niet het geval is.
   Een opmerking van dezelfde strekking geldt ook voor het programmeren in C. Dat Pascal zich uitermate goed leent voor educatieve doeleinden wil ik niet bestrijden (sterker nog: ik vind het zelfs spijtig dat C (i.p.v. Pascal) in de wereld zo wijd verspreid is).
2. zie bijlage 1.

4 Objective SIMPLEXYS

Een aanzet om te ontdekken wat de invloeden van objectgeoriënteerd programmeren op SIMPLEXYS zijn, is het als een objectgeoriënteerd programma implementeren van de Inference Engine. De overige tools (de Rule Compiler, Semantics en Protocol Checkers) kunnen in eerste instantie onaangetast blijven. Een eenvoudig te schrijven translator vertaalt het door de Rule Compiler gecompileerde regelbestand in objectstructuren.
Middels deze beperkte stap wordt ervaring opgedaan met objectgeoriënteerd programmeren. De voor en nadelen van een objectgeoriënteerde implementatie van SIMPLEXYS kunnen na deze beperkte stap geëvalueerd worden. Zo kan de opgedane kennis en ervaring als basis dienen bij de besluitvorming voor toekomstige uitbreidingen zoals die in het voorgaande hoofdstuk aangedragen zijn.

De gedachtengang achter objectgeoriënteerd programmeren is tamelijk onafhankelijk van de gebruikte programmeertaal. Voor een lezer die niet vertrouwd is met objectgeoriënteerd programmeren, is het raadzaam eerst bijlage 1 over objectgeoriënteerd programmeren te lezen. Niet alleen worden de implementatie details van Turbo Pascal met betrekking tot objectgeoriënteerd programmeren uitgewerkt, maar is bovendien een uitgebreid voorbeeld van een objectklassenhiërarchie ('Collections') in die bijlage opgenomen. Collections keren in de rest van dit hoofdstuk nog veelvuldig terug. Tevens wordt in de bijlage reeds melding gemaakt van de te verwachten voor en nadelen voor SIMPLEXYS, wanneer SIMPLEXYS objectgeoriënteerd geprogrammeerd wordt.

De oorspronkelijke implementatie heeft duidelijk model gestaan voor de "vertaling" van de bestaande implementatie naar een objectgeoriënteerde implementatie. Toch is er een aantal opmerkelijke wijzigingen:

- I.p.v. gebruik te maken van globale structuren (bijvoorbeeld door alle thelses van alle regels in één globaal array onder te brengen), is gekozen voor lokale opslag van bijvoorbeeld thelses bij een regel.
- Gedrag is gekozen als basis voor de objectstructuur; dus niet alleen de datastructuren. Voorheen werd het essentiële gedrag vastgelegd in de volgorde waarin procedures aangeroepen werden. Dit procedurele karakter is niet verdwenen, maar een deel van dit procedurele karakter is nu overgenomen door de methods van de verschillende objecten.

De eerste wijziging ligt voor de hand en is op zich niet ingrijpend. Een mogelijk probleem dat bij grote regelbestanden voor het array waarin de thelses opgeslagen worden (_TTStore) meer dan 64K nodig zou zijn, kan zich zodoende niet meer voordoen. De tweede wijziging gaat een stap verder en maakt een één op één vertaling onmogelijk. Deze stap is dus een stuk lastiger maar bewerkstelligt, zoals later zal blijken, een betere uniformiteit.

4.1 De objectstructuur

De volgende objectstructuur is tot stand gekomen door het gedrag "evalueren" voorop te stellen. Geëvalueerd worden in een SIMPLEXYS kennissysteem:
- regels;
- expressies (bij Eval-regels);
- operatoren binnen de expressies.

De evaluatie van een expressie bestaat echter uit de evaluatie van een aantal regels of operatoren. Een aparte objectklasse voor expressies is er dus niet.

Een drietal operatoren zijn te onderscheiden:
- operatoren met een enkelvoudig argument (NOT, MUST, POSS);
- operatoren met twee argumenten (AND, UAND, OR, UOR, ALT);
- history operatoren.

Deze observaties leiden tot de volgende objectklassenhiërarchie:

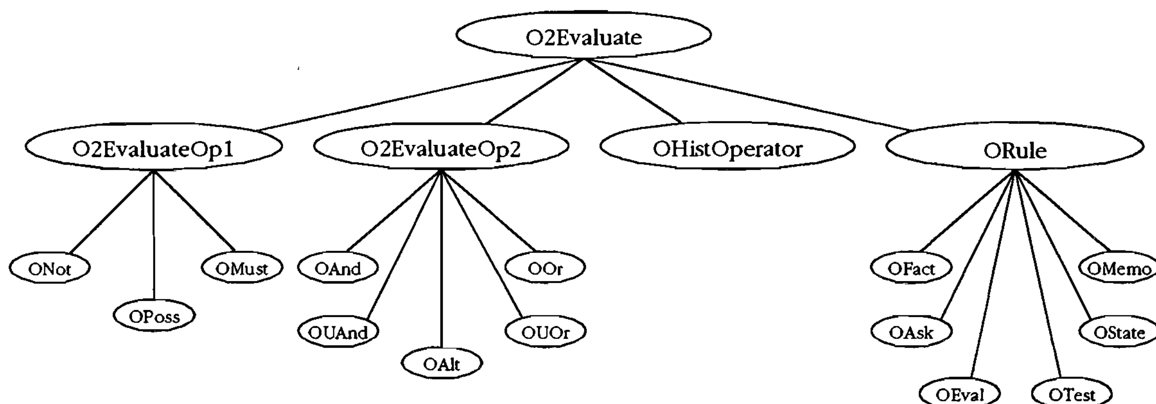

Fig. 3: Objectklassenhiërarchie van te evalueren objecten.

De letter 'O' die voorafgaat aan de naam van de objectklassen moet gelezen worden als 'Object'. Om bovenstaande figuur leesbaar te houden zijn de namen van de bladeren afgekort. De operatoren hebben in de code de extensie 'Operator' (dus: OUAndOperator i.p.v. OUAnd) en de Rule objecten hebben de extensie 'Rule' (dus: OFactRule i.p.v. OFact).

In figuur 12 van bijlage 4 is het inheritance diagram voor de objectklassehiërarchie uit figuur 3 terug te vinden.

Naast het gedrag evalueren is ook het uitvoeren van de thelses van een regel als gedrag te onderscheiden. Er zijn drie verschillende soorten thelses die ieder verschillend uitgevoerd moeten worden. De drie soorten thelses zijn:
- thelses die een waarde toekennen aan een andere regel;
- thelses die Pascal/C uitvoeren;
- thelses die als doel stellen dat bepaalde regels geëvalueerd worden.

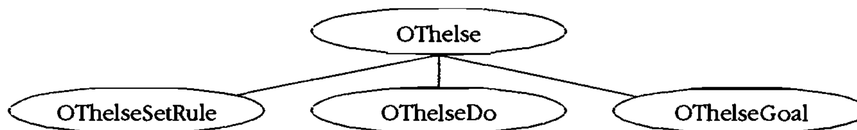

Fig. 4: Objectklassenhiërarchie van thelse-objecten.

4.2 Datastructuren

Met de objectstructuur uit de vorige paragraaf is pas de helft van de vertaling gerealiseerd. Een object van één van de klassen uit figuur 3 zal toebehoren aan één of meerdere datastructuren (bijvoorbeeld een Fact-rule kan toebehoren aan een verzameling van alle regels, maar ook aan de verzameling Fact-rules).

De twee belangrijkste dataverzamelingen vallen bij oppervlakkige inventarisatie van de datastructuren van de SIMPLEXYS implementatie direct op :
- een verzameling van On-statements;
- een verzameling rules.

Nadere inspectie van de code geeft voor deze verzamelingen de volgende specialisaties:
- De verzameling van On-statements is geen echte verzameling. De volgorde van de On-statements is van belang: de volgorde die wordt aangetroffen in het regelbestand is essentieel en moet behouden blijven in de datastructuur.
- De verzameling van rules is in de originele implementatie gesorteerd (naar rule-type). De sorteervolgorde was: Fact, Ask, Test, Eval, Memo en State. Deze volgorde is niet zozeer essentieel; wel dat alle rules van één bepaald type bij elkaar staan. Daar vaak bepaalde bewerkingen/operaties worden uitgevoerd op alle regels van hetzelfde type (bijvoorbeeld het updaten van de Memo-rules). Het uitvoeren van een bewerking/ operatie op regels van één type geschiedt zodoende zonder zoekoperaties.
  Voor het uiteindelijke expertsysteem kunnen aparte verzamelingen voor de verschillende rule-typen aangemaakt worden. Parallel naast deze zes verzamelingen kan eventueel een verzameling bestaan die alle regels van het expertsysteem beheert[1].

[1] Tijdens het parsen van het regelbestand is bovendien een verzameling met rulereferenties noodzakelijk, daar regels in een SIMPLEXYS regelbestand forward gerefereerd kunnen zijn. Uit de forward referentie is niet op te maken welk regeltype de forward gerefereerde regel zal hebben. Dus is nog niet bepaald in welk van de zes regelverzamelingen de regel uiteindelijk geplaatst wordt. Deze verzameling referenties zou het beste alfabetisch gesorteerd kunnen zijn, zodat tijdens het parsen een referentie naar een regel snel gevonden kan worden.

4.2.1 On-Statements

De On-statements zijn op te vatten als een verzameling met een volgorde. Een collection is een geschikte structuur om de On-statements in op te slaan. Afzonderlijke On-statements vormen de items in de collection van On-statements. De volgorde van de On-statements die de Rule Compiler tijdens het parsen van het regelbestand aantreft, wordt bewaard in de collection. Een volgend On-statement dat geparsed wordt, wordt achteraan in de collection toegevoegd.

Eén afzonderlijk on-statement heeft:
- een trigger rule
- een "FromCollection"
- een "ToCollection"

De From- en ToCollections van een On-statement zijn verzamelingen regels. Deze verzamelingen kunnen als collections geïmplementeerd worden, waarin de itempointers direct naar de betreffende regels wijzen.

De volgende type- en var-declaraties volstaan om On-statements op te kunnen slaan:

```
type
    OOnStatement = Object(TObject)
                     Trigger        : PORule;
                     FromCollection : OCollectionNoFreeItem;
                     ToCollection   : OCollectionNoFreeItem;
                   end;

    OOnCollection = object(TCollection)
                       procedure UpdateStateRules;
                     end;

var
   OnStatements : OOnCollection;
```

Listing 1: Type declaraties voor on-statements.

De method UpdateStateRules maakt deel uit van het interferentiemechanisme. UpdateStateRules wordt aangeroepen door het interferentiemechanisme en voert als voorheen na evaluatie van het regelbestand de mogelijke context-switches uit.

De volgende figuur geeft de datastructuren waar die Objective SIMPLEXYS gebruikt om On-statements op te slaan. De pijlen in de figuur beelden pointers uit en cursief is de naam van de objectklasse weergeven waarvan een structuur deel uitmaakt.

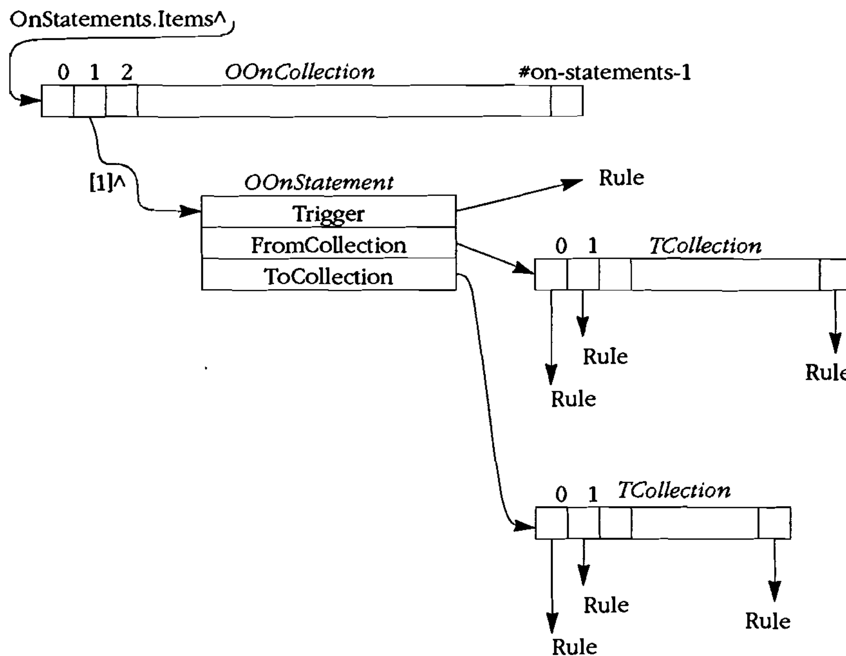

Fig. 5: Datastructuren On-statements.

4.2.2 Rules

Voor het opslaan van de regelverzamelingen worden zes collections gebruikt (één voor ieder regeltype). Iedere regel wordt op basis van zijn type ondergebracht in één van deze collections en in de algemene collection van rules:

```
var
   Rules       : ORuleCollection;
   FactRules   : OFactRuleCollection;
   AskRules    : OAskRuleCollection;
   TestRules   : OTestRuleCollection;
   EvalRules   : OEvalRuleCollection;
   MemoRules   : OMemoRuleCollection;
   StateRules  : OStateRuleCollection;
```

Listing 2: Type declaraties globale collections.

Het inheritance diagram van deze rule-collections is opgenomen in figuur 13 in bijlage 4. Daarin is weergegeven dat de objectklassen voor de zes gespecialiseerde verzamelingen subklassen zijn van ORuleCollection.

4.2.3 Datastructuren rule

De verzameling On-statements en de verzamelingen regels uit de vorige paragraaf beheren of refereren de regels. Een regel is de essentiële bouwsteen in SIMPLEXYS, daarom wordt de datastructuur van een regel in deze paragraaf uitvoerig beschreven.

De zes regeltypen worden afgeleid uit de superklasse ORule:

```
ORule = Object(O2Evaluate)
          Name      : PString;
          Text      : PString;
          Kind      : TRuleTypes;
          Value     : Bool;
          PrevValue : Bool;
          History   : longint;
          Busy      : boolean;
          Thelses   : TSortedCollection;
          ThelsedBy : TSortedCollection;

          function  Thelse   : Bool;
          function  ThelseBy : Bool;
          procedure SetRule(SetValue: Bool); virtual;
        end;
```

Listing 3: Superklasse regeltypen.

ORule legt algemene eigenschappen van regels vast.

Bovenstaande datavelden worden gebruikt voor:
- de (unieke) regelnaam;
- een tekst;
- het regeltype (Fact, Ask, Test, Eval, Memo, State);
- de waarde in de huidige evaluatie (voorheen _R);
- de waarde uit de vorige run (voorheen _S);
- HistoryCounter geeft aan wanneer de regel TR werd;
- een boolean die aanduidt dat de evaluatie van de regel gestart is;
- een verzameling Thelses van de regel, geïmplementeerd als een collection;
- een verzameling ThelsedBys, die bijhoudt welke regels een thelse hebben naar deze regel.

De methods van ORule worden gebruikt voor:
- Het uitvoeren van de thelses.
- Het bepalen van de waarde van deze regel via thelsebys, indien de regel door evaluatie de conclusie PO zou krijgen.
- Het toekennen van een waarde aan de regel, wanneer de waarde is bepaald door een thelse van een andere regel. ORule.SetRule is een abstracte method, die dus door de subklassen overruled moet worden.

Ondanks dat de method ThelseBy niet wordt gebruikt door Fact, Memo en State-rules, wordt deze method toch gedeclareerd in de superklasse van deze regels. Subklassen hoeven immers niet alle methods uit hun superklassen te gebruiken.
Zodoende kan dezelfde ThelseBy routine door de andere subklassen van ORule (voor Ask, Test en Eval regels) gebruikt worden.

De zes regeltypen hebben elk een eigen objecttype. De zes regeltypen overrulen alle ORule.SetRule en O2Evaluate.Evaluate:

```
OFactRule = object(ORule)
              function  Evaluate : Bool; virtual;
              procedure SetRule(SetValue: Bool); virtual;
            end;
OAskRule  = object(ORule)
              function  Evaluate : Bool; virtual;
              procedure SetRule(SetValue: Bool); virtual;
            end;
OTestRule = object(ORule)
              Test : TBoolFunction;
              function  Evaluate : Bool; virtual;
              procedure SetRule(SetValue: Bool); virtual;
            end;
OEvalRule = object(ORule)
              EvalExpr  : PO2Evaluate;
              function  Evaluate : Bool; virtual;
              procedure SetRule(SetValue: Bool); virtual;
            end;
OMemoRule = object(ORule)
              function  Evaluate : Bool; virtual;
              procedure SetRule(SetValue: Bool); virtual;
            end;
OStateRule = object(ORule)
               NewValue : boolean;
               function  Evaluate : Bool; virtual;
               procedure SetRule(SetValue: Bool); virtual;
             end;
```

Listing 4: Rule-typen.

Bij Test-regels is een functie-variabele toegevoegd, waarin een referentie naar de Test-code van de Test-rule bewaard wordt. Iedere Tes-rule heeft een directe referentie naar een aparte functie met de Test-code van die regel. Zonder een case-statement[1] uit te moeten voeren wordt de Test-code van een regel aangeroepen.
Een pointer naar een expressie (naar een O2Evaluate object) is bij Eval-regels toegevoegd, zodat de conclusie van de regel bepaald kan worden door de Evaluate method van de expressie aan te roepen.
State-rules hebben een extra variabele die de waarde van de regel voor de volgende run tijdelijk opslaat tijdens het uitvoeren van een context-switch.

De datastructuren van een regel zijn weergegeven in de volgende figuur. Als voorbeeld is een Fact-rule genomen, die opgenomen is in de collection FactRules.

1. Een case-statement in (Turbo) Pascal is eigenlijk een soort zoek-operatie. De juiste case wordt gevonden door de cases van het case-statement één voor één (en van boven naar beneden) af te lopen totdat de juiste case gevonden is.

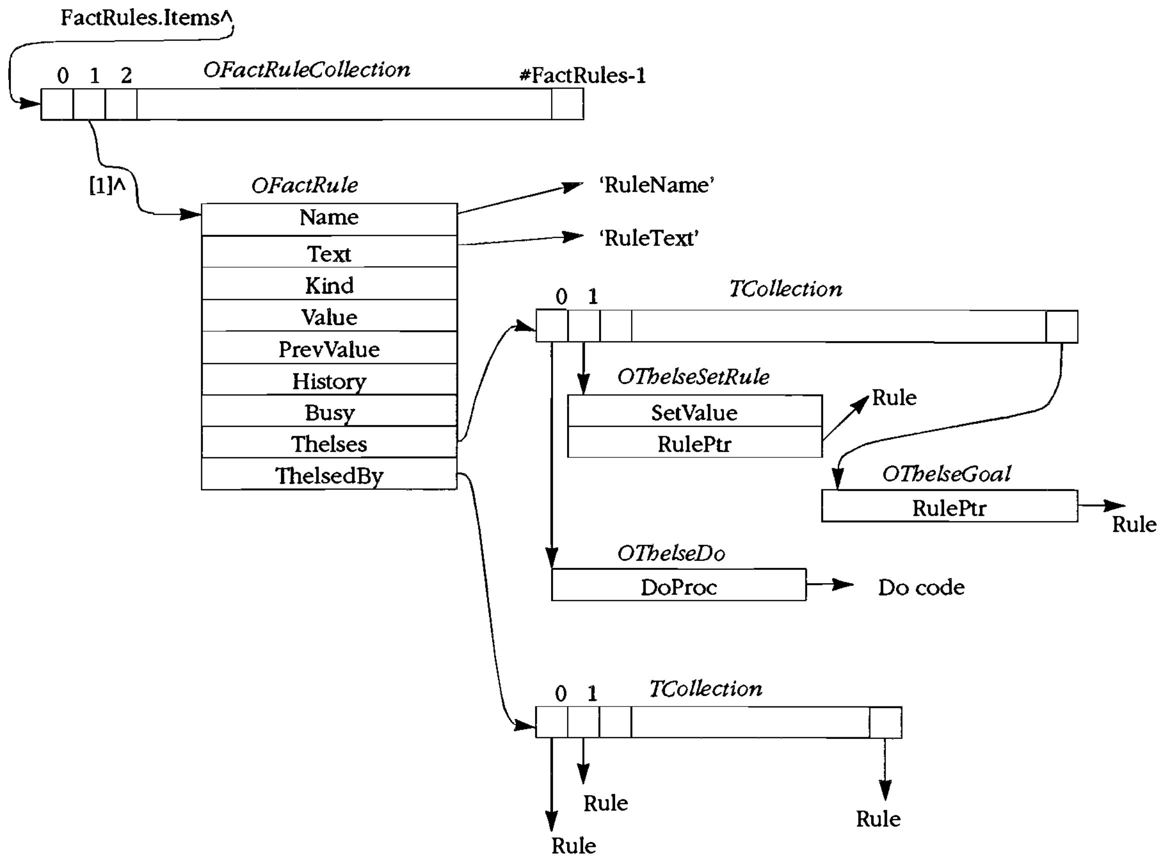

Fig. 6: Datastructuren regel.

