# Dijken en formele methoden

# DIJKEN EN FORMELE METHODEN

INTREEREDE

Prof.dr.ir. L.M.G. Feijs

Technische Universiteit Eindhoven

# INTREEREDE

Uitgesproken op vrijdag 12 april
1996 aan de
Technische Universiteit Eindhoven

Prof.dr.ir. L.M.G. Feijs

Mijnheer de Rector Magnificus,
Dames en heren,

## Inleiding

De Nederlanders zijn goed in dijkaanleg en formele methoden. Dat mag best wel eens gezegd worden en we mogen er ook wel trots op zijn. Maar dat is niet genoeg; we moeten er ook gebruik van maken. Het kan van vitaal belang worden voor onze welvaart en ons leven. Dat onze welvaart deels van de dijken afhangt, is makkelijk in te zien. De Nederlanders veroverden vruchtbare stukken land op de chaotische toestand die 'zee' heet. Die zijn nu landbouwgebied, maar ook ruimte om te wonen en te werken.
Dat onze levens van de dijken kunnen afhangen, is maar al te duidelijk. En de dreiging komt niet alleen van buiten, maar ook van binnen. Denk maar aan wat er 's winters kan gebeuren als de rivierdijken niet in orde zijn. Maar nu moet ik natuurlijk uitleggen wat die formele methoden zijn en waarom ik het belang ervan vergelijk met dat van dijken.

## Formele methoden

Het gaat over software-ontwikkeling, dat wil zeggen, het maken van computerprogramma's. En dan bedoel ik niet alleen het schrijven van de programma's, want dat is natuurlijk wel belangrijk. Software-ontwikkelaars doen echter nog veel meer. Veel software is 'embedded', dat wil zeggen, maakt deel uit van een apparaat of een machine. Ook voor een bedrijf is een computerprogramma nooit een alleenstaand doel, maar hoort het bij een produkten- of dienstenpakket. Het moet dus passen in een groter geheel, gegevens verwerken die uit andere programma's komen en gegevens produceren die door gebruikers en andere programma's te interpreteren zijn. Het programma moet kunnen werken met afgesproken computers en besturingssystemen en het moet de benodigde schaarse middelen zoals geheugenruimte, rekentijd en netwerkcapaciteit op ordelijke wijze delen met andere programma's. Daarom houden software-ontwikkelaars zich vooral bezig met communicatie en overleg. Ze bouwen applicatiekennis op. Ze overleggen met deskundigen die proberen te bepalen wat de gebruiker wil, ze stellen specificaties op, ze lezen elkaars specificaties en programma's, ze ontwikkelen hulp-programma's die zelf weer programma's manipuleren, ze ontwikkelen testen, ze zoeken naar fouten, ze maken projectplannen, ze meten hun voortgang, enzovoort. Om een aantal essentiële aspecten van softwareontwikkeling in goede banen te leiden worden methoden bedacht en die bereiken dan al of niet een zekere verspreidingsgraad. Deze methoden zijn zelf aan een soort evolu-

tie en ook aan modeverschijnselen onderhevig. Zo'n tien jaar geleden waren 'gestructureerde' methoden (SA/SD) erg in, nu zijn dat 'object-georiënteerde' methoden (OO). Formele methoden vormen ook een klasse van methoden. Ze worden vooral uitgedragen door onderzoekers aan wetenschappelijke instellingen en ze zijn minder populair bij de software-ontwikkelaars zelf. Natuurlijk is deze classificatie van methoden wat simplistisch en in feite zijn er allerlei interessante vormen van overlapping.

Formele methoden zijn systematische manieren om computerprogramma's te ontwikkelen, waarbij wordt gewerkt met gedragsbeschrijvingen die met wiskundige precisie geanalyseerd kunnen worden. Ik bedoel niet, dat elke stap met een expliciet wiskundig bewijs onderbouwd moet worden om van een formele methode te kunnen spreken. Maar wel dat dat voor één of meer belangrijke aspecten moet kunnen. Als zich dan eens een moeilijke of een verdachte ontwerpstap voordoet, kan die tenminste precies geanalyseerd worden.