In de figuur zijn als voorbeeld de drie typen thelses te zien.

4.2.4 Do-, Tes-t en History-code

Dos, Tests of Histories waren voorheen genummerd. Via een nummer werd de code behorend bij een Do-, Test- of History-operator uitgevoerd door resp. de functies _FDos, _FTest en _FHist. M.b.v. een case-statement werd de juiste code behorend bij het nummer in deze functies gevonden:

```
procedure _FDOS (_i: word);
begin
  case _i of
20000:
    begin
    dump('after evaluation of ANIM',false);
    end;
20001:
    begin
    writeln('*** I cannot identify your animal ***');
    end;
  end {case}
end;
```

Listing 5: Procedure _FDos.

In Objective SIMPLEXYS wordt alle code in aparte procedures/functies ondergebracht. Do procedures worden opgenomen in een unit DoProcs, Test functions in de unit FTests en History functions in de unit FHist. De Rule Compiler moet deze units aanmaken (naast RTEST.QQQ, RDODO.QQQ en RHIST.QQQ). Een voorbeeld van een dergelijke unit:

```
unit DoProcs;

interface

implementation

uses
    Objects, Types, SimLib;

procedure DoProc0; far;
begin
  dump('after evaluation of ANIM',false);
end;

procedure DoProc1; far;
begin
  writeln('*** I cannot identify your animal ***');
end;

begin
  new(DoCollection,Init(2,0));
  DoCollection^.Insert(@DoProc0);
  DoCollection^.Insert(@DoProc1);
end.
```

Listing 6: Unit met Do-code.

De volgende uitleg wordt gedaan aan de hand van de Do procedures uit bovenstaand voorbeeld. Dezelfde technieken worden ook gebruikt voor de code van Test en History functions.

Als voorheen is de Do code genummerd, alleen nu beginnen de nummers vanaf nul. De adressen van de Do procedures worden opgenomen in een collection (waarvan de items genummerd zijn overeenkomstig de procedures). Nadat dergelijke units aan de Inference Engine gelinkt zijn, kunnen voorafgaand aan het inferencing proces directe links vanuit de thelses naar de Do code worden gelegd. Aan de hand van de nummers, die de Rule Compiler nog steeds hanteert tijdens compilatie van het regelbestand, kunnen adressen van de procedures in de collection worden opgezocht en in de thelses worden gesubstitueerd i.p.v. de nummers. De Rule Compiler is gedwongen de code nog steeds te nummeren, omdat tijdens compilatie van het regelbestand de adressen van de Do procedures nog niet bekend zijn. De Do code wordt immers later gecompileerd.

Zoals de adressen van Do procedures worden ondergebracht in de (globale) collection DoCollection worden de adressen Test functions ondergebracht in FTestCollection en worden in HistCollection de adressen van de verschillende History functions bijgehouden.

4.3 Werking

De inferencing in de bestaande versie van SIMPLEXYS functioneert op basis van recursieve aanroepen van de procedures EvalRule, SetRule, Thelse en ThelseBy. Dit sluit nl. nauw aan bij de manier waarop de regels uit het kennisbestand met elkaar samenhangen. Bekijken we de volgende Eval-regel:

```
E1: 'Regel E1'
not R2
```

Wordt regel E1 geëvalueerd dan is het nodig R2 te evalueren. Recursie ligt dus voor de hand.

Objective SIMPLEXYS wijkt niet af van bovenstaande recursie. De recursie is echter uitgesmeerd over vele methods. De expressie is geïmplementeerd als pointer naar een te evalueren not-object. De Evaluate method van het not-object ontkent de waarde van zijn enige argument; een pointer naar wederom een te evalueren object. Deze pointer wijst naar de regel R2, die eveneens een Evaluate method heeft.

De gebruikte routines staan in listing 7 afgebeeld. De programmaregels waar de recursie plaatsvindt zijn vetgedrukt. OEvalRule.Evaluate wordt aangeroepen bij evaluatie van E1. De velden van E1 zijn binnen OEvalRule.Evaluate impliciet te adresseren dus is EvalExpr^.Evaluate een aanroep naar de Evaluate method van het object waar EvalExpr van E1 naar verwijst (in dit geval het not-object).
ONotOperator werkt identiek. Daar is de pointer Arg vergelijkbaar met de pointer EvalExpr van E1. De Evaluate method van R2 wordt aangeroepen. Indien R2 ook het type OEvalRule heeft, wordt wederom OEvalRule.Evaluate gebruikt met als impliciete parameter een pointer naar R2.

```
function OEvalRule.Evaluate;
var
   XValue : Bool;
begin
  Busy:=true;
  if Value=UD then
   begin
     if Debug1 then ShowProgress(2,@Self);  { starting evaluation of rule: }
     Value:=EvalExpr^.Evaluate;
     if not(Validate) then
       begin
         if Value=PO then Value:=ThelseBy;
       end
     else
       if Value=PO then Value:=ThelseBy
       else
         begin
           XValue:=ThelseBy;
           if (XValue<>PO) and (XValue<>Value) then
             NonFatalError(3,@Self);  { *** conflicting thelse to rule: }
         end;
     if (Value=TR) and (PrevValue<>TR) then History:=Time;
     if Debug0 then ShowValue(@Self,Value);
     Thelse;
     if Debug1 then ShowProgress(3,@Self);  { finished evaluation of rule: }
     if Value=UD then FatalError('OEvalRule.Evaluate returned UD value');
   end
  else if Debug1 then ShowProgress(8,@Self);  { obtained value      of rule: }
  Evaluate:=Value;
end;
```

```
function ONotOperator.Evaluate;
var
   u : Bool;
begin
  u:=Arg^.Evaluate;
  case u of
    FA : u:=TR;
    TR : u:=FA;
  end;
  if Debug2 then ShowApplied('NOT',u);
  if u=UD then FatalError('NOT returned UD value');
  Evaluate:=u;
end;
```

Listing 7: Recursieve evaluatie.

Op de volgende bladzijden is links de oude code van de routine _Thelse opgenomen. De rechter bladzijde geeft de objectgeoriënteerde implementatie van _Thelse weer. Vergelijken we beide implementaties dan valt direct op, dat één grote procedure is opgesplitst in meerdere (soms zelfs triviale) methods. De tijd, nodig voor het programmaontwerp in de objectgeoriënteerde versie, is gestoken in het ontwerp van de objectstructuur (zie de type declaraties).

De while J<>0 loop uit de routine _thelse wordt overgenomen door de for I:=0 to ... loop. Beide doorlopen de thelses van een regel. Duidelijk is, dat bij de objectgeoriënteerde implementatie het type thelse niet bepaald hoeft te worden d.m.v. een case-statement. I.p.v. van het case-statement wordt een directe call[1] naar de juiste Thelse uitgevoerd.
Een ander verschil zijn de parameters die worden meegeven: een regelnummer wordt expliciet meegegeven aan _evalrule. Dit is niet nodig in de objectgeoriënteerde code.

1. Deze calls naar methods zijn altijd far, i.e. kosten helaas meer tijd dan near calls.

```
procedure _thelse (rule: _sizeN; value: bool); {declared forward}
{execute thelses of ru
var
  ttstore_ptr:   0.._T;
  i, j:          word;
  current_value: bool;
begin
  ttstore_ptr := _ttindex [rule];
  if ttstore_ptr = 0 then exit; {rule has no thelses}
  if _R [rule] = UD then exit; {no thelses possible}
{$IFDEF DEBUG1}
  _show_progress (6, rule);
{$ENDIF}
  j := _ttstore [ttstore_ptr];
  while j <> 0 do
  begin
    inc (ttstore_ptr);
    i := _ttstore [ttstore_ptr];
    inc (ttstore_ptr);
    case j of
      _thentr:   if value = TR then _setrule (i, TR);
      _thenfa:   if value = TR then _setrule (i, FA);
      _thenpo:   if value = TR then _setrule (i, PO);
      _thengoal: if value = TR then
                   if not _busy [i] then
                     current_value := _evalrule (i);
{$IFDEF THERE_ARE_DOS}
      _thendo:   if value = TR then _FDOS (i);
{$ENDIF}

      _elsetr:   if value = FA then _setrule (i, TR);
      _elsefa:   if value = FA then _setrule (i, FA);
      _elsepo:   if value = FA then _setrule (i, PO);
      _elsegoal: if value = FA then
                   if not _busy [i] then
                     current_value := _evalrule (i);
{$IFDEF THERE_ARE_DOS}
      _elsedo:   if value = FA then _FDOS (i);
{$ENDIF}

      _ifpotr:   if value = PO then _setrule (i, TR);
      _ifpofa:   if value = PO then _setrule (i, FA);
      _ifpopo:   if value = PO then _setrule (i, PO);
      _ifpogoal: if value = PO then
                   if not _busy [i] then
                     current_value := _evalrule (i);
{$IFDEF THERE_ARE_DOS}
      _ifpodo:   if value = PO then _FDOS (i);
{$ENDIF}
    else {case}
      fatal_error ('invalid token in thelse list')
    end {case};
    j := _ttstore [ttstore_ptr]
  end {while};
{$IFDEF DEBUG1}
  _show_progress (7, rule);
{$ENDIF}
end;
```

Listing 8: Bestaande implementatie thelses.

```
type
    OThelse        = object(TObject)
                        Necessary : Bool;
                        procedure DoThelse; virtual;
                      end;
    OThelseDo      = object(OThelse)
                        DoProc : TProcedure;
                        procedure DoThelse; virtual;
                     end;
    OThelseGoal    = object(OThelse)
                        RulePtr : PORule;
                        procedure DoThelse; virtual;
                     end;
    OThelseSetRule = object(OThelse)
                        SetValue : Bool;
                        RulePtr  : PORule;
                        procedure DoThelse; virtual;
                     end;

procedure OThelse.DoThelse;
begin
  abstract;
end;

procedure OThelseDo.DoThelse;
begin
  DoProc;
end;

procedure OThelseGoal.DoThelse;
begin
  if not(RulePtr^.Busy) then RulePtr^.Evaluate;
end;

procedure OThelseSetRule.DoThelse;
begin
  RulePtr^.SetRule(SetValue);
end;

function ORule.Thelse;
var
   i          : integer;
   ThelseItem : POThelse;
begin
  if (Thelses.Count=0) or (Value=UD) then exit;
  if Debug1 then ShowProgress(6,@Self); { starting thelses    of rule: }
  for i:=0 to pred(Thelses.Count) do
    begin
      ThelseItem:=Thelses.At(i);
      if Value=ThelseItem^.Necessary then ThelseItem^.DoThelse;
    end;
  if Debug1 then ShowProgress(7,@Self); { finished thelses    of rule: }
end;
```

Listing 9: Object georiënteerde implementatie thelses.

In figuur 6 pag. 30 zijn de datastructuren van de drie thelse-typen te zien. Iedere thelse van een regel is opgenomen in de (gesorteerde) collection van die regel. De thelses worden uitgevoerd in de volgorde die de thelses hadden in het regelbestand.
De variabele 'Necessary' wordt gebruikt, om vergeleken te worden met de waarde van de regel. Necessary kan drie waarden aannemen:

- TR voor then... thelses.
- FA voor else... thelses.
- PO voor ifpo... thelses.

4.4 XLAT

In de inleiding van dit hoofdstuk wordt gesteld, dat door alleen de Inference Engine objectgeoriënteerd te programmeren, de voor en nadelen van objectgeoriënteerd programmeren m.b.t. SIMPLEXYS te kunnen inschatten. De Rule Compiler compileert het regelbestand echter tot een (niet objectgeoriënteerde) beschrijving. Een translator is dus nodig om deze gecompileerde regelbeschrijving uit RINFO.QQQ om te zetten in het objectgeoriënteerde formaat waarmee de objectgeoriënteerde Inference Engine werkt.
De objectgeoriënteerde Inference Engine en de translator gebruiken de volgende libraries (units), die de andere tools later kunnen gebruiken, wanneer ze objectgeoriënteerd geprogrammeerd worden:

- TYPES : declaratie van conventionele Pascal-typen en globale variabelen.
- SIMLIB : library met conventionele procedures en functies die door de overige componenten van het expertsysteem gebruikt worden. Deze library bevat o.a. Time, Dump en Ask routines.
- OTYPES : declaratie van de objectklassestructuur en de globale variabelen die een objecttype hebben. De gedefiniëerde methods worden in deze library geprogrammeerd (Turbo Pascal staat niet anders toe).
- EXPERT : een kleine library met als voornaamste procedure Infer.

Van bovenstaande libraries hoeven niet de sources verspreid te worden. De gecompileerde vorm (tpu files) volstaat.

In de code van de gemaakte translator (XLAT.PAS) is te zien dat veel case-statements gebruikt worden. Met deze case-statements wordt het regelbestand a.h.w. een stap verder gecompileerd, zodat case-statements niet meer nodig zijn in de objectgeoriënteerde Inference Engine.

In §4.2.4 is reeds vermeld, dat Do-, Test- en History-code in een andere formaat aan Objective SIMPLEXYS aangeboden moet worden dan voorheen. Dit andere formaat wordt niet gemaakt door de translator. Eenvoudiger was het de Rule Compiler uit te breiden, zodat de units uit §4.2.4 ook door de Rule Compiler aangemaakt worden[1].

Met behulp van de translator en de uitbreidingen van de Rule Compiler is het mogelijk uitgaande van hetzelfde regelbestand te experimenteren met zowel de bestaande als de objectgeoriënteerde versie van SIMPLEXYS. In het volgende hoofdstuk worden beide versies onderling vergeleken.

[1] Op dezelfde manier kan één Rule Compiler te werk gaan die geschikt is voor zowel Pascal als C code.

4.5 Objective Rule Compiler

In hoofdstuk 5 wordt geconcludeerd dat de doelstellingen van de objectgeoriënteerde implementatie van SIMPLEXYS voldoet aan de voorwaarden die voorafgaand aan de conversie gesteld werden. Nu de objectstructuur eenmaal bestond, was het een kleine moeite om ook de Rule Compiler objectgeoriënteerd te implementeren.

De objectgeoriënteerde Rule Compiler verzamelt alle Pascal-code in één unit genaamd 'pascode.pas'. Hierin worden de Uses, Decls, InitG, InitR, ExitR en ExitG secties van een SIMPLEXYS regelbestand opgenomen evenals alle Do-, Test- en History-code. De afzonderlijke files waarin de reeds bestaande Rule Compiler de Pascal-Code verzamelde, worden niet meer aangemaakt.

De objectgeoriënteerde Rule Compiler is nog niet in staat de gecompileerde (objectieve) regelstructuur op te slaan op disk. Dit maakt het noodzakelijk de controle-tools te integreren tot één applicatie (zie §4.6). Ook de Pascal-code die tijdens het compileren uit een regelbestand wordt gefilterd, moet gelinked worden aan de Rule Compiler om het expertsysteem te kunnen runnen, nadat de objectstructuur weer is opgebouwd in het werkgeheugen door opnieuw hetzelfde regelbestand te compileren.

Wanneer besloten wordt de objectgeoriënteerde implementatie van SIMPLEXYS door te zetten, zal de objectgeoriënteerde regelbeschrijving op disk bewaard moeten kunnen worden. Een losse applicatie die deze regelbeschrijving inleest en de Pascal-code bevat vormt dan een expertsysteem applicatie.

4.6 Integratie controle-tools in Rule Compiler

Zoals in de vorige paragraaf is vermeld, is de objectgeoriënteerde Rule Compiler niet in staat de objectgeoriënteerde regelbeschrijving, die tijdens compilatie opgebouwd wordt in het werkgeheugen van de computer, op te slaan op disk. De Semantics en Protocol Checkers moeten om deze reden geïntegreerd zijn in de Rule Compiler.

De datastructuren van beide controle-tools worden nu opgebouwd uit de objectgeoriënteerde regelbeschrijving i.p.v. uit de arrays die voorheen de gecompileerde regelbeschrijving vormden. De datastructuren van beide tools zijn dynamisch gemaakt, zodat geen geheugenruimte van te voren gereserveerd hoeft te worden. De geheugenruimte die gebruikt wordt voor het controleren van de semantiek, wordt weer vrijgegeven nadat de betreffende controles zijn uitgevoerd. Ditzelfde geldt voor de Protocol Checker, zodat naderhand zo veel mogelijk werkgeheugen vrij is voor het runnen van het expertsysteem.

De problemen die optreden bij het integreren van de Semantic en Protocol Checkers in de Rule Compiler, worden uitgewerkt in de bijlagen 5 resp. 6. Ook de oplossingen zijn in de bijlagen gegeven. De methoden uit de bijlagen kunnen ook voor de reeds bestaande versie

van SIMPLEXYS gebruikt worden, om tot integratie van de controle tools in de Rule Compiler te komen. Zodoende wordt te allen tijde vermeden dat de gecompileerde regelbeschrijving gesaved moet worden op disk, om vervolgens gecompileerd te worden te zamen met de Semantic en Protocol Checkers.

5 Objective SIMPLEXYS vs. SIMPLEXYS

In §3.4 werd gesteld, dat de goede eigenschappen bij conversie van de bestaande implementatie van SIMPLEXYS naar een objectgeoriënteerde behouden moesten blijven. Deze eigenschappen waren:

- de eenvoud van de codering;
- de redeneersnelheid.

Objective SIMPLEXYS is zodanig ontworpen dat de functionaliteit van het expertsysteem niet is gewijzigd. De regelbestanden zijn dus volledig compatibel.
Beide versies van SIMPLEXYS worden in dit hoofdstuk met elkaar vergeleken om na te gaan of Objective SIMPLEXYS geen afbreuk doet aan bovenstaande eigenschappen.

5.1 Complexiteit codering

Wat de complexiteit van de codering betreft kan gesteld worden, dat er zich een verschuiving van de complexiteit van het programma heeft voorgedaan. Uit de listings 8 en 9 van §4.3 bleek al, dat de complexiteit van objective SIMPLEXYS schuilt in de objectklassenhiërarchie. Alleen de declaratie van de objectklassen vergt in Objective SIMPLEXYS al ±250 programmaregels. Daar tegenover staat dat de methods veel eenvoudiger geworden zijn dan de vroegere procedures (soms zijn de methods zelfs triviaal).

De algoritmische werking van de inferencing is door het kiezen van andere data- en programmastructuur niet gewijzigd. Al met al mag geconcludeerd worden, dat Objective SIMPLEXYS niet essentieel complexer geworden is, hoewel deze mening misschien alleen gedeeld wordt door programmeurs, die dynamisch en objectgeoriënteerd programmeren als vaardigheden hebben.

Objective SIMPLEXYS heeft dus als nadeel, dat meer vaardigheden van de programmeurs vereist worden. Bijlage 1 over objectgeoriënteerd programmeren is in dit verslag opgenomen om dit probleem te verlichten. In een aantal bladzijden wordt objectgeoriënteerd programmeren volledig uitgewerkt. De voorbeelden in bijlage 1 zijn zodanig gekozen, dat ze aansluiten bij de codering van Objective SIMPLEXYS.

5.2 Verschillen in snelheid

Verschillen qua snelheid tussen twee implementaties van hetzelfde programma kunnen op twee manieren beoordeeld worden:
- door vergelijking van de implementatie;
- door het uitvoeren van tests.

Het plezierige aan vergelijking op basis van test-runs is, dat concrete resultaten verkregen worden (een meetbaar tijdverschil). Toch zijn enkele kanttekeningen bij het bepalen van snelheidsverschillen op zijn plaats:
- Een goede test test een variëteit van eigenschappen van een programma in een uitgewogen mix. Het opstellen van een dergelijke mix is buitengewoon lastig.
- Omwille van de moeilijkheid goede tests op te stellen worden eigenschappen van een programma afzonderlijk getest. Vele tests zijn nodig om de verschillende eigenschappen van een programma te testen. Een te rooskleurig beeld kan al snel verkregen worden door gunstige resultaten van bepaalde test(s) te generaliseren[1].
- Het is niet altijd zo, dat wanneer een kleine test 100 keer uitgevoerd wordt, hetzelfde resultaat oplevert, als het eenmalig uitvoeren van een 100 keer zo groot probleem. Een dergelijke conclusie mag alleen verondersteld worden op basis van kennis van het onderliggende probleem en programma.

Beter is de snelheid te vergelijken uitgaande van beide implementaties. Een meetverwachting voor tests die eventueel uitgevoerd gaan worden is dan te maken. Het vergelijken van de implementatie veronderstelt echter wel gedegen inzicht in de werking van beide implementaties en de werking van de gebruikte compiler.

Doordat Objective SIMPLEXYS is ontwikkeld uit de bestaande implementatie mogen geen grote snelheidsverschillen worden verwacht, waar een één op één vertaling van het programma heeft plaatsgevonden. Algoritmisch gezien is het programma nauwelijks gewijzigd. Alleen de procedure SkipExpr kon verdwijnen door de argumenten van expressies op te slaan in een boom i.p.v. consecutief in een array.

1. Aardige voorbeelden hiervan zijn bijvoorbeeld de advertenties van Borland. De snelheid van compileren wordt altijd (terecht) geroemd. De prijs dat de compilers niet echt optimale code aflevert, wordt dan verbloemd door te stellen dat de nieuwe compilers veel betere code afleveren dan de vorige versies. Als algehele indruk ontstaat dan een beeld, dat de Borland compilers snel zijn op alle gebieden.

De overige verschillen in run-tijd worden veroorzaakt doordat:

1. Case-statements zijn vervangen door directe adressering naar de juiste code (pointers).
2. If-statements die voor debug-code staan moeten in Objective SIMPLEXYS worden uitgevoerd (zie §3.3.6).
3. In de Objective implementatie van SIMPLEXYS worden de methods aangeroepen als far-calls. Dit kost een fractie meer run-tijd (en code) dan de near-calls naar procedures in de bestaande versie.
4. In Objective SIMPLEXYS worden minder expliciete parameters meegegeven aan de methods dan aan de procedures in de bestaande versie. Tegenover dit voordeel staat dat iedere method een impliciete parameter heeft (een verwijzing naar het object dat de method gebruikt). Het netto resultaat op de run-tijd zal door combinatie van beide effecten gering zijn, omdat in SIMPLEXYS weinig expliciete parameters (vaak alleen een regelnummer) worden meegegeven aan procedures.
5. In Objective SIMPLEXYS wordt vaker via een pointer (4 bytes) data geadresseerd dan met een array-index (2 bytes).
6. Evaluatie van lange expressies verlopen sneller doordat SkipExpr overbodig is geworden. Daar lange expressies zelden voor zullen komen in een goed gestructureerd regelbestand, is deze factor nauwelijks van invloed op snellere executie van regels.

Deze verschillen zijn nauwelijks van betekenis. Bij Objective SIMPLEXYS is er sprake van duidelijke winst bij punt 1, wanneer een case-statement veel cases bevat. Dit zou kunnen voorkomen bij SIMPLEXYS expertsystemen die veel Dos, Tests of Histories hebben.
Door de punten 2, 3 en 5 zal Objective SIMPLEXYS toch wat trager blijken dan de bestaande implementatie.

In bijlage 7 wordt concreet het snelheidsverschil van het redeneermechanisme tussen SIMPLEXYS en Objective SIMPLEXYS gemeten.
De voornaamste conclusies uit bijlage 7 zijn:

- Redeneren gaat in Objective SIMPLEXYS fractioneel sneller.
- Dos, Tests en Histories worden sneller afgehandeld, omdat de code niet gezocht behoeft te worden in Objective SIMPLEXYS.
- In Objective SIMPLEXYS duurt de overhead tussen twee runs langer. Dit komt niet op het conto van de context-switches die uitgevoerd moeten worden, maar op het initialiseren van variabelen van regels om die regels in de volgende run opnieuw te kunnen evalueren.
- Objective SIMPLEXYS ondervindt veel minder traagheid wanneer Range Checking geactiveerd is.

5.3 Verschillen in geheugengebruik

Het geheugengebruik van een programma valt uiteen in twee delen: het geheugen dat voor programmacode wordt gebruikt en het geheugen dat door datastructuren in beslag wordt genomen. Allereerst worden de grootten van de programmacodes vergeleken, daarna die van de datastructuren.
Voordat het geheugengebruik daadwerkelijk vergeleken wordt, is een relativerende opmerking van belang. In de huidige tijd is werkgeheugen goedkoop. Met de komst van DPM programma's (zie bijlage 3) kunnen in DOS programma's gebruik maken van de totale hoeveelheid werkgeheugen dat in een computer aanwezig is. Wanneer gekozen moet worden tussen bijvoorbeeld snellere of beter leesbare code i.p.v. kleine code, kan de keuze uitvallen ten nadele van kleine code.

5.3.1 Code-grootte

Het testen van programmagrootten is in tegenstelling tot het uitvoeren van snelheidstesten gemakkelijker. De testomstandigheden zijn gemakkelijker gelijk te krijgen. Een meting die verricht is, staat beschreven in bijlage 7. De voornaamste conclusie uit die test is dat de code van het redeneermechanisme van Objective SIMPLEXYS ongeveer 11% groter is dan SIMPLEXYS.

5.3.2 Geheugengebruik datastructuren

In Objective SIMPLEXYS wordt het regelbestand verder gecompileerd en er wordt meer met dynamische structuren gewerkt. De meeste referenties naar data zijn pointers (4 bytes), waarmee data direct geadresseerd kan worden. Voorheen werd data via indices (2 byte integers) in arrays geadresseerd. De datastructuren in Objective SIMPLEXYS gebruiken dus meer geheugen.
Ten eerste vergelijken we de globale structuren. In de bestaande versie van SIMPLEXYS werd alle data ondergebracht in globale arrays, waarin direct geadresseerd kon worden. Objective SIMPLEXYS heeft echter veel minder globale structuren. Alleen een aantal collections is globaal: collections die naar regels, On-statements, History operators, Test- en Do-code refereren. Als enige kunnen de globale arrays _first en _last uit de bestaande versie van SIMPLEXYS met deze collections vergeleken worden, de overige arrays bevatten namelijk de data, die vergelijkbaar is met de gegevens in de items die door de genoemde collections van Objective SIMPLEXYS beheerd worden.

In Objective SIMPLEXYS bestaan twaalf globale collections: voor ieder type regel één collection (totaal 6), een collection met referenties naar alle regels, een collection met alleen referenties naar namen van regels, een collection met on-statements en drie dynamische collections voor het beheren van Test, Do en History code. Iedere statische collection neemt 10 bytes geheugen in beslag plus 4*(#ItemsInCollection) bytes[1].

De volgende tabel vergelijkt het geheugengebruik voor het opslaan van de gegevens van een regel.

Tabel 3: Geheugengebruik regel.

<table>
<tr><th></th><th>SIMPLEXYS</th><th>Objective SIMPLEXYS</th><th>Verschil</th></tr>
<tr><td>_RuleName</td><td>(1 + MaxNameLen)</td><td>(4 + 1 +NameLen)</td><td>X</td></tr>
<tr><td>_RuleText</td><td>(1 + MaxTextLen)</td><td>(4 + 1 + TextLen)</td><td>X</td></tr>
<tr><td>_RuleType</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>_IValue</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>_R</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>_S</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>_History</td><td>4</td><td>4</td><td>0</td></tr>
<tr><td>_Busy</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>Thelses (# > 0)</td><td>4 + #Thelses * 4</td><td rowspan="2">10 + (#ThelseDos+#ThelseGoals) * 8 + #ThelseSetRule * 9</td><td rowspan="2">X</td></tr>
<tr><td>Thelses (# = 0)</td><td>2</td></tr>
<tr><td>ThelseBys (# > 0)</td><td>4 + #Thelses * 4</td><td rowspan="2">10 + #ThelseBys * 10</td><td rowspan="2">X</td></tr>
<tr><td>ThelseBys (# = 0)</td><td>2</td></tr>
<tr><td>EvalExpr EvalRule</td><td>2+#Arg1Operatoren*4 +#Arg2Operatoren*6</td><td>4+#Arg1Operatoren*4 +#Arg2Operatoren*8</td><td>2+ #Arg2Operatoren*2</td></tr>
<tr><td>EvalExpr ¬EvalRule</td><td>2</td><td>0</td><td>-2</td></tr>
</table>

Bovengenoemde aantallen bytes gelden per regel; #Thelses is dus het aantal Thelses van een regel, NameLen de lengte van de naam van de regel, enz. MaxNameLen en MaxTextLen zijn de maximale lengtes van de naam- resp. textstrings van alle regels.
_IValue is eigenlijk niet nodig in SIMPLEXYS, de initiële waarde kan i.p.v. via _IValue al direct in _R gezet worden. Bij Objective SIMPLEXYS bestaat _IValue nog steeds, om een tweede run van expertsysteem mogelijk te maken zonder opnieuw de regelbeschrijving te compileren.
In Objective SIMPLEXYS zijn de strings _RuleName en _RuleText dynamisch. Dit bespaart geheugenruimte t.a.v. de bestaande versie doordat niet voor iedere string MaxNameLen (resp. MaxTextLen) bytes gereserveerd worden.
In Objective SYMPLEXYS wordt alleen bij Eval-rules een pointer naar een expressie gedeclareerd. De bestaande versie van SIMPLEXYS gebruikte voor de andere regeltypen hiervoor onnodig twee bytes geheugen. Dit is in de bestaande versie van SIMPLEXYS te elimineren door het array _EVINDEX slechts voor het aantal Eval-regels te definiëren

1. Collections voor het beheren van de referenties naar Do, Test en History code en de collection met referenties naar de regelnamen zijn alleen nodig tijdens compilatie van het regelbestand. De Inference Engine maakt geen gebruik van deze collections. Deze collections zijn dus dynamisch, zodat ze van de heap kunnen worden verwijderd, zodra ze overbodig geworden zijn.

Operatoren met twee argumenten in expressies gebruiken 25% meer geheugenruimte in Objective SIMPLEXYS. Thelses en de ThelseBys zijn in Objective SIMPLEXYS meer dan twee keer zo schadelijk.

Tabel 4: Geheugengebruik On-statements.

	SIMPLEXYS	Objective SIMPLEXYS	Verschil
Trigger	2	4	
FromList	2 + #Froms*2 + 2	10+ #Froms*4	
ToList	2 + #Tos*2 + 2	10+ #Tos*4	
Totaal	10 + 2 * (#Froms + #Tos)	24 + 4 * (#Froms + #Tos)	14 + 2 * (#Froms + #Tos)

Uit de tabel blijkt dat het opslaan van On-statements in Objective SIMPLEXYS iets meer dan twee maal zoveel geheugen kost als voorheen. Aangezien het aantal On-statements meestal veel kleiner is dan het aantal regels in een regelbestand, is deze forse stijging minder van belang.

Objective SIMPLEXYS gebruikt ook meer geheugenruimte voor Do-, Test en History code doordat de code van iedere Do, Test of History operator in een aparte procedure of functie is opgenomen. Het verschil wordt niet veroorzaakt, doordat de eigenlijke code van een operator anders gecompileerd wordt. Bij iedere procedure/functie wordt nl. entry en exit code toegevoegd. Afhankelijk van de procedure/functie declaratie worden in totaal ongeveer 12 bytes code toegevoegd als entry en exit code.

Tot dusver kan niet gesproken worden over schokkende verschillen tussen SIMPLEXYS en Objective SIMPLEXYS. Het nadeel dat Objective SIMPLEXYS meer vaardigheden van zijn programmeurs vereist, zal gecompenseerd moeten worden door voordelen die nog niet naar voren zijn gebracht. De interessante voordelen van de objectgeoriënteerde implementatie van SIMPLEXYS komen in de volgende paragraaf en in het volgende hoofdstuk pas naar voren.

5.4 Verruiming limieten

In Objective SIMPLEXYS zijn de beperkende limieten, die geconstateerd zijn voor de bestaande SIMPLEXYS implementatie (zie tabel 2 pag. 13), sterk verruimd.

De limieten die nu te onderscheiden zijn, worden voornamelijk bepaald door de totale geheugenruimte in de computer en het maximale aantal items dat een collection kan herbergen.

Ondervindt een uitgebreid SIMPLEXYS systeem hinder van een te kleine hoeveelheid werkgeheugen, dan kan een SIMPLEXYS programma in BP 7.0 als DPM programma gecompileerd worden. De totale grootte van het werkgeheugen kan dan gebruikt worden. Blijkt dit werkgeheugen niet toereikend te zijn voor een bepaalde applicatie dan kan dit eenvoudig worden uitgebreid.
Het maximale aantal items in een collections is 16.380 (65.520/4).
Dit maximum geldt voor:

- het aantal Fact-, Ask-, Test-, Eval-, Memo- en State-rules afzonderlijk;
- het totaal aantal regels (hoewel een geringe verandering van de implementatie deze beperking kan doen verdwijnen);
- het aantal Thelses van een regel;
- het aantal ThelseBys van een regel;
- het aantal regels in een From-list;
- het aantal regels in een To-list;
- het aantal On-statements.

Deze ruime limieten mogen alleen vergeleken worden met de limieten voor de Rule Compiler (RUC41) en de Inference Engine (SIM41) uit tabel 2. De verruiming van de limieten voor de Rule Compiler gelden wanneer de Rule Compiler op dezelfde objectbasis geïmplementeerd wordt als Objective Inference Engine uit hoofdstuk 4.

De meest beperkende limiet: de 180 regels van CHK41 is, zoals in §3.2 reeds is geconstateerd, niet op te lossen door objectgeoriënteerd te programmeren. Door regelbestanden modulair te maken zoals wordt voorgesteld in §3.1.1 behoeft de tool CHK41 niet veranderd te worden om toch te kunnen komen tot grotere SIMPLEXYS expertsystemen. In diezelfde paragraaf is reeds vermeld dat de stringente limieten, die PET41 oplegt door de datastructuren dynamisch te maken, eenvoudig te verruimen zijn.

6 Conclusies en Aanbevelingen

6.1 Conclusies

SIMPLEXYS is vooral geschikt voor diagnostische en regel-toepassingen en niet voor constructie, planning of ontwerp. Binnen de elektrotechniek zijn de toepassingen van veel expertsystemen juist constructie, planning of ontwerptaken. De meeste toepassingen voor SIMPLEXYS liggen wellicht in de (proces-)chemie, waar net als de huidige regelaars slechts één (of slechts enkele) grootheden geregeld hoeven te worden.

De voornaamste uitbreiding die in dit verslag aangedragen wordt, is het ondersteunen van kennismodulen. Kennismodulen maken grotere applicaties in SIMPLEXYS mogelijk.
Een objectieve implementatie van SIMPLEXYS is onderzocht. Deze implementatie doet mijns inziens niets af aan de oorspronkelijke kracht van SIMPLEXYS. De implementatie is ongeveer even snel, beslaat ongeveer dezelfde hoeveelheid code en herbergt alle constructies in zich om de bestaande SIMPLEXYS applicaties te verwerken.
Doordat de code in objective SIMPLEXYS modulair is (een eigenschap van objectgeoriënteerd programmeren), zijn kennismodulen eenvoudiger te realiseren.

6.2 Aanbevelingen

De aanbevelingen die in dit hoofdstuk gedaan worden, zijn een samenvatting van hetgeen tussen de regels door in de vorige hoofdstukken reeds te lezen was en tot nu toe nog niet gerealiseerd is in de SIMPLEXYS toolbox:

- modulaire kennissystemen (zie §3.1.1).
- één parser die regelbestanden kan compileren met daarin C of Pascal code (zie §3.3.2).
- één Inference Engine die externe code gebruikt uit DLLs (zie §3.3.2).
- constructies die selectief debuggen mogelijk maken (zie §3.3.6).
- het terugkoppelen van mogelijke optimalisaties die de Semantics Checker ontdekt in de objectstructuur (zie §3.3.6).
- het toevoegen van constructies om beter met tijd te kunnen redeneren (zie §3.3.9).

Het realiseren van deze punten vraagt veel tijd en een gedegen inzicht in de code van SIMPLEXYS. Ik ben van mening dat deze werkzaamheden het best verricht zouden kunnen worden door een AIO-opvolger.
Voor toepassingen van SIMPLEXYS op chemisch gebied zou contact gezocht kunnen worden met de faculteit Scheikunde.

Bijlage 1 Objectgeoriënteerd programmeren

De kreet "objectgeoriënteerd" is stilaan een trendy begrip aan het worden. Er wordt over gesproken alsof er een enorme revolutie binnen het programmeren gaande is. Daar de achtergronden van dit modeverschijnsel in een paar kantjes volledig uit de doeken kan worden gedaan, denk ik dat deze zaak een beetje overtrokken wordt[1].
Iemand die reeds goed gestructureerd te werk ging in zijn/haar procedurele taal (bijvoorbeeld Pascal, C), zal er misschien niet eens zo heel veel op vooruitgaan door objectgeoriënteerd te gaan programmeren, en tot de ontdekking komen dat hij/zij al enigszins zo programmeerde, als dat bij **OOP** *(Object Oriented Programming)* gebruikelijk is.

Objectgeoriënteerd programmeren heeft als voordeel dat een zeer gestructureerde en uniforme omgang met data- en objectstructuren gehanteerd kan worden (door o.a. inheritance en polymorfisme). Een hiërarchie van objectklassen kan zeer gemakkelijk worden aangemaakt. Eenmaal ontwikkelde objecten zijn eenvoudig opnieuw te gebruiken. Dit komt bij grote (of bij meerdere) projecten binnen een organisatie goed van pas. Indien veel verschillende objectklassen ontstaan, zal de nodige aandacht besteed moeten worden aan documentatie en onderhoud (zoals ook het geval is bij procedure bibliotheken).

Een **object** is het beste voor te stellen als een record. De velden van een object kunnen niet alleen variabelen zijn (zoals bij een conventioneel record), maar kunnen ook **method** (procedure/functie) declaraties zijn. Deze methods kunnen de variabelen van het object impliciet adresseren, dus zonder dat deze variabelen worden meegegeven als parameters aan de methods. Er wordt zelfs gesteld, dat de variabelen van een object alleen gemanipuleerd en gelezen mogen worden door de methods van het object (sommige objectgeoriënteerde programmeertalen staan niet anders toe).

---

1. [EZZ89] is een boek waarin objectgeoriënteerd programmeren in Turbo Pascal duidelijk wordt uitgelegd. Het boek is rijkelijk voorzien van voorbeelden. Een zeer storende blunder is dat in de eerste hoofdstukken herhaald gehamerd wordt op het niet gebruiken van globale variabelen inmethods van objecten. In een van de laatste hoofdstukken is een voorbeeld opgenomen waarin de methods van een object toch gebruik maken van globale variabelen.
Ook wordt het programmeren met de Self operator van Turbo Pascal niet besproken, maar voor het overige is dit boek de moeite van het lezen zeker waard voor iemand die objectgeoriënteerd wil leren programmeren en Pascal als achtergrond heeft.