Formele methoden zijn nuttig om drie redenen. Allereerst betekent software-ontwikkeling omgaan met computers, en dat zijn nu eenmaal formele machines, die niet zo welwillend zijn om kleine vergissingen goedwillend te interpreteren. Ten tweede kunnen computers pas als gereedschap ingezet worden als het probleem met voldoende precisie in de taal van de computer gecodeerd is. Tenslotte evenaart de complexiteit van programma's die van de ingewikkeldste dingen die de mens gecreëerd heeft en is bezig die ruim te overschrijden. Om goed met die complexiteit om te gaan hebben we technieken nodig, gebaseerd op wiskunde die gaat over structuur en modulariteit. Formele technieken kunnen een belangrijke rol spelen bij het beheersen van die complexiteit. Ze kunnen helpen te voorkomen dat het maken van programma's ontaardt in een chaos. Formele technieken zijn als de dijken, die het land beschermen tegen de chaos van de zee.

## Formaliteit is niet alles

Soms wordt het belang van formaliteit in de informatica ook wel eens overschat. Het betoog is dan ongeveer als volgt: computers zijn formele machines die slechts geprogrammeerde instructies volgen en daarbij niets dan nullen en enen manipuleren. In principe kunnen we hun gedrag dus begrijpen, mits we zelf voldoende formeel redeneren. Persoonlijk denk ik dat dit een misvatting is en dat bij het ontwerpen van systemen die geprogrammeerde computers bevatten, allerlei andere, niet-formele zaken ook een rol spelen. Zo zijn er de ergonomische aspecten van het gebruik van

het systeem. En het feit dat de software-ontwerpers mensen zijn, die met elkaar moeten overleggen, zodat bepaalde sociale factoren een rol spelen. Bovendien worden we soms geconfronteerd met deelsystemen waarvan we niet alle eigenschappen kennen. Je kunt natuurlijk definiëren dat informatica een vorm van wiskunde is, net zoals je kunt zeggen dat mechanica dat is. Maar het is om diverse redenen niet zo dat het mogelijk, laat staan handig is, om alles wat in de realiteit van belang is, wiskundig te analyseren. Ook in de klassieke mechanica is het in principe zo dat je met de wetten van Newton alle bewegingen kunt analyseren, mits je natuurlijk nog wat beginvoorwaarden en wat materiaaleigenschappen weet. Maar als ik morgen een rauw ei in de keuken laat stukvallen, dan is het onzin om de precieze plaats en omvang van de landing uit te rekenen volgens de wetten van de mechanica. Dan kijk ik gewoon waar de troep ligt en veeg die dan op basis daarvan op. Zo is het ook in de toegepaste informatica: sommige dingen lenen zich niet (of nog niet) voor formele analyse.

Formele methoden zijn vrijwel onmisbaar als het erom gaat echt moeilijke programma's correct te bewijzen en als het erom gaat de afspraken betreffende samenwerkende deelprogramma's precies vast te leggen. Maar ik mag dat niet omkeren en zeggen dat ze overal toegepast moeten worden. Ik pleit hier voor een gulden middenweg.

## Historische ontwikkeling

Nu een stukje geschiedenis. Voor speciale klassen van problemen zijn er de afgelopen decennia goede resultaten geboekt met bepaalde formele methoden. Bijvoorbeeld bij het vastleggen van de syntaxis van talen, zoals van een van de eerste programmeertalen, FORTRAN. De regels die zeggen wat een geldige zin in de taal is, worden vastgelegd met een soort formules, de zogenaamde produktieregels. Een zin hoort pas bij de taal, als je dat via toepassing van de produktieregels kan vaststellen. Hier zien we de essentie van een formele methode. Belangrijke gegevens worden vastgelegd in formules en die moet je in principe als een robot kunnen toepassen. Natuurlijk is het niet de bedoeling dat ontwikkelaars als robots te werk gaan; het gaat om het principe dat in geval van twijfel een specificatie op een objectieve manier geduid kan worden. Voor het vastleggen van de syntaxis van talen is deze techniek al lang gemeengoed geworden. Voor het redeneren over programma's en voor het afleiden van programma's, gegeven hun specificaties, zijn ook formele methoden uitgevonden. Bijvoorbeeld het door Hoare in 1969 voorgestelde systeem van redeneerregels om na te gaan of een pre-conditie,