De achterliggende gedachte hierbij is dat een object zich op een bepaalde manier gedraagt (een gedrag dat middels signalen aan de methods (aanroepen van methods) van het object wordt verlangd). Hoe dat dit gedrag tot stand komt is voor de buitenwereld absoluut niet van belang. Dus dient de buitenwereld volgens deze redeneerwijze ook geen toegang te hebben tot de variabelen binnen dat object die bijdragen tot het bepaalde gedrag.

Persoonlijk onderstreep ik dat variabelen van een object alleen door de methods van hetzelfde object veranderd mogen worden. Echter kan ik het direct lezen van de variabelen van een object best door de vingers zien:
Om de waarde van een veld van een simpele variabele (bijvoorbeeld integer of char) uit een record te gebruiken op een willekeurige plaats in het programma, schrijf je toch ook niet voor elke variabele van een recordtype een functie (waaraan je het record als parameter meegeeft) die de betreffende waarde retourneert. Dit doe je misschien wel als een record een complexe structuur in zich herbergt, waaruit bijvoorbeeld een combinatie van gegevens nodig is.
Ik denk dat het beste is om op deze manier ook met objecten om te springen, daar het zeker veel efficiënter is wanneer direct variabelen van een object geadresseerd worden, dan wanneer de waarde via een functie-call achterhaald moet worden.

Bovenstaande begrippen worden verduidelijkt door een wat uitgebreider (niet triviaal) voorbeeld (afkomstig uit de objectklasse bibliotheek van Turbo Pascal). Aan de hand van dit voorbeeld worden de meer geavanceerde mogelijkheden van objectgeoriënteerd programmeren uitgewerkt.

```
const
     MaxCollectionSize = 65520 div SizeOf(Pointer);

type
    PItemList = ^TItemList;
    TItemList = array[0..MaxCollectionSize - 1] of Pointer;

    TCollection = Object
                    Count: Integer;
                    Delta: Integer;
                    Limit: Integer;
                    Items: PItemList;

                    constructor Init(ALimit, ADelta: Integer);
                    destructor  Done; virtual;

                    function    At(Index: Integer): Pointer;
                    function    IndexOf(Item: Pointer): Integer; virtual;

                    function    FirstThat(Test: Pointer): Pointer;
                    function    LastThat(Test: Pointer): Pointer;

                    procedure   ForEach(Action: Pointer);
                    procedure   Insert(Item: Pointer); virtual;
                    procedure   FreeItem(Item: Pointer);
                    procedure   SetLimit(ALimit: Integer); virtual;
                  end;
```

Listing 10: Objectdeclaratie TCollection.

Een **objectklasse** wordt gedefinieerd als ieder ander door de programmeur te definiëren Pascal-type. TCollection is dus een objectklasse met vier variabelen en in dit geval tien methods (in werkelijkheid zijn er nog andere methods in TCollection gedefinieerd).

Na de type declaratie is het mogelijk een variabele van deze klasse aan te maken. Een collection kan een willekeurige hoeveelheid (tot een maximum MaxCollectionSize) items beheren. De hoeveelheid items hoeft van te voren niet vast te liggen. De items kunnen een verschillend type hebben; een item kan bijvoorbeeld een object zijn.
TCollection is hiermee een meer algemeen concept dan bijvoorbeeld een array, set of lijst. In een array kunnen immers alleen items van hetzelfde type worden opgeslagen en een array heeft een vaste grootte. In een set kunnen alleen enumerated types ondergebracht worden (in Turbo Pascal kan een set niet meer dan 256 elementen bevatten, een set of integer is in Turbo Pascal uitgesloten).
Een lijst komt nog het beste overeen met een collection. Een lijst kan dynamisch groeien en slinken, en middels sommige trucs kunnen ook items van verschillend type in één lijst worden opgenomen.

De variabelen van TCollection:

`Count` : Het aantal items dat in de collection aanwezig is.

`Limit` : Het maximaal aantal items dat op dit moment in de collection kan worden ondergebracht.

`Delta` : Het aantal items waarmee de collection wordt uitgebreid, indien een nieuw item aan de collection wordt toegevoegd wanneer de collection vol is (Count = Limit).

`Items` : Een pointer naar een array van pointers. Het array is in staat Limit pointers naar de items te herbergen. Indien de collection uitgebreid moet worden, wordt een nieuw array van pointers dynamisch aangemaakt (ter grootte Limit + Delta). De inhoud van het oude array wordt gekopiëerd naar het nieuwe array. Items wijst vervolgens naar dit nieuwe array en het oude array wordt van de heap verwijderd. De overhead van het vergroten van de collection is gering. De memory move wordt efficiënt uitgevoerd (door één machinetaal instructie). Deze operatie komt bovendien niet veelvuldig voor wanneer Delta verstandig gekozen wordt[1].

---

[1]. Turbo Pascal kent geen garbage collection (het verminderen van fragmentatie van de heap). Wanneer dynamisch geheugen geclaimd wordt, probeert Turbo Pascal zo klein mogelijke vrije gaten van de heap te benutten, zodat grote blokken op de heap vrij blijven. Garbage collection is om meerdere redenen ongewenst: garbage collection verhindert vaste responsietijden (dus real-time applicaties ondervinden hiervan vooral hinder), de programmeur moet werken met handles (een pointer naar een pointer). Dit is inefficiënt wat geheugen en runtijd betreft en bovendien moet de programmeur sterk rekening houden welke dynamische datastructuren op een bepaald moment verplaatst mogen worden. In de regel betekent dit, dat geheugenblokken die de programmeur op een bepaald moment wil adresseren in een algoritme, dit blok van te voren moet markeren als niet verplaatsbaar. Dit verhoogt de programma complexiteit en tevens benadeelt dit de snelheid van executie. Wanneer veel blokken gemarkeerd staan als niet verplaatsbaar, dan wordt het effect van garbage collection bovendien snel minder.

Na de variabelen vinden we een constructor en een destructor method declaratie. Dit zijn gewone procedure definities. Een constructor dient aangeroepen te worden bij objecten die virtuele methods (zie paragraaf "Statische en virtuele methods" op pag. 55) bevatten (in Turbo Pascal is als conventie aangenomen dat één constructor altijd Init heet). Een objectklasse heeft meestal één constructor, hoewel meerdere constructors toegestaan zijn. Aan TCollection.Init worden twee parameters meegegeven, waarmee Limit en Delta worden geïnitialiseerd. Voorts zet TCollection.Init Count op 0, en claimt Init dynamische geheugenruimte voor het array van pointers (Items^).
Een destructor (volgens Turbo Pascal conventie altijd Done geheten) wordt meestal gebruikt voor het opruimen en vernietigen van dynamische objecten. De destructor is in bijna alle gevallen virtueel.

Hieronder worden de methods van TCollection beschreven, ter illustratie van hetgeen een collection biedt. In latere paragrafen wordt hierop teruggegrepen, wanneer bijvoorbeeld inheritance en late binding aan bod komen.

`procedure Insert(Item: Pointer); virtual;`

Voegt een nieuw item achteraan de collection toe.

`procedure FreeItem(Item: Pointer);`

Verwijdert Item uit de collection.

`procedure SetLimit(ALimit: Integer); virtual;`

Zet Limit op maximum(ALimit, Count) en past de array-grootte aan. Deze method komt van pas wanneer er geen nieuwe items meer aan de collection worden toegevoegd, om het array een minimale grootte te geven. Daar de grootte van de collection niet vanzelf afneemt, kan deze method ook gebruikt worden, wanneer het aantal items in de collection sterk is geslonken.

`procedure ForEach(Action: Pointer);`

Roept voor ieder element uit de collection een procedure aan, die door de procedure-referentie Action aangewezen wordt. De aangeroepen procedure dient als FAR gedeclareerd te zijn en één argument te hebben; een pointer naar een item waarop Action een bewerking moet uitvoeren. De volgende programmaregels lichten dit toe:

```
var
   List : TCollection;

procedure PrintItem(Item: Pointer); far;

List.ForEach(@PrintItem);
```

`function At(Index: Integer): Pointer;`

Geeft een pointer terug naar het item dat gevonden wordt op Collection.Items^[Index].

`function IndexOf(Item: Pointer): Integer; virtual;`

Deze functie is de inverse functie van At: de index van Item binnen een collection wordt teruggevonden.

```
function FirstThat(Test: Pointer): Pointer;
```

Retourneert een pointer naar het eerste item in de collection waarvoor de boolean function Test evalueert tot true. De function Test dient als FAR gedeclareerd te zijn en één parameter te hebben; een pointer naar het item dat getest moet worden.

```
function LastThat(Test: Pointer): Pointer;
```

Zie FirstThat, alleen wordt nu een pointer geretourneerd naar het item met de hoogste index, waarvoor Test true oplevert.

In het programmafragment bij ForEach is te zien, dat de variabele List gedeclareerd wordt van het type TCollection. Voordat de collection gevuld kan worden (middels de method Insert), dient de constructor aangeroepen te worden:

```
List.Init(10,5);
```

Merk op dat de syntax van het aanroepen van een method behorend tot een object, is analoog aan het adresseren van een record veld. List wordt geïnitialiseerd als een collection die 10 items kan bevatten. Wanneer een nieuw item wordt toegevoegd en er Limit items aanwezig zijn in de collection, dan wordt de capaciteit van de collection met 5 vergroot.

Inheritance

Zoals reeds eerder is vermeld, is het mogelijk de objectklassen hiërarchisch te ordenen:

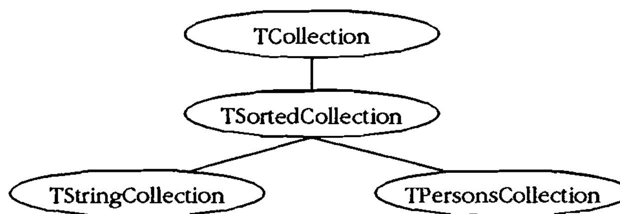

Fig. 7: Een hiërarchie van objectklassen.

De klasse TSortedCollection is afgeleid (**inherited**) van TCollection. Dit houdt in dat alle eigenschappen (zowel alle variabelen alsook alle methods) van TCollection worden geërfd door TSortedCollection. TCollection wordt een **superklasse** van TSortedCollection genoemd, en omgekeerd is TSortedCollection een **subklasse** van TCollection.

```
TSortedCollection = object(TCollection)
                      Duplicates: Boolean;

                      function Compare(Key1,Key2: Pointer): Integer; virtual;
                      function IndexOf(Item: Pointer): Integer; virtual;
                      function Search(Key:Pointer;var Index:Integer):Boolean;
                      virtual;

                      procedure Insert(Item: Pointer); virtual;
                    end;
```

Listing 11: Objectdeclaratie TSortedCollection.

TSortedCollection is een specialisatie van TCollection; er zijn dus specifieke eigenschappen toegevoegd aan TSortedCollection (bijvoorbeeld blijven de items van TSortedCollection gesorteerd op een bepaalde key wanneer items worden toegevoegd middels Insert).
In de eerste regel van de listing wordt TSortedCollection gedefinieerd als een objectklasse, die wordt afgeleid van TCollection. Een boolean variabele (Duplicates) en twee methods worden toegevoegd. Bovendien worden twee reeds in TCollection gedeclareerde methods (Insert en IndexOf) opnieuw gedeclareerd (overruled). Insert dient in dit geval de sorteervolgorde in stand te houden wanneer een item wordt toegevoegd aan de collection. Bij TCollection was dit zeker niet het geval, immers daar werd het nieuwe item na het laatste element in de collection toegevoegd.

Van TSortedCollection worden niet direct objecten afgeleid. TSortedCollection is een raamwerk voor verder gespecialiseerde subklassen. In figuur 7 zijn als voorbeeld twee mogelijke subklassen van TSortedCollection te zien:

- TStringCollection, een voorgedefinieerde objectklasse uit de Turbo Pascal bibliotheek, waarin Pascal-strings kunnen worden ondergebracht.
- TPersonsCollection, een collection waarvan de items bijvoorbeeld weer objecten met gegevens over één persoon zijn. De te programmeren method Compare kan een sortering op bijvoorbeeld achternaam in stand houden (maar ook op geboortedatum, postcode of een combinatie van keys).

De functies van de in TSortedCollection toegevoegde methods zijn:

```
Compare(Key1, Key2: Pointer): Integer;
```

De method Compare van TSortedCollection is een **abstract** method. Dit houdt in dat wanneer de Compare method van TSortedCollection aangeroepen wordt, er een run-time error optreedt. Zo wordt een programmeur gedwongen TSortedCollection.Compare in de subklassen van TSortedCollection te overrulen.
De virtuele method Compare die de programmeur voor een gesorteerde collection moet schrijven, vergelijkt twee items met elkaar. De functie Compare retourneert -1 als Key1 < Key2, 0 wanneer beide keys gelijk zijn en 1 indien Key1 > Key2.

```
function Search(Key: Pointer; var Index: Integer): Boolean;
```

De functie Search zoekt (binair) waar een bepaalde key in de collection thuishoort (gebruik makend van Compare). Indien een item met die key daadwerkelijk voorkomt in de collection, dan retourneert Search true. De variabele index duidt de positie van het item in de collection aan als het item in de collection is gevonden, en anders de positie waarop het item met de aangeboden key in de collection moet worden opgenomen, zodanig dat de sorteervolgorde in stand blijft wanneer het item daadwerkelijk toegevoegd wordt.

```
function IndexOf(Item: Pointer): Integer;
```

IndexOf vindt middels Search de array-index van Item.

```
procedure Insert(Item: Pointer);
```

Insert voegt het item toe, zodanig dat de sorteervolgorde in stand blijft. Insert is als volgt geïmplementeerd:

```
if not Search(KeyOf(Item), I) or Duplicates then AtInsert(I, Item);
```

P.S. de procedure AtInsert voegt Item toe op de I^e plaats in het Items^ array.

Statische en virtuele methods

Statisch zijn die methods, die maar één keer in de objecthiërarchie voorkomen. De method TCollection.At(Index: integer) : pointer; (de routine die een pointer naar een item retourneert die in het Items^ array op positie Index te vinden is), zal in geen enkele subklasse van TCollection op een andere wijze geïmplementeerd hoeven te worden. Een call naar een statische method wordt gerealiseerd als een call naar een absoluut adres dat tijdens het linken van het programma in de programmacode wordt opgenomen (=**early binding**).

Virtuele methods maken het mogelijk dat een method met dezelfde naam binnen de objecthiërarchie bij verschillende objectklassen verschillend geïmplementeerd kan worden (e.g. Insert). Dit wordt **polymorfisme** genoemd; Insert voegt zowel bij TCollection als ook bij TSortedCollection een Item aan de collection toe. Toevoegen is het polymorf gedrag van Insert.
De binding van een call naar een virtuele method aan een bepaald adres, vindt plaats tijdens de uitvoering van het programma (=**late binding**). Dat late binding noodzakelijk is blijkt uit de volgende situatie:

⇒ overridden method

Class:	TObject	TCollection		TSorted-Collection	TString-Collection
Variables:		Count Delta	Items Limit	Duplicates	
Methods:	⇒ Init ⇒ Free ⇒ Done	Init ⇒ Load Done At AtDelete AtFree AtInsert AtPut Delete DeleteAll Error FirstThat	ForEach Free FreeAll ⇒ FreeItem ⇒ GetItem ⇒ IndexOf ⇒ Insert LastThat Pack ⇒ PutItem SetLimit ⇒ Store	Load ⇒ Compare IndexOf Insert KeyOf Search Store	Compare FreeItem GetItem PutItem

Fig. 8: Inheritance-diagram van TStringCollection.

Zoeken we een bepaalde string in een TStringCollection, dan wordt TSortedCollection.Search gebruikt. Search gebruikt de method Compare. Stel dat Compare als een statische method gedeclareerd is, dan zal Search altijd TSortedCollection.Compare aanroepen, maar TStringCollection.Search dient in dit geval gebruikt te worden.

Voor iedere objectklasse met één of meerdere virtuele methods wordt een VMT (**Virtual Method Table**) aangemaakt, waarin de adressen van de virtuele methods van die klasse worden geplaatst. Een instantie (object) van een dergelijke objectklasse krijgt bij het aanroepen van een constructor van zijn klasse een binding met de juiste VMT. Deze late binding vindt dus slechts eenmalig plaats. De overhead van een aanroep van een virtuele method bestaat dus slechts uit een indirecte adressering.
Daar Compare uit het voorbeeld hierboven een virtuele method is, zal TSortedCollection.Search Compare aanroepen via de referentie in de VMT van het TStringCollection object. Zo wordt dus het adres van TStringCollection.Compare gevonden.

Dynamische objecten in BP

Tijdens het runnen van een programma kan het aantal objecten van een bepaald type toe of afnemen (denk bijvoorbeeld aan het aantal personen in een database met persoonsgegevens). Zoals Pascal pointers naar dynamische variabelen kent, kent Pascal ook pointers naar dynamische objecten.
Voorbeeld:

```
type
    PPersonObject = ^PersonObject;
    PersonObject  = object
                      Name   : PString;
                      GebDat : longint;
                      . . . .
                      constructor Init;
                      destructor Done;

                      function GetAge : byte; virtual;
                      procedure Print; virtual;
                      . . . .
                    end;

var
   P : PPersonObject;
```

Listing 12: Typedeclaratie dynamische objecten.

Voordat we P^ kunnen gebruiken moeten eerst twee initialisaties worden uitgevoerd:
- Voor P^ moet geheugenruimte op de heap worden verkregen (New(P)).
- P^.Init moet worden aangeroepen, omdat het object virtuele methods bezit.

Daar beide handelingen veelvuldig terugkeren in Turbo Pascal is het New-statement uitgebreid: New(P,Init); nadat de geheugenruimte voor P^ is verkregen wordt automatisch P^.Init aangeroepen.

Ook het dispose-statement is op dezelfde wijze uitgebreid: dispose(P, Done). Dispose roept nu eerst de destructor P^.Done aan, voordat de ruimte die P^ op de heap inneemt, wordt vrijgegeven.

Programmeren van methods

Getoond is, dat objecten d.m.v. type definities worden gedeclareerd. De methode headers behorend tot een objectklasse worden vastgelegd bij declaratie van de objectklasse. De implementatie van methods volgt na de objecttype declaraties. Dit is vergelijkbaar met de scheiding tussen de interface en de implementation secties van Turbo Pascal units. In de interface sectie vindt alleen de declaratie van de procedures plaats, die buiten de unit gebruikt kunnen worden. De daadwerkelijke implementatie van dergelijke procedures vindt plaats in de implementation sectie.

Voorbeeld van de implementatie van PersonObject.GetAge:

```
function PersonObject.GetAge;
begin
  GetAge:=Year(ThisDate-GebDat);      { pre: Year and ThisDate existing functions }
end;
```

PersonObject.GebDat wordt hier impliciet gebruikt, zonder dat GebDat aan GetAge is meegegeven als parameter. GebDat wordt namelijk achterhaald via Self. Self is een parameter die impliciet wordt meegegeven aan een method die aangeroepen wordt. Self is een record met daarin de variabelen van het object (en de referentie naar de VMT behorend bij het object). @Self is binnen een method te gebruiken als een pointer naar het object dat door de method wordt bewerkt.
Zowel Self als @Self zijn zelden nodig tijdens het programmeren. Hooguit kan bijvoorbeeld een method die een PersonObject vult, zichzelf bijvoorbeeld toevoegen aan een TPersonsCollection (door TPersonsCollection.Insert(@Self) aan te roepen, nadat de key-variabelen van het PersonObject zijn gevuld).

Het ontwerpen van een objectstructuur

Nu de theorie van objectgeoriënteerd programmeren behandeld is, is een notitie omtrent het gebruik van objecten op zijn plaats.
Bij het opzetten van de objecthiërarchie dient de programmeur het eigenlijk denkwerk reeds gedaan te hebben; een goede objecthiërarchie komt namelijk niet vanzelf tot stand. Worden bij het vastleggen van de objectklassenhiërarchie verkeerde keuzen gemaakt, dan is correctie na implementatie arbeidsintensief. Ook kan het voorkomen dat een probleem zonder objectgeoriënteerd programmeren goed te implementeren is.
De keuzen die de programmeur uiteindelijk maakt zullen dus op gestructureerd probleemoplossen gestoeld moeten zijn. Ervaring in het kiezen van een objectstructuur zal het maken van de keuzen vergemakkelijken.

Voor en nadelen van objectgeoriënteerd programmeren

De meeste voordelen van objectgeoriënteerd programmeren zijn in de bovenstaande uitleg over objectgeoriënteerd programmeren reeds belicht.
Kort samengevat zijn dit:
- hiërarchiestructuur
- polymorfisme, uniformiteit
- herbruikbaarheid van objectklassen
- noodzaak tot gestructureerd probleem oplossen

Object georiënteerd programmeren beïnvloedt de executiesnelheid van een gecompileerd programma zowel in positieve als ook in negatieve zin.
De executiesnelheid wordt positief beïnvloed wanneer veel case-statements, die noodzakelijk zijn bij niet objectgeoriënteerd programmeren, vermeden kunnen worden:

```
case TypeOperator of
  And : DoAndProc;
  OR  : DoOrProc;
  . . . . .
end;
```

Wanneer Operator als een object geïmplementeerd wordt, kan voor de verschillende type operatoren een virtuele DoProc geprogrammeerd worden:

```
type
    OOperator     = object
                      procedure DoProc; virtueel;
                    end;
    OAndOperator  = object(OOperator)
                      procedure DoProc; virtueel;
                    end;
    OOrOperator   = object(OOperator)
                      procedure DoProc; virtueel;
                    end;
    . . . . .
```

I.p.v. bovenstaande case is het aanroepen van de DoProc voor een operator object reeds voldoende:

```
Operator.DoProc;
```

De executiesnelheid van een objectgeoriënteerd programma wordt bovendien positief beïnvloed doordat een method meestal minder parameters kent dan een procedure bij een niet objectgeoriënteerd programma. Een method heeft namelijk impliciet de mogelijkheid de variabelen behorend tot een object te adresseren.

Het nadeel dat objectgeoriënteerd programmeren met zich meebrengt voor wat betreft de executiesnelheid is, dat het adres van een virtuele method moet wordt via een indirecte adressering in de VMT wordt bepaald, wanneer een virtuele method wordt aangeroepen. Het adres van de procedure is dus niet absoluut in de code opgenomen.

Over het effect van objectgeoriënteerd programmeren op de executiesnelheid van een programma valt niet meer te zeggen dan dat dit sterk afhankelijk is van het programma[1].

De grootste prijs die betaald dient te worden, wanneer er objectgeoriënteerd geprogrammeerd wordt, is de toename van de omvang van de gecompileerde code[2]:

- wanneer bepaalde objecten uit een objectbibliotheek gebruikt worden, dan worden vaak veel methods meegelinkt die misschien wel niet gebruikt worden;
- er is meer code nodig voor de indirecte calls via de VMT.

1. Wanneer SIMPLEXYS objectgeoriënteerd geprogrammeerd wordt, zullen zeker veel case-statements vervallen, wat tijdwinst op zal leveren. Daar de meeste procedures in SYMPLEXYS weinig parameters hebben, zal deze factor de executiesnelheid niet merkbaar beïnvloeden. SYMPLEXYS zal traagheid ondervinden door toepassing van virtuele methods. Wanneer een groot SYMPLEXYS programma veel TEST, DO en HISTORY operators heeft, zal een objectgeoriënteerde implementatie van SIMPLEXYS een merkbare snelheidswinst opleveren.
2. De grootte van de executable code van een programma speelt tegenwoordig een ondergeschikte rol: geheugen is immers erg goedkoop, moderne compilers (DPM compilers, zie bijlage Borland Pascal 7.0) voor PCs (ATs) hebben geen last meer van het 640K syndroom. Hierdoor is het niet meer noodzakelijk elke byte werkgeheugen uit te knijpen, maar kunnen afwegingen als executiesnelheid vs. executable-grootte en ontwikkeltijd vs. executable-grootte eenvoudigweg uitvallen ten nadele van de executable-grootte.

Bijlage 2 DLLs

Dynamic Linked Libraries zijn bibliotheken van gecompileerde routines, waarvan de routines door andere programma's gebruikt kunnen worden. DLLs zijn enigzins te vergelijken met Turbo Pascal units. Het belangrijkste verschil is echter dat de informatie uit een unit die een programma gebruikt, statisch wordt gelinked in de exe-file die van dat programma wordt gemaakt. Procedures en functies van DLLs kunnen tijdens runtijd van een programma dynamisch geladen worden uit de reeds eerder gecompileerde DLL-file. Een DLL kan ook uit het geheugen verwijderd worden, wanneer deze zijn diensten heeft bewezen. Een ander verschil met units is dat slechts procedures en functies door een DLL geëxporteerd kunnen worden. Een DLL kan net als units beschikken over lokale variabelen, die niet publiek gemaakt kunnen worden.
DLLs kunnen alleen gebruikt worden in DPM- of Windows-programma's. De procedures en functies van een DLL kunnen door meerdere programma's gebruikt worden. De programmeertaal waarmee een DLL gemaakt is, is niet van belang. Een Turbo Pascal programma kan gebruik maken van routines in een DLL-file, die gemaakt is met bijvoorbeeld Visual Basic. Real mode programma's kunnen slechts gebruik maken van overlays voor het dynamisch laden van executable code. Overlays zijn taal-afhankelijk en daarmee niet algemeen toepasbaar zoals DLLs dat zijn. Overlays worden in DPM en Windows programma's, die met Turbo Pascal aangemaakt worden, niet meer ondersteund; in deze modes moet dus gebruik gemaakt worden van DLLs wanneer dynamische code noodzakelijk is.

Gebruik van DLLs in vooral Windows (en Borland Pascal) is oppassen geblazen. Om fouten te voorkomen dienen procedures en functies niet met naam gerefereerd te worden. Een voorbeeld is het meest illustratief voor het aanmaken en gebruiken van DLLs.

```
library MyLib;

uses
    Crt;

function DllFunc(a,b: word) : longint; export;
begin
  DllFunc:=a*b;
end;

procedure DllProc; export;
begin
  Sound(1000);
  delay(300);
  NoSound;
end;

exports
       DllFunc index 1 name 'Mul' resident,
       DllProc index 2 name 'Beep' resident;

begin
end.
```

Listing 13: Voorbeeld DLL.

Een DLL-file kenmerkt zich door het keyword 'library' i.p.v. unit. Procedures en functies die buiten de DLL gebruikt gaan worden, moeten 'export' achter hun declaratie hebben en moeten opgenomen worden in de 'exports' sectie. De procedures en functies worden in deze sectie nogmaals genoemd, worden voorzien van een unieke index (een positief getal van 1..32.768) en krijgen eventueel een naam waaronder de functie bekend staat in de programma's die de DLL gebruiken. Het keyword 'resident' duidt aan, dat deze referentienamen in het werkgeheugen komen, wanneer de DLL geladen wordt. Dit heeft een positieve invloed op de snelheid indien procedures en functies met naam gerefereerd worden.
Ondanks dat het eenvoudiger voor de programmeur is om de procedures en functies met naam te refereren, dient dit vermeden te worden. Afgezien van traagheid werkt dit niet geheel vlekkeloos (bijvoorbeeld wanneer DLL-files aangemaakt zijn met Visual Basic). In deze bijlage wordt het voorbeeld dan ook slechts met index-referenties voortgezet.

```
library MyLib;

uses
    Crt;

function Mul(a,b: word) : longint; export;
begin
  Mul:=a*b;
end;

procedure Beep; export;
begin
  Sound(1000);
  delay(300);
  NoSound;
end;

exports
       Mul  index 1,
       Beep index 2;

begin
end.
```

Listing 14: DLL met slechts index referenties.

Een DLL kan overigens net als een unit initialisatiecode hebben tussen begin en end..

Gebruik van DLLs

Er zijn twee manieren om DLLs te gebruiken. De eerste methode is het eenvoudigst. Turbo Pascal gebruikt dan expliciet de DLL. De DLL wordt automatisch geladen wanneer het programma start. De geheugenruimte die door de DLL gebruikt wordt, wordt pas weer vrijgegeven, wanneer het programma eindigt.
Een voorbeeld van het expliciet gebruik van een DLL is te vinden in de volgende listing:

```
program Explicit;

uses
    Crt;

function Mul(a, b: word) : longint; far; external 'MyLib' index 1;
procedure Beep; far; external 'MyLib' index 2;

begin
  ClrScr;
  writeln(Mul(4,2));
  Beep;
end.
```

Listing 15: Expliciet gebruik DLL.

De procedure en functie headers uit de DLL worden opnieuw gedeclareerd. Deze declaratie is niet onderhevig aan de type-checking. Het is verantwoordelijkheid van de programmeur, dat de header overeenstemt met de declaratie in de DLL-file. De procedures/functies worden als far gedeclareerd. Het keyword 'external' duidt aan dat de procedure/functie geladen moet worden uit een externe file (in dit geval uit MYLIB.DLL). De procedures en functies worden gerefereerd middels hun index (de procedure en functienamen kunnen zodoende afwijken van de namen die de procedures en functies hebben in de DLL-file.

Door functies en procedures te declareren als 'external', worden deze geladen zodra het programma start. De programmeur hoeft dus niet expliciet statements in het programma op te nemen die de DLL-file laden en beschikbaar maken in het werkgeheugen.

De tweede manier om DLLs te gebruiken is door DLLs zelf te laden in (of te verwijderen uit) het werkgeheugen. Om een DLL-file te laden dient de 'LoadLibrary' routine van Windows gebruikt te worden. Een DLL wordt uit het geheugen verwijderd door 'FreeLibrary'. Beide procedures zijn opgenomen in de WinProcs unit van Turbo Pascal voor Windows. Wordt echter geen Windows maar een DPM programma gemaakt, dan moet de WINAPI unit gebruikt worden.
Met de routine 'GetProcAddress' zijn referenties naar procedures/functies uit de DLL te bewerkstelligen. Procedures en functies zijn middels hun naam via 'GetProcAddress' in de DLL te achterhalen. Veiliger en sneller is het gebruik van adressering over de index-nummers. I.p.v. de procedure/functie naam moet als tweede parameter de index worden meegegeven (en wel als longint).

```
program Dynamic;

uses
    WinAPI, Crt;

type
    MulT  = function (a, b: word) : longint;
    BeepT = procedure;

var
   Mul   : MulT;
   MulF  : TFarProc absolute Mul; {To avoid TypeCasting procedure/function param.}
   Beep  : BeepT;
   BeepF : TFarProc absolute Beep;

var
   hDLL : THandle;

procedure Error;
begin
  writeln('GetProcAddress returned nil');
  Halt(0);
end;

begin
  ClrScr;
  writeln(MemAvail);
  writeln;
  hDLL:=LoadLibrary('MYLIB.DLL');
  MulF:=GetProcAddress(hDLL,PChar(longint(1)));
  if MulF=nil then Error;
  BeepF:=GetProcAddress(hDLL,PChar(longint(2)));
  if BeepF=nil then Error;
  writeln(MemAvail);
  Beep;
  writeln(Mul(4,2));
  writeln;
  FreeLibrary(hDLL);
  Writeln(MemAvail);
  ReadKey;
end.
```

Listing 16: Dynamisch gebruik DLLs.

In Turbo Pascal kunnen procedure/functie variabelen minder goed getype-cast worden dan andere parameters. Daar 'GetProcAddress' is gedefinieerd als een functie van het type TFarProc, moet voor zowel Mul als Beep type-casting toegepast worden. Deze type-casting wordt uitgevoerd door resp. MulF en BeepF, die dezelfde geheugenadressen beslaan als Mul resp. Beep.

Gebruik DLLs in SIMPLEXYS

De verschillende SIMPLEXYS tools zouden als DLLs geïmplementeerd kunnen worden. Do-, Test-, History-code en de Init en Exit procedures kunnen dan dynamisch gelinked worden aan het gecompileerde regelbestand in een ontwikkelomgeving van SIMPLEXYS tools. Wanneer de ontwikkelomgeving Pascal/C-code moet compileren kan dit als subproces. De nieuw aangemaakte DLL kan zodoende geladen worden, wanneer het expertsysteem uitgevoerd wordt. Een voordeel is ook, dat het niet uitmaakt of externe code geschreven is in Pascal of C (of desnoods een andere programmeertaal). De ontwikkelomgeving dient alleen de juiste compiler aan te roepen die de externe code compileert tot een DLL file.

Dit alles veronderstelt wel, dat SIMPLEXYS als DPM programma gecompileerd wordt. Het nadeel hiervan is, dat de redeneersnelheid in DPM ongeveer 50% lager ligt dan in real-mode[1].

---

1. Zie bijlage 7.

Bijlage 3 Borland Pascal 7.0

Borland Pascal is een ontwikkelomgeving met naast de Turbo Pascal compiler: debuggers, assemblers en een profiler. De meest in het oog springende wijzigingen t.o.v. Turbo Pascal 6.0 zijn:

Break	Stapt buiten scope van for/while/repeat loop (als C).
Continue	Springt naar het beginpunt van een lus (for/while/repeat) (als C).
Open array parameters	Een open array dat meegegeven wordt aan een procedure of functie (als: var X: array of real), begint altijd met index 0 (ook al was dat niet het geval bij het array dat meegeven is). De hoge bound van de arrayindex is m.b.v. de functie High te achterhalen. Open arrays kunnen maar in één dimensie open zijn. Het array dat aan een procedure wordt meegegeven als open array, is een bestaand van te voren gedeclareerd Pascal array. Een dergelijk array kan dus slechts maximaal 64 Kb groot zijn (de maximale grootte van een segment in DOS).
C-strings	Naast de bestaande Pascal strings (die voorafgegaan worden door een lengte byte), zijn er nu ook 0-terminated strings (C-strings). In de 'Strings' unit zijn procedures te vinden die op dit type strings operaties uitvoeren. Het voorgedefinieerde type PChar is een pointer type naar een dergelijke C-string.
Include(Set, Item);	Efficiënte manier om één element aan een set toe te voegen.
Exclude(Set, Item);	Efficiënte manier om één element uit een set te verwijderen.
VER70	Dit symbool is gedefinieerd bij iedere (toekomstige) compiler die Borland Pascal versie 7.0 source accepteert. Voorbeeld van een toepassing:

```
{$IFDEF VER70}
    Break;
{$ENDIF}
```

Borland Pascal is staat programma's te genereren voor verschillende targets:

- Real mode programma's (gewone DOS programma's);
- DPM "*Dos Protected mode*" programma's die ook het extended geheugen van een computer gebruiken. Voorwaarde is dat een memory manager geladen is die het extended geheugen beheert (bijvoorbeeld HIMEM.SYS). BP is een DPM programma;
- Windows programma's.

Alleen in real mode kunnen programma's overlay-files gebruiken. Dynamic Linked Libraries (DLLs) vervangen de overlay-techniek wanneer een programma als DPM of Windows programma ontwikkeld wordt. In bijlage 2 is beschreven hoe DLLs gebruikt kunnen worden.

De volgende voorgedefiniëerde constanten staan in relatie tot het maken van DPM programma's:

Seg0040 Segment descriptor om de BIOS variabelen te adresseren. Wanneer niet via deze segment descriptor een locatie in dit gedeelte van het geheugen geadresseerd wordt, treedt een 'memory violation error' op.

SegA000 Segment descriptor voor segment $A000.

SegB000 Segment descriptor voor het monochroom video geheugen.

SegB800 Segment descriptor voor het kleuren video geheugen.

Wanneer een DPM programma dat deze descriptors gebruikt weer als real mode programma gecompileerd wordt, worden de segment adressen gesubstitueerd; de programmacode behoeft niet gewijzigd te worden.

De ontwikkelomgeving is niet in staat integrated debugging toe te passen op programma's die als DPM programma gecompileerd worden. Er kan echter een snelle verbinding gelegd worden vanuit de ontwikkelomgeving naar een externe debugger. De stand-alone debugging optie in de debugging opties dialog box moet geactiveerd zijn. Indien een DPM programma gedebugged moet worden, dan dient dit te geschiedden door een externe debugger die DPM programma's kan debuggen (TDX.EXE).
Een real mode programma kan als vanouds door de ontwikkelomgeving gedebugged worden, maar indien wenselijk kan van een externe debugger (e.g. TD.EXE, DEBUG.EXE) gebruik gemaakt worden.

SIMPLEXYS kan zonder wijzigingen door Borland Pascal 7.0 gecompileerd worden. De Semantics Checker (CHK41) draait na compilatie in Borland Pascal 7.0 40% sneller. Dit komt doordat de code voor het omgaan met sets efficiënter is geworden. De Semantics Checker werkt nagenoeg alleen maar met sets, waardoor de snelheidswinst zo groot is.

Foutjes in Borland Pascal 7.0

Hoewel Borland gecomplimenteerd kan worden voor het geringe aantal bugs die de Turbo Pascal compilers sinds versie 3.0 hadden, gaat dit compliment helaas niet op voor Borland Pascal 7.0.
Ondanks het kleine aantal uitbreidingen van de programmeertaal, hebben blijkbaar de uitbreidingen als DPM en uitgebreide tools die bij Borland Pascal 7.0 geleverd worden ertoe geleidt dat minder aandacht is besteed aan de Pascal compiler.

De fouten die ik aantrof zijn:
- De compiler reageert niet of slecht op compiler directives (als: $R+,$S+,D+,L+,Y+) die in de source code zijn opgenomen. De enige manier om zeker te zijn hoe source code gecompileerd wordt, is door alle compiler directives uit de source code te verwijderen, en middels het 'options' menu de gewenste compiler opties in te stellen.
- De run-tijden zijn afhankelijk van source die in bodies van procedures die **niet** uitgevoerd worden is opgenomen.
- De method TCollection.Search is niet correct geïmplementeerd.
- Wanneer de 'EXE & TPU' directory niet op de huidige directory ingesteld staat, treden soms fouten op wanneer getracht wordt een DPM-programma te starten vanuit de ontwikkelomgeving. Er treedt een internal error 10 (een load error) op in de DPM-loader, die een DMP in het werkgeheugen laadt. De DPM-loader is een losstaande utility van Borland, daar de standaard loaders van DOS slechts real-mode programma's kunnen laden in werkgeheugen[1].
- De on-line help is niet volledig wat bijvoorbeeld de Turbo Pascal Object Class Library betreft.
- De on-line help heeft een behoorlijk aantal foutieve interne referenties: indien een onderwerpreferentie gekozen wordt, verschijnt niet altijd de juiste informatie.

Vooral de eerste drie items zijn echt storend. Ik weet niet of Borland inmiddels een nieuwere versie van Borland Pascal heeft uitgebracht, waarin de fouten zijn verbeterd.
Wordt om deze fouten heen gewerkt, dan ben ik tot dusver geen fouten in executables die aangemaakt zijn met Borland Pascal 7.0 tegengekomen[2].

---

1. Sommige DPM-programma's kunnen ook zonder een dergelijke load-utility. De functionaliteit van een dergelijke loader is dan opgenomen in de executable van een DPM-programma. Borland heeft gekozen de DPM-loader op te nemen als een losstaand programma (RTM.EXE), zodat de DPM-exe-files (±10K) kleiner blijven.
2. Ditzelfde geldt niet alleen voor de executables van SIMPLEXYS maar ook voor de 25.000 regels van eigen programma's die ik in Turbo Pascal geschreven heb.