een programma en een post-conditie met elkaar in overeenstemming zijn [1]. Of het door Dijkstra in 1975 voorgestelde systeem van rekenregels om voor een gegeven programma en een gegeven post-conditie de zwakste pre-conditie uit te rekenen [2]. Op dit gebied is sindsdien erg veel goed werk verricht, met name ook aan deze universiteit. En hoewel deze formele systemen best wel aansluiten bij het programmeren in Pascal of in C, wordt in de industriële praktijk het overgrote deel van de programma's in deze talen gemaakt, zonder dat gewerkt wordt met pre-condities, post-condities en invarianten. En als er al pre-condities, post-condities of invarianten zijn, dan zijn die heel informeel. Dat wil zeggen, als je de regels er als een robot op probeert toe te passen, blijkt dat ze te onvolledig zijn om toegepast te kunnen worden. Of ze zijn wel volledig, maar inconsistent, dat wil zeggen, ze leiden tot uitspraken van het soort 1 + 1 = 3. Het typische van deze formele systemen is dat ze bedoeld zijn om correctheidsbewijzen van programma's te geven, of andersom programma en bewijs samen af te leiden. De pre- en post-condities worden opgeschreven in een standaard-vorm van logica en het kunnen opstellen van pre- en post-condities als zodanig wordt nauwelijks gezien als een onderwerp van studie. Dat is ook niet zo verwonderlijk, want het zijn nog steeds pittige oefeningen om voor enigszins slimme algoritmen (denk aan sorteerproblemen) de correctheidsbewijzen helemaal formeel uit te werken, en dat terwijl de pre- en post-condities ervan zonder probleem op één regel op te schrijven zijn.

Een andere belangrijke stap voorwaarts was de publikatie van VDM, de Vienna Development Method in 1978 [3]. De methode heet zo omdat veel van de ontwikkeling eraan plaats vond in het laboratorium van IBM in Wenen. Onderdeel van deze methode was een taal, Meta IV geheten. De grammatica van deze taal was ongeveer zo complex als die van een typische programmeertaal, maar in plaats van als programma's moeten de VDM-beschrijvingen als specificaties of als 'modellen' worden opgevat. Oorspronkelijk bedoeld om de semantiek (d.w.z. de betekenis) vast te leggen van nieuwe programmeertalen, bleek de taal ook krachtig genoeg om allerlei andere systemen mee te modelleren: gegevensbanken, stukken van besturingssystemen en nog veel meer. Er kwamen diverse varianten van VDM, waarbij ik met name het werk van Jones wil noemen, die het idee van een 'rigoureuze' aanpak helder naar voren bracht: de methode schrijft niet voor dat alle stappen echt formeel onderbouwd worden, maar zorgt wel dat in geval van twijfel een probleem op een objectieve manier geanalyseerd kan worden. Ik meen dat dit een erg verstandig idee is. Het typische van beschrij-

vingen in VDM is dat ze niet alleen gaan over gemakkelijk te specificeren maar moeilijk te ontwerpen programma's van algoritmische aard, maar dat ze betrekking hebben op een heel systeem. Dat wil zeggen dat er ook een abstract, maar toch precies model van de variabele interne systeem-toestand in zit; en dat er een hele lijst van programmaatjes gespecificeerd is die daar dan op werken.
Deze specificaties lijken nog op programma's in Pascal of in C, ook met variabelen en if-then-else constructies en dergelijke. Maar daarnaast worden verzamelingenleer en logica gebruikt en dus zijn de specificaties echt niet geschikt om als programma geëxecuteerd te worden. Daar zijn ze ook niet voor bedoeld, trouwens. Ze zijn bedoeld om orde te scheppen in de enorme brij aan details die ontstaat als een echt groot systeem gespecificeerd en ontworpen moet worden. En dat is iets anders dan het correct bewijzen van een slim algoritme. Ook moeilijk en ook belangrijk, maar de accenten liggen anders.
Voor de volledigheid moet ik ook vermelden dat er diverse methoden zijn die enigszins vergelijkbaar zijn met VDM, bijvoorbeeld Z. Maar ik heb het hier vooral over VDM gehad, omdat ikzelf in het begin van de jaren tachtig hiermee al in aanraking kwam en ermee experimenteerde en omdat dat toen op mij een grote indruk maakte.

Een andere ontwikkeling kwam uit wiskundig gericht onderzoek en deed rond 1975 haar entree in de wereld van de informatica: het specificeren van data-typen met behulp van vergelijkingen. Het werkt zoals in de wiskunde, waar een vermenigvuldiging op gehele getallen gespecificeerd kan worden door twee vergelijkingen, namelijk $0 \times X = 0$ en $(N+1) \times X = (N \times X) + X$. En omdat alleen maar gewerkt wordt met vergelijkingen, kan een specificatie zo geïnterpreteerd worden dat hiermee ook de waardenverzameling van het data-type zelf vastligt. De aanpak wordt onderbouwd door een wiskundige stelling die zegt, dat er precies één waardenverzameling is met niet te veel en niet te weinig objecten erin.
In de praktijk wordt er vaak wat losjes mee gewerkt en velen menen de aanpak te begrijpen als zijnde niets anders dan het idee, dat waardenverzameling en operaties erop bij elkaar horen en dat de specificatie of de realisatie ervan een soort software-module vormen. Formeel gezien is er nog wel meer aan de hand; er zijn ook vervelende technische problemen zoals de vraag wat te specificeren voor N : 0 of voor de het eerste element van een lege lijst. Ook bleek dat met een techniek die termherschrijving heet, dit soort specificaties nog als programma te executeren zijn ook. Het simulatiegereedschap PSF, dat ik afgelopen jaar bij Philips gebruikt heb, werkt zo [4].

## Enkele ervaringen

Op het Natuurkundig Laboratorium van Philips is er in de periode 1984 tot 1994 betreffende formele specificaties een interessant stuk onderzoek en ontwikkeling uitgevoerd, dat ik van dichtbij mocht meemaken en waaraan ikzelf ook heb mogen bijdragen.
Het onderzoek gebeurde deels in het kader van subsidieprojecten van de EEG en in samenwerking met de universiteit van Utrecht. Jonkers ontwierp de taal COLD [5,6]. In deze taal werd een aantal uitgangspunten van VDM gecombineerd met een deels nieuwe wiskundige onderbouwing, geïnspireerd door de wiskundige onderbouwing van de data-typeaanpak met vergelijkingen. Deze taal COLD werd een heel bouwwerk, met allerlei interessante stukken wiskunde en logica erin opgenomen, inclusief predikatenlogica, import en export van modules, vergelijkingen, pre-condities, post-condities en nog meer. Soms dachten we ook wel eens: „wordt de taal zelf niet te complex?". Maar anderzijds hadden we ook al wat ervaring met het specificeren van echte systemen en gezien de diversiteit van de details die daarin opdoken, was toch vaak de conclusie: nee, in de praktijk willen we dit handig kunnen uitdrukken en daarom moet constructie zus-en-zo toch echt in de taal. In feite willen we nog veel meer. De praktijk wacht ook niet totdat de specificatietalen klaar zijn als er een leuk idee is. Integendeel, regelmatig worden we verblijd met nieuwe mechanismen die worden toegevoegd aan programmeertalen: vererving, uitgestelde evaluatie, virtuele procedures en ga zo maar door, tot en met de verworvenheden van Visual Basic toe. Hier ligt een enorm spanningsveld en het toepassen van formele methoden in de praktijk is er niet gemakkelijker op geworden.
Maar dit even terzijde, laat ik nu het verhaal over COLD afmaken.