Bijlage 4 Inheritance diagrammen

In deze bijlage zijn hiërarchieën van objectklassen die in Objective SIMPLEXYS gebruikt worden in diagrammen weergegeven. Een objectklasse wordt gekenmerkt door haar:

- naam
- variabelen
- methods

Een virtuele method wordt *cursief* in de diagrammen weergegeven. Virtuele methods worden soms in subklassen overruled (waar de desbetreffende method ook virtueel dient te zijn), om zodoende een polymorf gedrag te kunnen bewerkstelligen (zie bijlage). Subklassen van een bepaalde klasse staan rechts van die klasse in het diagram, links staat de superklasse. Een voorbeeld ter illustratie:

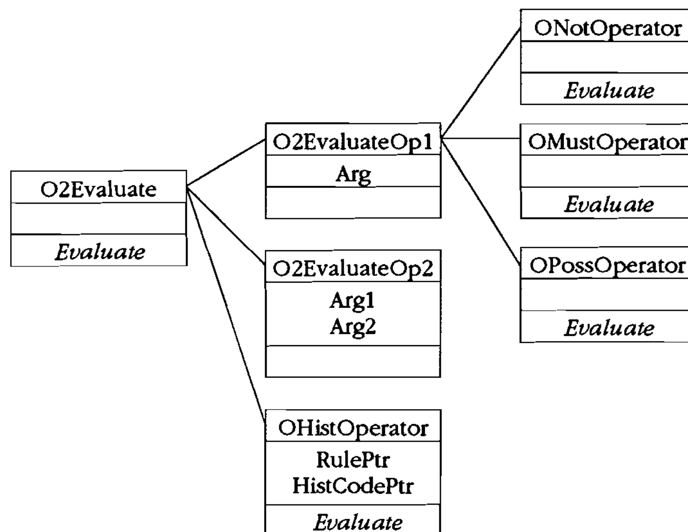

Fig. 9: Voorbeeld inheritance diagram.

De objectklasse O2Evaluate heeft geen variabelen maar wel een virtuele method "*Evaluate*". O2Evaluate is hier de superklasse van drie objectklassen: O2EvaluateOp1, O2EvaluateOp2 en OHistOperator.

O2EvaluateOp1 heeft één variabele "Arg" en is de subklasse van O2Evaluate. Deze objectklasse definiëert geen nieuwe methods (maar erft als subklasse wel *Evaluate* van O2Evaluate). O2EvaluateOp1 heeft drie subklassen die elk de method *Evaluate* van de superklasse O2Evaluate overrulen.

Wanneer een objectklasse in de onderstaande inheritance diagrammen begint met een hoofdletter T, dan wordt verwezen naar een objectklasse uit de Turbo Pascal Object Class Library. Deze objectklassen worden hier niet nader uitgewerkt maar zijn in de documentatie en de on-line help van Turbo Pascal worden deze objectklassen uitvoerig beschreven.
De objectklassen, die voor Objective SIMPLEXYS gedefinieerd zijn, beginnen met een hoofdletter O. De typen die de variabelen en methods van een objectklasse hebben, zijn niet opgenomen in de diagrammen. Een geïnteresseerde lezer wordt verwezen naar de source code, die naast de diagrammen in deze bijlage een volledig inzicht verschaft.

Allereerst worden zes objectklassen in een diagram gepresenteerd, die geen subklassen hebben. De meer complexe objectklassestructuren zijn in diagrammen weergegeven.

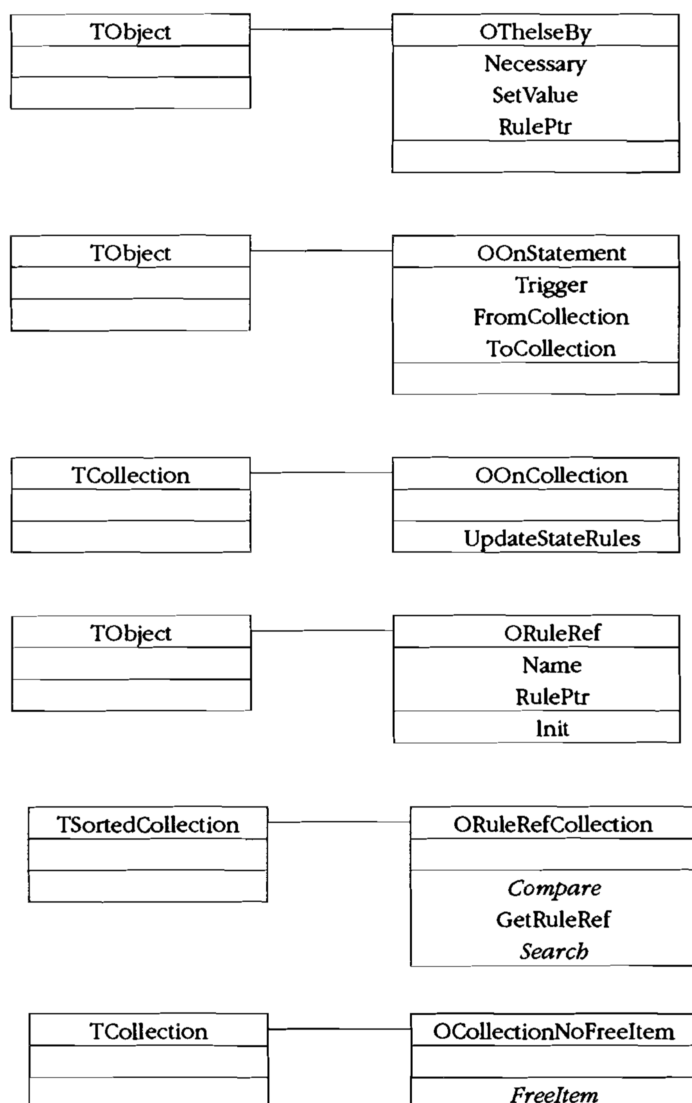

Fig. 10: Enkelvoudige inheritance diagrammen.

OCollectionNoFreeItem moet soms gebruikt worden i.p.v. TCollection. Bij TCollection wordt namelijk verondersteld, dat de items in de collection alle afstammen van TObject. Wordt een TCollection opgeruimd door de destructor TCollection.Done, dan roept de TCollection.Done voor ieder item uit de collection TCollection.FreeItem aan. TCollection.FreeItem ruimt een item op door de Done destructor van het item dat aan FreeItem is meegegeven als parameter aan te roepen.
Is een item echter opgenomen in meer dan één collection, dan dreigt het gevaar, dat meer dan eens getracht wordt de Done destructor van een item aan te roepen. OCollectionNoFreeItem is een subklasse van TCollection die alleen FreeItem dusdanig overrulet, dat van het item niet de Done destructor aangeroepen wordt.
Wanneer het voorkomt, dat items worden ondergebracht in twee of meer collections, dan moet dus zorgvuldig bepaald worden welke collection de destructor van de items aanroept. De collection die de items vernietigt, moet bestaan zolang andere collections met referenties naar dezelfde items nog gebruikt worden.
Voorbeelden van het gebruik van OCollectionNoFreeItem zijn FromCollection en ToCollection uit de objectklasse OOnStatement. De items van deze collections verwijzen nl. direct naar regels, waarvan de destructor aangeroepen wordt door de betreffende regelcollection.
In figuur 10 is te zien, dat de method TSortedCollection.Search overruled wordt door de objectklasse ORuleRefCollection. Search hoeft eigenlijk nooit in een subklasse van TSortedCollection overruled te worden omdat Search binair zoekt in de gesorteerde collection. Het vergelijken van twee items uit de collection besteedt Search uit aan de method Compare (die in iedere subklasse van TSortedCollection overruled moet worden). Helaas bleek dat TSortedCollection.Search door Borland niet foutloos geïmplementeerd is, zodat de method eigenhandig gemaakt is.

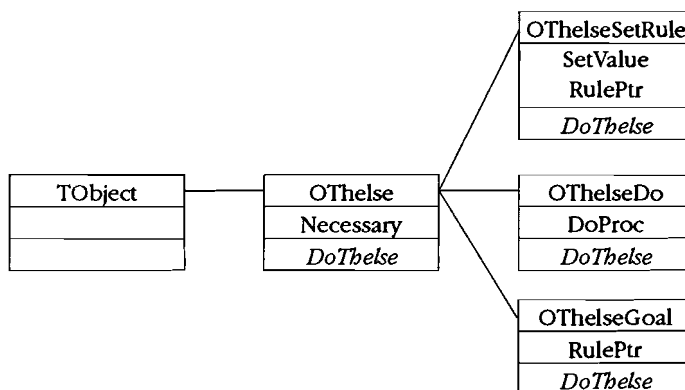

Fig. 11: Inheritance diagram OThelse.

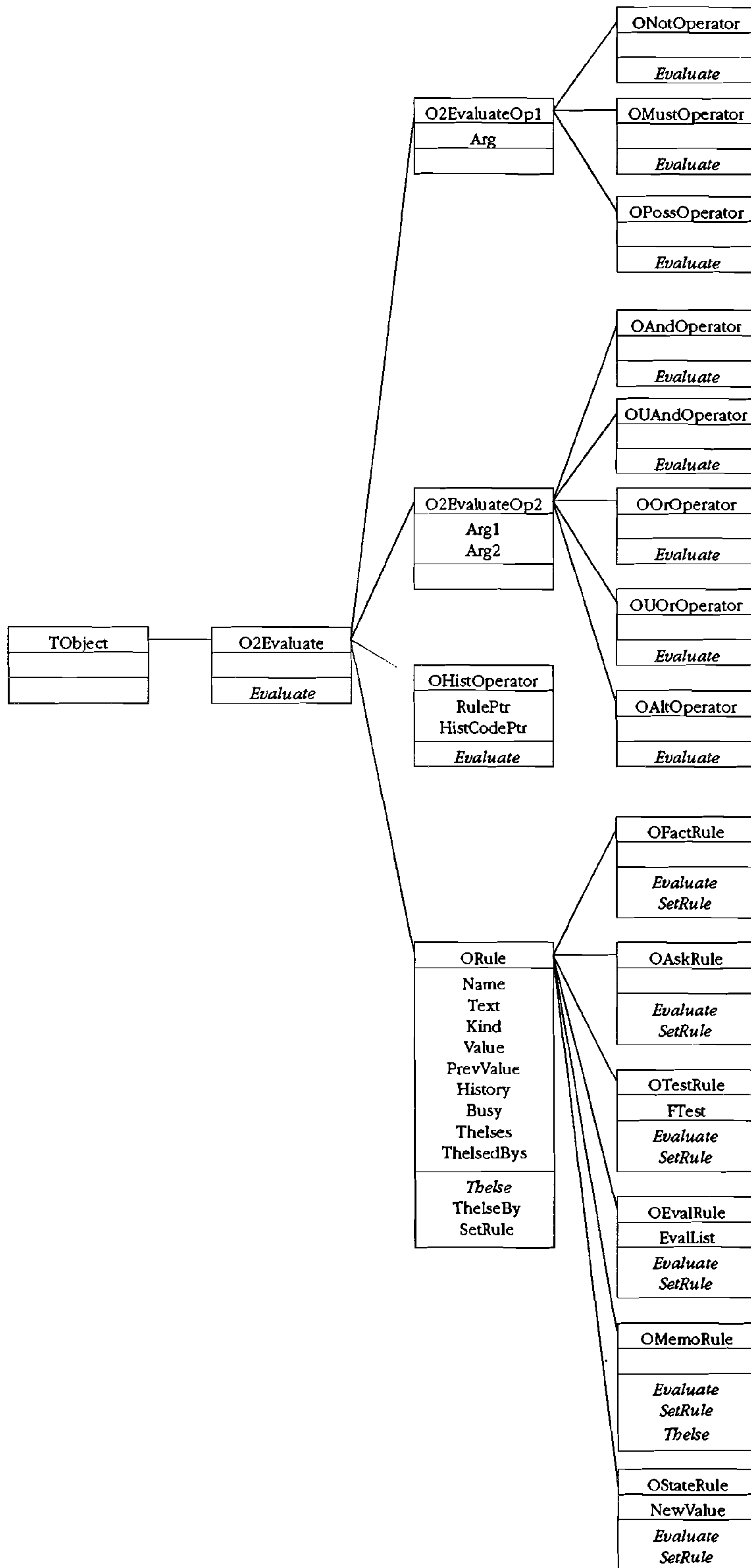

Fig. 12: Inheritance diagram O2Evaluate.

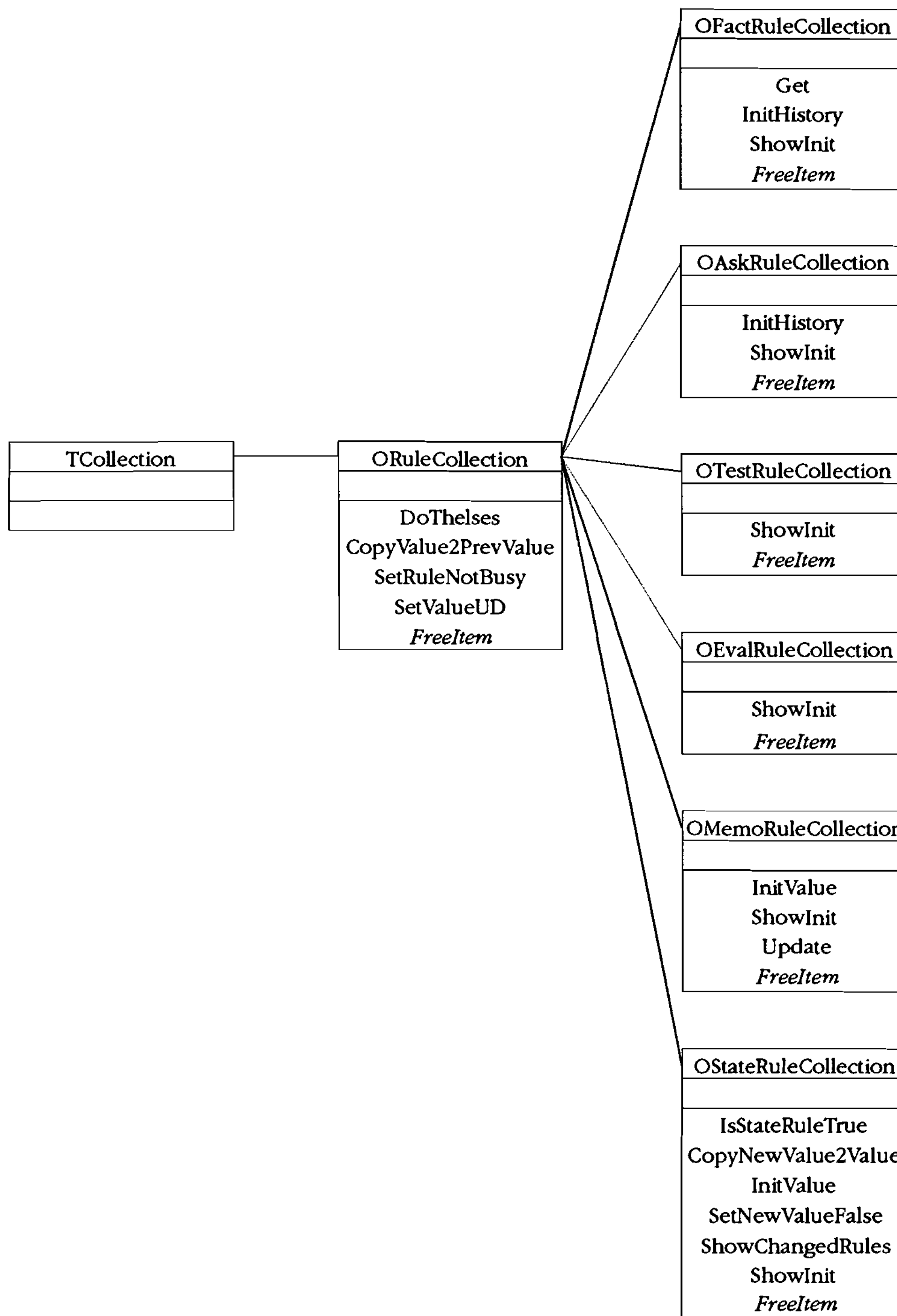

Fig. 13: Inheritance diagram ORuleCollection.

De method TCollection.FreeItem in figuur 13 wordt overruled door ORuleCollection **en** wederom in de subklassen van ORuleCollection. ORuleCollection.FreeItem roept **niet** de destructors van de items aan. Dit wordt overgelaten aan de subklassen van ORuleCollection. Voor deze ogenschijnlijke overbodig complexe constructie is gekozen zodat geen fouten kunnen ontstaan indien de algemene collection van regels overbodig wordt geacht. Nu bestaat wel zo'n algemene collection, die alle regels in het expertsysteem beheert, om ineens

van alle regels in het systeem de waarde van bijvoorbeeld "Value" UD te kunnen maken. Een reden om af te zien van een algemene collection, die alle regels in het expertsysteem beheert, is dat deze collection het totaal aantal regels in het systeem beperkt (tot het aantal regels als dat er items in een collection kunnen: 16380).
De zes collections voor ieder regeltype kunnen ieder 16380 regels herbergen.

Bijlage 5 CHK41

Deze bijlage geeft aan, welke veranderingen in CHK41 aangebracht moeten worden, zodat de Semantics Checking geïntegreerd kan worden in de Rule Compiler. Het voordeel hiervan is dat de gecompileerde regelbeschrijving niet naar disk hoeft te worden geschreven om met de Semantics Checker gecompileerd te worden. Hercompilatie van de Semantics Checker wordt dus vermeden. Ook source-code van de Semantics Checker hoeft niet verspreid te worden aan gebruikers van de SIMPLEXYS tools.
De Rule Compiler met daarin geïntegreerd de Semantics en de Protocol (zie bijlage 6) Checkers wordt een tiental Kb's groter. Dit is echter nauwelijks een bezwaar in de huidige tijd waarin werkgeheugen goedkoop is.

Het huidige programma CHK41 is eenvoudig te veranderen in een unit, die gebruikt kan worden door de Rule Compiler. De bestaande main body van het programma moet een procedure worden die de Rule Compiler kan aanroepen.
De Semantics Checker kan blijven werken met het 2-dimensionale connections array Con. Kleine aanpassingen in de code zijn nodig om het array te initialiseren met de juiste gegevens, die in de objectstructuur zijn opgeslagen. De routines die de eigenlijke controles uitvoeren kunnen geheel onaangetast blijven. De nummers die de regels hebben in de algemene collection van regels kunnen als indices gebruikt worden. I.p.v. statische declaratie van Con is het nodig Con dynamisch te maken zodat:

- Con alleen geheugen gebruikt, wanneer de semantiek gecontroleerd wordt. Con verbruikt dus geen geheugen, dat de Rule Compiler misschien nodig heeft.
- De array-grootte niet vast ligt tijdens compilatie, omdat het aantal regels dat gecontroleerd moet worden nog niet vast ligt.

Niet zozeer het dynamisch maken van dit array is problematisch, maar dat het array in twee dimensies open is. Pascal is niet in staat te werken met een array dat in twee dimensies open is. In de volgende paragraaf wordt de oplossing hiervoor gegeven.

Connection array dynamisch

Statisch is Con als volgt gedeclareerd:

```
type
    CCSet = set of CC;

var
   Con : array[1.._N,1.._N);
   P   : ^ccset;
```

_N is een constante die het aantal regels van het SIMPLEXYS expertsysteem aanduidt. Con[I,J] refereert in Pascal naar een cel in Con. Een pointer P naar deze cel is in Pascal als volgt te verkrijgen: P:=@Con[I,J].

Een referentie naar een cel kan ook worden uitgerekend. In het algemeen geldt voor een 2-dimensionaal array A, dat het adres van een cel gevonden wordt door:

```
P:=Ptr(Seg(A),Ofs(A)+(J-LowBound(Index2))*SizeOf(CellSize)+(I-
LowBound(Index1)*(HighBound(Index2)-LowBound(Index2))*SizeOf(CellSize));
```

Substitueren we de waarde die geldt voor Con, dan krijgen we (onder aanname dat SizeOf(CCSet) gelijk is aan 2):

```
P:=Ptr(Seg(Con),Ofs(Con)+(J-1)*2+(I-1)*_N*2;
```

Deze formule kan ondergebracht worden in een functie:

```
function ConCell(I, J: integer) : CCSetPtr;
begin
  P:=Ptr(Seg(Con),Ofs(Con)+(J-1)*2+(I-1)*_N*2;
end;
```

ConCell(I,J)^ is equivalent aan Con[I,J].
Er is echter één verschil: _N kan in ConCell variabel i.p.v. constant zijn.

Een nadeel van het gebruik van ConCell is dat een enorm snelheidsverlies optreedt. Voor iedere referentie binnen het array moet een functie aangeroepen worden, die bovendien op zich minder efficiënt wordt gecompileerd als Con[I,J].
Deze beide vertragende factoren kunnen opgelost worden door toepassing van:

- Gebruik van macros ter voorkoming van calls. Een macro heeft net als een procedure een identifier (naam) en een (meestal klein) aantal regels programmacode. Wanneer een macro gerefereerd wordt, wordt geen call gemaakt naar de macro, maar worden de programmaregels van de macro ingevoegd in het programma op de plaats van de macro referentie.
  Helaas kent Turbo Pascal geen Pascal-macros, maar gelukkig wel inline-macros. Een inline-macro is een stukje machinetaal dat in een programma geplaatst wordt op de plaats van zijn referentie. Een inline-macro wordt in Turbo Pascal gedeclareerd als een procedure of een functie zonder begin/end statements. De inline-procedure of -functie kan parameters ontvangen via de stack en het functionresultaat retourneren als iedere andere functie. Aan inline-procedure of -functies wordt bovendien geen entry en exit code toegevoegd, wat gewoonlijk bij andere procedures en functies wel gebeurt.
- Eenvoudige Pascal code die honderdduizenden keren wordt uitgevoerd, kan soms beter in machinetaal geprogrammeerd worden. Onder eenvoudig is te verstaan: enkele regels Pascal code waarin met elementaire Pascal-typen wordt gewerkt (char, byte, boolean, integer, word, longint). Een snelheidswinst van een factor vijf t.o.v. Pascal code met dezelfde functionaliteit is soms te behalen.

Beide technieken zijn in listing 17 gebruikt.

```
var
   CCArrayPtr : pointer;  { Must be in DS Con }
   NumOfRules : word;     { Must be in DS Con }

function Con(I,J: word) : CCSetPtr;
{ This is a macro-function; i.e. this function isn't called but is }
{ inserted in the program code whenever Con is referenced          }
{ Pre: NumOfRules both CCArrayPtr local }
{ Pre: SizeOf(CCSet)=2                  }
{ Post: CCSetPtr as DX:AX               }
inline(
  $59/                   { pop  cx             ; CX:=J                  }
  $49/                   { dec  cx             ; CX:=CX-1               }
  $D1/$E1/               { shl  cx,1           ; CX:=CX*2               }
  $A1/NumOfRules/        { mov  ax,NumOfRules  ; AX:=NumOfRules         }
  $D1/$E0/               { shl  ax,1           ; AX:=AX*2               }
  $5B/                   { pop  bx             ; BX:=I                  }
  $4B/                   { dec  bx             ; BX:=BX-1               }
  $F7/$E3/               { mul  bx             ; AX:=AX*BX              }
  $01/$C8/               { add  ax,cx          ; AX:=AX+CX              }
  $C4/$3E/CCArrayPtr/    { les  di,CCArrayPtr  ; ES:=Seg(CCArrayPtr);   }
                                               ; DI:=Ofs(CCArrayPtr)    }
  $01/$F8/               { add  ax,di          ; AX:=AX+DI              }
  $8C/$C2                { mov  dx,es          ; DX:=ES                 }
);
```

*Listing 17: Functie om in een N * N open array een cel te adresseren.*

I.p.v. Con[I,J] kan in CHK41 nu Con(I,J)^ geschreven worden. De geheugenruimte voor het array moet aan het begin van CHK41 gereserveerd worden door:

```
getmem(CCArrayPtr,NumOfRules*NumOfRules*2);
```

Deze macro is getest. CHK41 werkt sneller met dan zonder de functie-macro wanneer de compiler-opties Range en Stack checking aan staan. Dit komt omdat in de inline-code geen Range checking gedaan wordt. De inline-macro is ±10% langzamer dan Con[I,J] is, in het geval dat CHK41 gecompileerd wordt zonder de opties Range en Stack checking.
Een kleine winst is nog te behalen indien de array-indices vanaf 0 zouden beginnen. Beide decrement instructies in de inline code komen dan te vervallen[1]. Wanneer de Rule Compiler objectgeoriënteerd geprogrammeerd is, is dit het geval en zijn de items in een (rule-)collection genummerd vanaf 0.

1. Arrays met indices vanaf 0 zijn dus sneller. In C bestaan omwille van de snelheid alleen arrays beginnend met indices vanaf 0.

Bijlage 6 PET41

In deze bijlage wordt beschreven hoe problemen, die optreden wanneer PET41 geïntegreerd wordt in de Rule Compiler, kunnen worden opgelost. Net als bij CHK41 in de vorige bijlage is er het probleem dat type-declaraties in PET41 afhankelijk zijn van de gecompileerde regelbeschrijving zoals de Rule Compiler die opslaat in RINFO.QQQ.
Naast dit probleem kan PET41 de correctheid van slechts een klein protocol analyseren (zie tabel 2, pag. 13). Een oplossing voor dit probleem vormt de keuze van een andere datastructuur voor het opslaan van contexten in de Protocol Checker. Deze oplossing wordt in deze bijlage uitgewerkt.
De voor en nadelen van het integreren van de Protocol Checker in de Rule Compiler zijn dezelfde als het integreren in de Rule Compiler van de Semantics Checker. Deze zijn in het begin van bijlage 5 genoemd.

Dynamische typen

In de Protocol Checker worden sets gedeclareerd afhankelijk van constanten die in de file RINFO.QQQ worden gedeclareerd. Een voorbeeld:

```
type
    SizeS = 0 .. _N_S;
    State = set of SizeS;
```

_N_S is een constante uit RINFO.QQQ, die het aantal State-rules in het gecompileerde regelbestand aanduidt. Wanneer de Protocol Checker geïntegreerd wordt in de Rule Compiler kunnen sets niet meer statisch gedeclareerd worden als in bovenstaand voorbeeld. Een tweetal oplossingen is denkbaar:

- Definieer elke set maximaal groot: set of 0..255. Dit is weinig elegant: het kost onnodig veel geheugen en set-operaties als +, - en * kosten meer runtijd dan wanneer kleinere sets gebruikt zouden zijn.
- Maak een eigen dynamische set-type met daarop de setoperaties geïmplementeerd als functies of procedures. Een voordeel boven standaard sets is dat dergelijke sets geschikt gemaakt kunnen worden om meer dan 256 elementen te kunnen bevatten, zodat dergelijke limieten mede verdwijnen. Een dynamische set kan geïmplementeerd worden als een record met drie velden:

```
type
    SetPtrT = ^SetT;
    SetT    = record
                LenInBits  : word;
                LenInBytes : word;
                Elements   : array[0..8191] of byte;
              end;
```

Listing 18: Type declaratie dynamische sets.

Er kan niet gewerkt worden met statische variabelen van het type SetT. Sets moeten altijd dynamisch gedeclareerd worden (via SetPtrT). Een "InitSetPtr" procedure kan een pointer naar een dergelijke set aanmaken en initialiseren. De grootte van de set moet meegegeven worden aan InitSetPtr:

```
function InitSetPtr;
{ Post: Generate a new dynamic set. The fields LenInBits and LenInbytes
        are set according to the requested number of elements the set
        should hold. The newly created set is returned empty.
}
var
   Bytes     : word;
   NewSetPtr : SetPtrT;
begin
  Bytes:=succ(succ(Bits) div 8);
  getmem(NewSetPtr,4+Bytes);
  with NewSetPtr^ do
    begin
      LenInBits:=Bits;
      LenInBytes:=Bytes;
      fillchar(Elements,Bytes,0);
    end;
  InitSetPtr:=NewSetPtr;
end;
```

Listing 19: InitSetPtr.

Alle Pascal setoperatoren zijn ook voor deze dynamische sets te maken. In de volgende tabel zijn deze operatoren opgenomen. In de tabel zijn a en b sets van gelijke typen en is e een element van een dergelijke set.

Tabel 5: Setoperatoren Pascal.

Operator type	Operation	Programmering
a = b	gelijk	compare op bit-rijen
a <> b	ongelijk	compare op bit-rijen
a <= b	subset	bytewise (not(b) and a =0)
a >= b	superset	bytewise (not(a) and b =0)
e in a	Element in set	Test bit
Include(a,e)	Voeg element toe	Set bit
Exclude(a,e)	Verwijder element	Unset bit
a + b	vereniging	bytewise or op bit-rijen
a - b	verschil	bytewise not(b) and a
a * b	doorsnede	bytewise and op bit-rijen

Een complete library met bovenstaande operatoren is gerealiseerd voor set van het type SetT, met daarin toegevoegd de relaties kleiner en groter.dan voor dynamische sets. Hoewel in standaard Pascal deze relaties niet bestaan, maar dat toch van kleiner (of groter) dan bij sets gesproken kan worden, wordt in de volgende paragraaf toegelicht.

Verandering datastructuur

Het opslaan van contexten in een datastructuur is het voornaamste probleem van de Protocol Checker. Contexten moeten in de Protocol Checker zodanig opgeslagen worden dat na te gaan is of een context al dan niet in de datastructuur is opgeslagen.
Een **context** is een verzameling state-rules. Het aantal mogelijke contexten is groot: $2^{\#\text{state-rules}}$ (wanneer er 32 state-rules zijn dan zijn meer dan 4 miljard mogelijke contexten denkbaar). Meestal komen slechts een beperkt aantal contexten in een SIMPLEXYS protocol voor, zodat de Protocol Checker de meeste permutaties van state-rules niet tegenkomt.

De bestaande oplossing slaat een context op in arrays. Van een context (een Pascal-set van state-rules) wordt een key (integer) gemaakt waarmee middels **hashing** een positie wordt bepaald in een array (dit hashing-array kan 5000 contexten bevatten). Wanneer de context moet worden toegevoegd in het hashing-array en de plaats in het array reeds bezet is, dan wordt de key opnieuw aangeboden aan het hashing-algoritme. Dit gebeurt maximaal 5 keer. Wanneer dan nog niet een vrije positie gevonden is, wordt de context geplaatst in een overloop-array dat maximaal 500 contexten kan bevatten. Dit overloop-array is niet gesorteerd: de context wordt achteraan toegevoegd.
Wanneer gecontroleerd moet worden of een context al dan niet is opgeslagen in de datastructuur, wordt middels de key die van de set gemaakt wordt, eerst in het hashing-array gezocht m.b.v. de hashing-functie. Wordt de context niet gevonden, terwijl nog geen lege positie in het array is bereikt (na maximaal 5 keer), dan moet het overloop-array lineair doorzocht worden. Al met al een complex geheel dat maximaal slechts 5500 contexten kan bevatten.

In de Protocol Checker komt dus een zoekoperatie voor: van een context moet bepaald kunnen worden of deze al dan niet is opgeslagen in de datastructuur. Zoek-algoritmen op standaard datastructuren (zoals bijvoorbeeld een binary search algoritme in een gesorteerde collection) gaan uit van het bestaan van operaties als: kleiner dan, gelijk aan en groter dan, zodat items in de datastructuur met elkaar vergeleken kunnen worden.
Helaas bestaan de relaties kleiner/groter dan niet voor sets (contexten) in Pascal. Toch is een dergelijke ordening denkbaar: een set is eigenlijk niets anders dan een bit-rij. Een bit-rij is op haar beurt weer op te vatten als een getal in het tweetallig stelsel. Pascal integer-typen als byte, integer en longint zijn niets anders dan bit-rijen. Deze bit-rijen hebben een vaste lengte: resp. 8, 16, 32 bits waarop wel relaties als kleiner/groter dan gedefiniëerd zijn.

Bit-rijen van gelijke lengte kunnen onderling v.l.n.r. bytes-gewijs vergeleken worden. Zijn de eerste bytes van de te vergelijken bit-rijen gelijk, dan worden de volgende bytes vergeleken, totdat een onderling verschil optreedt of totdat het einde van de rijen bereikt is. Stopt dit proces dan geldt de relatie van de laatst gedane onderlinge vergelijking van de bit-rijen als relatie tussen de bit-rijen.

Een functie kan in Pascal eenvoudig geprogrammeerd worden, die het bovenstaande proces uitvoert. Aan de functie worden als parameters twee sets gegeven. De functie retourneert -1 indien Rij1<Rij2, 0 wanneer beide sets gelijk zijn en 1 wanneer Rij1>Rij2. In machinetaal kan een dergelijke functie net zo eenvoudig als in Pascal gerealiseerd worden:

```
function SetCompare;
{ Post: returns -1 if P1^.Elements<P2^.Elements }
{       returns  0 if P1^.Elements=P2^.Elements }
{       returns  1 if P1^.Elements>P2^.Elements }
label
     L1, Error, ExitL;
begin
  asm
    mov   bx,ds          { bx:=ds          Save ds                }
    lds   si,[bp+6]      { ds:si:=@P2^                            }
    les   di,[bp+10]     { es:di:=@P1^                            }
    cld                  { set direction autoincrement            }
    cmpsw                { compare LenInBits of two sets          }
{$IFOPT R+}
    jne   Error          { if sets not equal length than error    }
{$ENDIF}
    mov   cx,[si]        { cx:=LenInBytes                         }
    xor   ax,ax          { ax:=0           Result:=0              }
    cmpsw                { Step over LenInBytes                   }
    repe  cmpsb          { compare sets                           }
    jl    L1             { if P1^<P2^ then goto L1                }
    je    ExitL          { if equal then exit (Result=0)          }
    inc   ax             { ax:=1           Result:=1              }
    jmp   ExitL          { goto exit                              }
L1: dec   ax             { ax:=-1          Result:=-1             }
ExitL:
    mov   ds,bx          { ds:=bx          Restore ds             }
    mov   sp,bp          { Overruled standard exitcode            }
    pop   bp
    retf  8
{$IFOPT R+}
Error:
    mov   ds,bx          { ds:=bx          Restore ds             }
  end;
  writeln('Run-time error: SetCompare tries to compare sets of different lengths');
  halt(1);
{$ELSE}
  end;
{$ENDIF}
end;
```

Listing 20: Een set-compare functie.

De prefix repe "*repeat while equal*" die de instructie cmpsb "*Compare string bytewise*" vooraf gaat, herhaalt cmpsb totdat een verschil in de bit-rijen wordt bemerkt of totdat de teller cx nul wordt. De pointers P1 en P2 in SetCompare wijzen naar de te vergelijken dynamische sets. In Turbo Pascal wordt een integer-function result teruggegeven in register AX.

Nu sets onderling vergeleken kunnen worden, is het mogelijk contexten onder te brengen in een sorted-collection. In totaal kunnen maximaal 16.380 items worden ondergebracht in een collection. Dit is aanzienlijk meer dan in de hashing-arrays (het maximum was daar 5.500). In

een sorted-collection wordt gezocht met een binair-search algoritme. De tijdcomplexiteit van een binair-search algoritme is $^{2}\log(n)$, waarbij n het aantal items in de collection is[1]. Voor een volle collection is dit: $^{2}\log(16.380)=14$.
Een nadeel van het toepassen van een collection is dat dit gemiddeld iets trager zal zijn dan hashing. Het hashing-algoritme heeft echter meer beperkingen: maar maximaal 5000 items kunnen in het array worden ondergebracht. Dit array zal nooit voor de volle 100% benut worden. Wanneer een item in het overloop-array gezocht moet worden, duurt dit veel langer dan in de collection.
Het aantal state- en trigger-rules dat de Protocol Checker kan verwerken kan groter zijn dan 256, wanneer met dynamische sets gewerkt wordt. De exacte grootte van het protocol dat door de aangepaste Protocol Checker gecontroleerd kan worden, is sterk afhankelijk van het protocol en dus minder voorspelbaar. Wanneer het werkgeheugen niet groot genoeg blijkt, kan de Rule Compiler met daarin de Semantics en Protocol Checkers als DPM programma gecompileerd worden, zodat voor de dynamische variabelen ook het extended geheugen van de computer gebruikt kan worden.

De Protocol Checker heeft niet alleen een datastructuur waarin de aan- of afwezigheid van contexten gecontroleerd wordt. De volgorde waarin contexten worden opgeslagen in de datastructuur is ook van belang. De datastructuur (een sorted-collection) is alleen gericht op het snel kunnen bepalen of een context in de datastructuur aanwezig is of niet. Een tweede (niet gesorteerde) collection kan voor dit doeleinde gebruikt worden. Wordt een context toegevoegd in de gesorteerde collection, dan wordt deze context achteraan deze tweede collection toegevoegd. Voorheen werd deze volgorde bijgehouden in het array "Map".

1. zie [AHO87] pag. 155 e.v.

Bijlage 7 Vergelijkingstests

In deze bijlage zijn twee tests opgenomen om tot een eerlijke vergelijking qua snelheid en code grootte van het inference mechanisme te komen. Wanneer een vergelijkingstest wordt uitgevoerd, dienen de testomstandigheden zoveel mogelijk gelijk te zijn:

- Dezelfde compiler moet gebruikt worden (BP 7.0).
- Resultaten van compilaties met dezelfde compiler opties moeten vergeleken worden.
- Hetzelfde regelbestand moet worden aangeboden.

Objective SIMPLEXYS ('OS') wordt vergeleken met de originele versie van SIMPLEXYS ('ORG') en met een versie van SIMPLEXYS ('S') die is geoptimaliseerd volgens **alle** optimalisaties die worden aangedragen in §3.3.5. Het eerlijkst is de vergelijking tussen OS en de geoptimaliseerde versie van SIMPLEXYS ('S'). In OS wordt duplicate code namelijk ook vermeden. De andere optimalisaties uit §3.3.5 zijn niet in OS verwerkt, omdat ze niet op OS van toepassing zijn.

Snelheidsvergelijking

Om het verschil in snelheid tussen SIMPLEXYS en Objective SIMPLEXYS te bepalen is gemeten aan een AND-OR net (Michie's example, zie §4.3 [BLO90]):

```
Gegeven:
   A,B,C,D,E zijn primitieven
   F = A and B
   G = C and D
   H = not(E)              {in originele voorbeeld was: H = E}
   J = B and G
   K = G and E
   X = (F and H) or (J and K)

Gevraagd: X
```

Listing 21 is de vertaling van dit AND-OR net naar een SIMPLEXYS regelbestand. Met de timer routine afkomstig uit de unit USubs wordt de tijd gemeten, die nodig is om het net 1000 keer te evalueren. De timer wordt gestart voorafgaand aan de inferencing (in de INITG sectie). Na afloop van de inferencing plaatst de timer routine de verstreken tijd in beeld (in honderdsten van een seconde). De metingen van de test in deze bijlage zijn verricht op een 20 MHz Intel 80386 DX.

```
USES
    USUBS

INITG
     Timer(true)

EXITG
     Timer(false)

RULES

A: 'A'
FACT
initially FA

B: 'B'
FACT
initially FA

C: 'C'
FACT
initially FA

D: 'D'
FACT
initially FA

E: 'E'
FACT
initially FA

F: 'F'
A and B

G: 'G'
C and D

H: 'H'
not(E)

J: 'J'
B and G

K: 'K'
G and E

X: 'X'
(F and H) or (J and K)

WaitLoop: 'Wait loop'
MEMO
INITIALLY FA

RUNNING: 'analyzing animals'
STATE
INITIALLY TR
THEN GOAL: X
THEN TR: WaitLoop

READY: '1000 runs done'
WaitLoop >= (999)
THEN DO writeln('1000 runs done');

PROCESS

ON READY FROM RUNNING TO *
```

Listing 21: Regelbestand voor snelheidstest.

Welke regels geëvalueerd worden, is afhankelijk van de waarden van de primitieven en de debugging opties. De debugging opties zijn alle gedeactiveerd: de interessante grootheid is nl. de maximale snelheid van redeneren en niet de snelheid van het uitvoeren van de debug-code. Omdat de debugging optie 'Validate' af staat, wordt de evaluatie van een expressie gestopt wanneer het resultaat van de expressie bekend is.
In de listing zijn alle primitieven FA. Alleen de volgende regels worden dan tijdens een run geëvalueerd:

- evaluatie van achtereenvolgens X, F, A, J, B, veroorzaakt door goal 'X' van de State-rule Running;
- evaluatie van de trigger rule 'READY'.