We hebben COLD op diverse plaatsen, vooral binnen Philips, kunnen uitproberen. Zo heb ikzelf bijvoorbeeld kunnen werken aan specificaties van geometrische CAD-software voor beeldbuisontwerp, van optimalisatiesoftware voor plaatsingsmachines, en aan de specificatie van diverse audio-video-componenten. Het voornaamste gebruik van COLD is gelegen in de SPRINT-methode, waarmee ons team van het Natuurkundig Laboratorium, samen met een van de ontwikkellaboratoria binnen CE (consumenten-elektronica) zekere vormen van succes heeft weten te boeken [7]. Deze SPRINT-methode bouwt voort op COLD, maar voegt er nog een heleboel aan toe: gereedschappen en methodische aanwijzingen. Tijdens de ontwikkeling van SPRINT hebben wij veel geleerd. Laat ik enkele punten noemen. Ten eerste dat voor overdracht van een nieuwe methode

veel tijd ingeruimd moet worden. Het opschrijven van methodische aanwijzingen kost bijvoorbeeld veel tijd. Eigenlijk ging het ook niet zo dat eerst de methode ontwikkeld werd en toen overgedragen. De methode is ontwikkeld tijdens een periode van intensieve samenwerking in gemeenschappelijke projecten waaraan produktontwikkelaars en onderzoekers deelnamen. Een belangrijk mechanisme van overdracht is ook het doorstromen van mensen vanuit het laboratorium waar het relatief fundamentele onderzoek plaatsvond, naar het produkt-ontwikkellaboratorium. We hebben ook geleerd dat het maken en onderhouden van gereedschappen een taak is die veel tijd kost. Wij hebben dat in het algemeen onderschat. Gereedschappen zijn erg nuttig, maar de ontwikkelkosten ervan moeten gedeeld kunnen worden door een voldoende groot aantal gebruikersgroepen. Het is niet mijn bedoeling om te beweren dat SPRINT de uiteindelijke oplossing van alle problemen is. Zelfs niet voor de betreffende ontwikkelgroep. Nieuwe eisen dienen zich aan, het karakter van het produkt verandert geleidelijk en gelukkig zitten noch wij, noch de andere methoden-ontwikkelaars stil.

## Interoperabiliteit

Twee belangrijke groepen problemen in de informatica hebben te maken met efficiëntie en interoperabiliteit. Er is een efficiëntieprobleem als een programma wel goed werkt, alleen teveel rekentijd, teveel geheugen of teveel van een ander schaars middel nodig heeft. Natuurlijk zijn er fundamentele grenzen aan de efficiëntie, en zo kan bijvoorbeeld een handelsreizigersprobleem niet in lineaire tijd opgelost worden. Maar er is reeds veel bekend en voor bijvoorbeeld sorteerproblemen valt er te kiezen uit een hele catalogus van efficiënte algoritmen, en in de praktijk kan er dan één uitgekozen en gekopieerd worden.
De interoperabiliteitsproblemen lijken zich veel minder te lenen voor een wetenschappelijke aanpak. Er is een interoperabiliteitsprobleem als twee deelsystemen niet goed samenwerken. Vaak is niet expliciet vastgelegd wat 'goed samenwerken' betekent, maar blijkt er wel degelijk een zekere verwachting te zijn van hoe de dingen zouden moeten samenwerken. Laat ik enkele voorbeelden geven.
Als de plug van de CD-I-speler niet in mijn oude TV past, en in de winkel weten ze ook niet goed welk kastje er precies tussen moet, dan is er een interoperabiliteitsprobleem. Of als er een e-mail binnenkomt vanaf een Macintosh-computer en bij het afdrukken is de helft van de tekst verdwenen omdat het afbreken van de regels op een Unix-machine anders gaat dan op de Mac, dan noem ik dat een interoperabiliteitsprobleem. Nu zijn dit problemen die