We zien dat zes regels niet gebruikt worden (C, D, E, G, H, K). Een tweede test is uitgevoerd zodat ook deze regels nodig zijn om de waarde van X vast te kunnen stellen. Alle primitieven zijn hiertoe TR gemaakt.
Doordat in beide situaties dezelfde aantallen regels in het bestand zijn opgenomen, is de overhead tussen twee runs nagenoeg hetzelfde.

De metingen zijn verricht met verschillende instellingen van de compiler opties Range en Stack Checking. De resultaten zijn in de volgende tabellen weergegeven.

Tabel 6: Run-tijden (alle primitieven FA).

	OS		S		ORG	
Compiler opties	10ms	%	10ms	%	10ms	%
$R-,S-	66		55		60	
$R-,S+	83	25.7	71	29.1	82	36.7
$R+,S-	142	115	225	309	291	385
$R+,S+	159	141	236	329	302	403

De kolommen aangeduid met '%' geven de groei in procenten aan t.o.v. de situatie met compiler opties $R-,S-.

Eerste conclusies geldig voor deze testsituatie:

- OS is 20% langzamer dan S.
- Stack Checking vertraagt de drie programma's de executietijd met grofweg 30%. OS steekt hier iets gunstiger af.
- OS is veel minder gevoelig voor Range Checking.

Worden dezelfde tests uitgevoerd in DPM dan zijn de resultaten als volgt:

Tabel 7: Run-tijden DPM (alle primitieven FA).

	OS		S		ORG	
Compiler opties	10ms	%	10ms	%	10ms	%
$R-,S-	104		61		66	
$R-,S+	131	26.0	82	34.4	93	40.9
$R+,S-	209	101	307	403	401	507
$R+,S+	241	132	329	439	423	541

Conclusies na deze eerste twee tests:

- De invloeden van Range en Stack Checking op de run-tijden zijn ongeveer gelijk aan de vorige test situatie.
- S wordt 11% langzamer als DPM programma. OS echter 58%. Dit grote verschil komt doordat in DPM een segment register (of segment part van een pointer) niet in directe relatie staat tot het fysieke base adres van een segment. Het fysieke base adres wordt gevonden middels een table look-up met als index het segment register. Deze tabel bevindt zich in het werkgeheugen. Op een Intel 80X86 processor vindt iedere adressering in het werkgeheugen plaats op basis van een segment en een offset adres. Om te voorkomen dat voor iedere adressering in het werkgeheugen een segmenttabel geraadpleegd moet worden, is op de processor een cache aanwezig die de laatste 32 segment translations bevat[1].

  S maakt nagenoeg alleen gebruik van drie segmenten: één code, één data en één stack segment. In OS zijn er zeer veel verschillende segmenten: de verschillende methods zijn alle far en iedere dynamische datastructuur heeft ook zijn eigen segment. De hit-rate van de cache is dus veel lager in OS dan in S, hetgeen de grotere vertraging van de executietijd verklaart.

Tabel 8: Run-tijden (alle primitieven TR).

	OS		S		ORG	
Compiler opties	10ms	%	10ms	%	10ms	%
$R-,S-	88		82		88	
$R-,S+	111	26.1	104	26.8	116	31.8
$R+,S-	198	125	357	335	412	368
$R+,S+	225	156	368	349	439	399

Conclusies na deze test in vergelijking tot de eerste test situatie:

- OS is nu maar 7% langzamer dan S. Ten opzichte van de eerste testsituatie is de overhead tussen twee runs nagenoeg gelijk. In deze situatie worden meer regels geëvalueerd, zodat geconcludeerd mag worden, dat de regels fractioneel sneller geëvalueerd worden in OS dan in S.

1. zie [IBM92] appendix A.

- Dat de overhead tussen twee runs in S veel geringer is dan in OS, komt doordat enkele operaties daar zeer optimaal geprogrammeerd zijn. Bijvoorbeeld het false maken van het array _Busy, geschiedt in S door één machinetaal instructie. In OS is dit niet mogelijk, de variabelen van een regel worden in OS per regel in een record opgeslagen bij andere variabelen die deel uitmaken van de regel.
  In ORG wordt iedere variabele van een regel apart geadresseerd (als in OS). Het verschil in snelheid tussen ORG en OS blijkt in deze situatie niet meetbaar.

Hoewel geen opmerkelijke resultaten mogen worden verwacht, zijn voor de volledigheid de resultaten in DPM voor deze testsituatie:

Tabel 9: Run-tijden DPM (alle primitieven TR).

	OS		S		ORG	
Compiler opties	10ms	%	10ms	%	10ms	%
$R-,S-	137		88		94	
$R-,S+	176	28.5	127	44.3	137	45.7
$R+,S-	291	112	495	463	577	514
$R+,S+	336	145	532	605	610	549

Opmerkingen:

- Tijdens het testen heb ik gemerkt, dat de resultaten niet altijd reproduceerbaar bleken in BP 7.0. Soms week een resultaat van een meting 6% af van het resultaat van een meting onder dezelfde testcondities. Dit verschil was zelfs reproduceerbaar, ook onder condities waarbij geen overbodige drivers in DOS geïnstalleerd waren, die onverwacht processortijd verbruiken (MOUSE.SYS, SMARTDRV.SYS e.d.).
  Het vermoeden bestaat, dat dit verschijnsel in verband staat met de eerste 2 fouten van BP 7.0 die in bijlage 3 genoemd worden. Ondanks dit, stemmen de resultaten overeen met hetgeen verwacht mocht worden, zodat de gemaakte conclusies verantwoord zijn.
- De uitgevoerde snelheidstests waren alleen gericht op het testen van de redeneersnelheid. OS voert in ieder geval Test, Do en History code sneller uit dan S, doordat in OS de betreffende code direct (zonder case) gevonden kan worden.
  De overige processortijd die in een real-time nodig is (voor het verwerken van interrupts, schermafhandeling e.d.), is niet te beïnvloeden door de Inference Engine en zijn dus niet in deze beschouwing opgenomen.

Absolute redeneersnelheid

Door de testsituatie (Test2), waarin alle regels geëvalueerd werden, af te zetten tegen de situatie waarin slechts X, F, J, A en B geëvalueerd werden (Test1), is het aantal regels dat de Inference Engine per seconde kan verwerken te bepalen. Een derde meting is hieraan toegevoegd (Test0): door de thelse 'THEN GOAL X' te verwijderen worden 5 regels en een thelse minder geëvalueerd dan in Test1. De tests zijn uitgevoerd zonder Range en Stack Checking, zodat de maximale redeneersnelheid bepaald kan worden.

Tabel 10: Run-tijden verschillend aantal regels.

	OS		S		ORG	
$R-,S-	10ms	vers.	10ms	vers.	10ms	vers.
Test0	44		28		33	
Test1 (+5+thelse)	66	22	55	27	60	27
Test2 (+6 regels)	88	22	82	27	88	28
Totaal verschil		44		54		55

Bij de drie programma's is de tijd, die nodig is voor het uitvoeren van een thelse, evengroot als de tijd nodig voor het evalueren van een regel. Het aantal regels dat per seconde geëvalueerd wordt, is eenvoudig uit te rekenen: wanneer de thelse als een regel wordt beschouwd, dan worden 12 regels per run meer geëvalueerd in Test2 dan in Test0. De runtijden uit tabel 10 zijn bepaald door 1000 keer het regelbestand te evalueren.

Tabel 11: Redeneersnelheid.

	OS	S	ORG
#regels/sec	27.3 10^3	22.2 10^3	21.8 10^3

Deze aantallen regels per seconde gelden voor een 20 MHz Intel 80386 DX. Op snellere hardware neemt de redeneersnelheid evenredig toe. Een 50 MHz Intel 80486 is vijf maal sneller: 2.5 maal op basis van verschil in kloksnelheid (wanneer een cache geheugen aanwezig is) en nog ongeveer een factor 2 door interne caching van instructies en verdergaande optimalisaties waardoor instructies sneller uitgevoerd worden. Op een dergelijk snelle hardware kunnen ongeveer 125 duizend regels per seconde geëvalueerd worden. De opvolger van de 486 zal dit aantal wederom verdubbelen.

Een relativerende waarschuwing: de redeneersnelheden in deze bijlage zijn bepaald door te meten aan eenvoudige evaluatie regels. Onder andere testsituaties kunnen andere resultaten verkregen worden. Sneller dan in de geteste situaties worden Fact en Memo rules geëvalueerd. Langzamer wordt het geheel bij complexere Eval regels en door Test regels die complexe condities testen.

Vergelijking code-grootte

Het beste inzicht in de verschillen qua code-grootten wordt verkregen door na compilatie de information dialog box uit het compiler menu te raadplegen. Hierin staat apart vermeld, hoe groot de code na compilatie is geworden. Dit is niet af te lezen aan de grootte van een exe-file. De grootte van een exe-file is namelijk een sommatie van: code, data en grootten. Ook segment informatie wordt opgeslagen in exe-files. Deze factor doet DPM programma's groeien.
In de bestaande versies van SIMPLEXYS wordt getracht zo vaak mogelijk overbodige code van de Inference Engine niet mee te compileren.
Twee oorzaken dat code niet meegecompileerd wordt zijn aan te wijzen:

- een debug optie is niet geactiveerd, zodat debug-code niet wordt toegevoegd:

```
{$IFDEF DEBUG2}
  _show_applied('NOT',u);
{$ENDIF}
```

- bepaalde typen regels komen niet voor:

```
{$IFDEF THERE_ARE_MEMOS}
  for rule:=_first[_FACT] to _last[_FACT] do
    _thelse(rule,_R[rule]);
{$ENDIF}
```

In Objective SIMPLEXYS komt conditionele compilatie niet voor; i.e. alle code wordt altijd meegecompileerd. Voor het verkrijgen van gelijke testomstandigheden worden alle debug-opties in S en ORG geactiveerd en wordt een regelbestand aan de Rule Compiler aangeboden met daarin één regel van elk type. Het regelbestand waarmee getest is:

```
rules

F1: 'F1'
FACT
initially TR

A1: 'A1'
ASK

T1: 'T1'
BTest 1>0

E1: 'E1'
not(F1) or A1 or not(T1) or M1

M1: 'M1'
MEMO
initially FA

Running: 'Processing'
STATE
initially tr
then goal: E1

process

ON Running from Running to *
```

Listing 22: Regelbestand om code-grootte te testen.

Evenals bij de snelheidstests zijn ook hier de metingen verricht met verschillende instellingen van de compiler opties Range en Stack Checking. De resultaten zijn in de volgende tabel weergegeven.

Tabel 12: Code-grootten.

	OS		S		ORG	
Compiler opties	bytes	%	bytes	%	bytes	%
$R-,S-	15920		14320		15632	
$R-,S+	16320	2.5	14528	1.4	15824	1.2
$R+,S-	16832	5.7	17392	21.4	19248	23.1
$R+,S+	17248	8.3	17616	23.0	19440	24.4

Ook in deze tabel wordt in de kolom '%' de groei van de code aangegeven t.o.v. de situatie $R-,S-.

Conclusies:

- OS is ruim 11% groter dan S wanneer Range Checking af staat. Ook dit verschil wordt goeddeels veroorzaakt door de vele methods die in OS aanwezig zijn: aan iedere method worden namelijk ongeveer 12 bytes[1] entry en exit code toegevoegd. Bovendien zijn de calls naar methods in OS complexer: de calls zijn altijd far, en voor virtuele methods moet het adres van een method gezocht worden in de VMT.
- OS groeit sterker dan S, indien Stack Checking geactiveerd wordt. Dit komt doordat Stack Checking o.a. plaatsvindt bij het begin van een procedure/functie. In OS zijn veel meer methods aanwezig, dan dat er procedures en functies in S zijn.
- Range Checking heeft op S bijna vier keer zoveel invloed dan op OS. Oorzaak hiervoor is dat in OS meer met pointers gewerkt wordt, die niet onder invloed van Range Checking staan en omdat in OS meer toekenningen (van constanten) aan variabelen plaatsvinden die tijdens compilatie gecontroleerd worden i.p.v. tijdens run-tijd. Dat in OS meer gewerkt wordt met toekenningen aan variabelen die tijdens compilatie gecontroleerd kunnen worden, komt doordat de methods meer specifiek zijn dan de procedures/functies in S (denk hierbij aan het bestaan van aparte evaluatiemethods voor elk regeltype één).
- Tegenover dit laatste voordeel van OS staat dat Range Checking een geringer effect sorteert, omdat pointerstructuren minder gecontroleerd worden op fouten. In het algemeen geldt, dat dynamische structuren met uiterste zorgvuldigheid geprogrammeerd moeten worden. Gebeurt dit niet, dan zijn crashes niet van de lucht en duurt het debuggen lang.

Voor de volledigheid zijn dezelfde tests uitgevoerd, wanneer het expertsysteem als een DPM programma gecompileerd wordt. De code-grootten in tabel 13 zijn ongeveer een half procent groter geworden. De verhoudingen zijn niet veranderd.

---

1. zie [BOR88] pag. 210.

Tabel 13: Code-grootten in DPM.

	OS		S		ORG	
Compiler opties	bytes	%	bytes	%	bytes	%
$R-,S-	16009		14497		15819	
$R-,S+	16400	2.4	14713	1.4	16016	1.2
$R+,S-	16938	5.8	17582	21.2	19429	23.8
$R+,S+	17329	8.2	17798	22.7	19626	24.1

Bijlage 8 Foutjes in implementatie SIMPLEXYS

In deze bijlage zijn enkele punten opgenomen, die in de bestaande versie van SIMPLEXYS eenvoudig te verbeteren zijn.

Run-Time Error

Het volgende triviale expertsysteemp wordt door de Rule Compiler, Semantics en Protocol checkers geaccepteerd. Desondanks treedt in de Inference Engine (SIM41) een Range Check error op.

```
RULES

Running: 'Processing'
STATE
initially TR

PROCESS
ON Running FROM Running to *
```

De Range Check error wordt veroorzaakt door:

```
var
   fs : 1.._N_S;

   fs:=_first[_state]-1;
```

Omdat in dit denkbeeldige expertsysteem alleen een State-rule aanwezig is, is _first[state] gelijk aan 1 en krijgt fs dus de waarde 0. Twee oplossingen zijn denkbaar:

- Declareer fs vanaf 0.
- Sta niet toe dat een SIMPLEXYS expertsysteem uit louter State-rules bestaat.

De laatste oplossing is de meest logische: een expertsysteem met louter State-rules kan niet als expertsysteem door het leven gaan, hooguit kan dan van een Petrinet simulator gesproken worden. De Rule Compiler kan de controle uitvoeren, dat een expertsysteem niet alleen uit State-rules mag bestaan.

Objective SIMPLEXYS verwerkt dit triviale expertsysteem correct.

Thelses van Memo-rules worden niet uitgevoerd

In de Inference Engine wordt in iedere run getracht de Thelses van Memo-rules uit te voeren:

```
{$IFDEF THERE_ARE_MEMOS}
 {thelses of MEMO rules}
 for _rule:=_first[_MEMO] to _last[_MEMO] do
   _thelse(_rule,_R[_rule]);
 {$ENDIF}
```

De procedure header van de routine _thelse:

```
procedure _thelse(rule: _sizeN; value: bool);
```

Het uitvoeren van de Thelses van Memo-rules gaat vooraf aan het uitvoeren van Thelses van State-rules. State-rules zetten meestal de goals; i.e. starten de inferencing van een run. Tijdens de inferencing van een run wordt van Memo-rules alleen de vorige waarde gebruikt (_S). Aan de procedure Thelse uit bovenstaande code zou dus ook _S[rule] i.p.v. _R[rule] moeten worden meegegeven; zodat die Thelses worden uitgevoerd, die overeenstemmen met de waarde die een Memo als resultaat van een evaluatie tijdens de run retourneert.

Wordt dit gedaan, dan wordt door een ander foutje nog altijd niet de Thelses van Memo-rules uitgevoerd. In het begin van de procedure _thelse is nl. de volgende test opgenomen:

```
if _R[rule]=UD then exit;
```

Omdat de eigenlijke inferencing (door de goals van State-rules) van een run nog niet heeft plaatsgevonden en omdat _R van een Memo-rule aan het begin van een run UD gezet is, wordt geen enkele Thelse van Memo-rules uitgevoerd. De oplossing is dus:

```
{$IFDEF THERE_ARE_MEMOS}
 {thelses of MEMO rules}
 for _rule:=_first[_MEMO] to _last[_MEMO] do
   _thelse(_rule,_S[_rule]);
 {$ENDIF}
```

in de procedure thelse moet staan:

```
if value=UD then exit;
```

In Objective SIMPLEXYS wordt de method Thelse uit de objectklasse ORule overruled voor Memo-rules. Dit is in Objective SIMPLEXYS noodzakelijk omdat daar geen parameter 'value' wordt meegegeven aan de method Thelse. Thelse grijpt in Objective SIMPLEXYS impliciet toe op de variabelen van de rule waarvan de thelses worden uitgevoerd. Meestal moet de method Thelse dus toegrijpen op _R ('Value' in OS) maar voor Memo-rules op _S ('PrevValue' in OS).

Array grootte

In Pet41 wordt een array bijgehouden ('Map') voor het bijhouden van een 'route' (om van de begintoestand te geraken in een bepaalde context) voor iedere context die in het hashing-array opgenomen is. De hashingstructuur voor het opslaan van contexten bestaat uit twee arrays: 'Main' en het overlooparray 'Extra'. Map moet evenveel cellen hebben als beide eerdergenoemde arrays te zamen. Map moet dus als volgt gedeclareerd worden:

```
var
   Map : array[SizeMain + SizeExtra] of integer;
```

Referenties

[AHO87] AHO, Alfred v. en J.E. Hopcroft, J.D. Ullman,
Data Structures and Algorithms.
Amsterdam: Addison-Wesley, 1987

[BLO90] Blom, Johannes Abraham,
The SIMPLEXYS experiment: real time expert systems in patient monitoring.
Technische Universiteit Eindhoven, 1990.
Proefschrift.

[BOR88] TURBO PASCAL Reference guide.
Borland International Inc., 1988.

[DUR92] Duren, E.C.G.J. van,
Expertsystemen: principes en toepassingsmogelijkheden in de productie-automatisering.
Vakgroep Meten en Regelen,
Faculteit der Elektrotechniek,
Technische Universiteit Eindhoven, 1992.
Stageverslag.

[EZZ89] Ezzell, Ben,
Object oriented programming in Turbo Pascal 5.5.
Amsterdam: Addison-Wesley, 1990

[IBM92] OS/2 Version 2.0 Volume 1: Control Program.
IBM International Technical Support Centres, 1992.
IBM report: GG24-3730-00

[JAC90] Jackson, Peter,
Introduction to expert systems.
Amsterdam: Addison-Wesley, 1990.

[SLU92] Sluis, Edwin van de,
Design and implementation of a real-time expert system shell.
Instituut Vervolgopleidingen,
Technische Universiteit Eindhoven, 1992.
Eindverslag van de ontwerpersopleiding Technische Informatica.

[STA91] Jaarboek Intelligente Software,
Jaarboek van het tijdschrift Kennissystemen.
Rijswijk: Stam tijdschriften, 1991.

[WIN93] Sebastian, Kiesel,
DLLs programmieren, Über alle Grenzen hinweg,
WinDos, Das Windows-Magazin der DOS International, jrg. 3 (1993), no. 1, p. 64.

Index

B

binding
  early 55
  late 55

C

context 83

D

DPM, Dos Protected Mode 12
DPM, zie Dos Protected Mode

H

hashing 83

I

infer, zie redeneren
inheritance 53
interface
  naar databases, spreadsheets 12

K

kennis
  acquisitie 10
  eliciatatie 10
  model 10

M

method 49
  abstract 54
  statisch 55
  virtueel 55
modularisatie 10

O

object 49
  dynamisch 56
  inheritance 53
  klasse 51
    sub 53
    super 53
  structuur 57
OOP 49

P

polymorfisme 55

R

redeneren
  achterwaarts 7
  voorwaarts 7

S

self 57

V

VMT, Virtual Method Table 56