iedereen gewoon kan meemaken met deelsystemen uit de winkel. Maar ditzelfde vindt aan één stuk door plaats binnen de ontwikkellaboratoria waar software gemaakt wordt. Het te maken systeem wordt steeds opgedeeld in deelsystemen waarvan sommige nog helemaal gemaakt moeten worden, terwijl andere gekocht worden of gemaakt worden door bestaande deelsystemen en -programma's aan te passen. Dat aanpassen is meestal aanpassen van de software, die is immers zo flexibel. De eisen van de gebruiker veranderen terwijl het systeem ontwikkeld wordt, en tegelijk verandert het aanbod van in te kopen en aan te passen deelsystemen en -programma's. Van de gebruikte programmeertalen en gereedschappen komen trouwens ook regelmatig nieuwe versies uit. In een dergelijke situatie is een slordigheidsfout gauw gemaakt of is een misverstand gauw ontstaan. En dan is er een interoperabiliteitsprobleem.
Vroeg of laat komt zo'n probleem boven water. Als het vroeg is, dan is het meestal zo verholpen, maar als het laat is, dan spreken we van een 'bug', een storende fout [8]. „De mensen die werken in het ontwikkellaboratorium zouden geen fouten mogen maken en de eisen van gebruikers en de produkten van de toeleveranciers zouden bevroren moeten worden"... Precies, dat gaat niet.

Ik meen dat formele specificatietechnieken een rol kunnen en ook moeten spelen, opdat in de ontwikkellaboratoria ordelijk met deze problematiek omgegaan kan worden. Oefening met talen zoals LOTOS, PSF, Z, ExSpect, VDM of COLD is van onschatbare waarde om specificaties van deelsystemen te kunnen schrijven en lezen. Predikatenlogica in pure vorm voldoet hier niet, want de specificaties worden zelf te groot. De formele specificatietechnieken zijn nog niet af en het is wel zeker dat de huidige talen en gereedschappen weer opgevolgd zullen worden door een volgende generatie. Daarover straks meer, maar volgens mij zijn de genoemde talen nu al goed bruikbaar. En wie ze wil proberen maar het nog moeilijk vindt, die wil ik graag proberen te helpen.

Nu wil ik nog even verder ingaan op de problematiek van het specificeren. Meestal wordt een afgesproken of veronderstelde specificatie niet precies ingevuld, ook niet als in eerste instantie beweerd wordt dat dat wel het geval is. Een typisch vraaggesprek hierover verloopt als volgt.
Ontwerper: „Wat wij gemaakt hebben klopt precies met standaard X".
Vraag: „klopt het ook helemaal?".
Ontwerper weer: „Nou, een aantal parameters die wij niet nodig hebben, hebben we wel weggelaten, dat scheelde tijd en ruimte".
Vraag: „Maar is het dan niet toch minder dan de standaard wat jullie gemaakt hebben?".

Ontwerper: „Ik zou daar toch niet te min over denken, want wij hebben ook nog een aantal extra's geïmplementeerd, die ons erg nuttig lijken". Tot zover het gesprek. In twee stappen is dan vastgesteld dat het hier gaat om een 'extended subset'.
Het lezen en schrijven van abstracte en toch precieze specificaties van deelsystemen is soms niet gemakkelijk. De formele methoden zijn er wel, maar ze zijn nog niet volmaakt, en de tijd om te oefenen ontbreekt vaak. Een officiële standaard is vaak behoorlijk goed opgeschreven en kennis erover is meestal makkelijk te verkrijgen. Maar als een 'extended subset' van een gegeven systeem of taal gemaakt wordt, dan ligt de verantwoordelijkheid en de zorg voor het creëren en onderhouden van een goede beschrijving toch weer bij de maker. In de wereld van de personal computers (PC's) heeft het interoperabiliteitsprobleem een naam. Het heet CONTROL-ALT-DELETE.

Eerder beweerde ik dat onze welvaart en ons leven af kunnen gaan hangen van formele methoden (zoals ze al lang afhangen van dijken). Door met een formele methode te specificeren en aspecten van systeemcorrectheid te analyseren, zijn wij technisch in staat om een gezonde basis te leggen voor een goed begrepen en grotendeels correct systeem. In combinatie met allerlei andere degelijke ontwerp- en verificatietechnieken, zoals programma-inspecties, testen en vastleggen van procedures kunnen we de situatie bereiken dat we voor de software van een systeem alles gedaan hebben wat we konden, om te zorgen dat het systeem begrepen en correct is. Ook testen kan zorgen voor kwaliteitsverbeteringen van software, maar als er geen precieze specificaties zijn, wat betekent dat testen dan? Ik meen dat formele specificaties een rol horen te spelen als we echte kwaliteit willen afleveren.

Hangt ons leven ervan af? Een fly-by-wire systeem in een vliegtuig is ook een fly-by-software systeem. In beveiligingssystemen voor treinen zit ook software. En als je een ambulance moet bellen via de telefooncentrale, dan gaat dat alleen goed als de embedded software in de telefooncentrale goed werkt. Apparatuur voor medische diagnose zit trouwens ook boordevol software. Laat ik eens een retorische vraag stellen: lijkt het U een goed idee als de ambulance ook via de e-mail opgeroepen kan worden? Misschien lijkt het nu nog wat eng, maar misschien is het een geruststellende gedachte dat er nog altijd een CONTROL-ALT-DELETE combinatie op de computer zit... Hoewel?

Misschien lijkt dit maar een klein aantal gezochte voorbeelden. Misschien is het nog maar de vraag of de Nederlanders zo nodig hun eigen vliegtuig-, treinen-, telefooncentrale-

en communicatie-softwareindustrie moeten hebben. Natuurlijk is veel software niet zo kritisch, omdat dat spelletjes-, educatie- en kantoorsoftware is. De meeste software komt toch uit het buitenland. Inderdaad, maar als we dit soort gedachten zouden doortrekken dan komen we uit bij het inzicht dat als we niets doen, we ook niets fout doen.

## Economische aspecten

Inmiddels is ook het economische aspect in beeld gekomen. Er is sprake van een hele nieuwe industrie: de softwareindustrie. Een schone industrie, waarin veel geld verdiend kan worden en die ook heel goed gedaan kan worden in een land zonder bijzondere bodemschatten. Ja, toch.. iets bijzonders is wel nodig: een goede kennis-infrastructuur. En we kunnen kiezen of we willen proberen om sterk mee te doen in deze software-industrie of niet. De software-industrie zal steeds meer verweven raken met de andere industrieën.
Dat is evident voor de elektronische industrie en de vliegtuigindustrie, maar ook voor de 'diensten-industrie': banken, verzekeringen, telecommunicatie, medische zorg, personenverkeer en goederenverkeer.
Ik ben van mening dat het belang van software wel ingezien wordt, maar te langzaam. In de elektronische industrie is het denken in termen van het produceren van kastjes sterk verankerd.
Soms wordt al gedacht in termen van families van kastjes of systemen die uit meerdere kastjes bestaan. Maar in veel gevallen zou nog beter vanuit de software gewerkt kunnen worden.
Als de software het essentiële produkt is, dan is het kastje nog slechts een hulpmiddel om die software ten uitvoer te brengen. Dan is het misschien beter om geen eigen kastje de wereld in te sturen, alleen voor die ene applicatie. Dan is het misschien beter gebruik te maken van de kastjes die iedereen al heeft en aldus snel een grote omzet te halen met het produkt: de software.
De software-industrie is een harde industrie. Kennis kan snel verouderd zijn. Processorsnelheid en geheugen verdubbelen om de pakweg twee à drie jaar. En de snelheid waarmee informatie verspreid kan worden is enorm toegenomen, onder andere via het Internet en via CD-ROM. Bovendien wordt er in de software-industrie een nieuw spel gespeeld: ik noem het 'interoperabiliteits-spelletje' [9]. Het spel is hard en er zijn geen spelregels, maar het gaat ongeveer zo: bedrijf A en bedrijf B zijn in competitie om een softwareprodukt X te verkopen. Dat moet werken in een omgeving waar al softwareprodukten Y en Z draaien. De tijd is pas rijp op een bepaald moment: als er genoeg krachtige hardware is. Wie te vroeg op de markt is, heeft al zijn ideeën gratis weggegeven en verliest. Wie

te laat is verliest ook. Van de leveranciers die ongeveer op tijd zijn, wint diegene die zorgt voor de minste interoperabiliteitsproblemen tussen X, Y en Z. De leverancier van Y en Z heeft dus de beste kaarten. In de volgende ronde staan X, Y en Z in het veld en gaat het om de verkoop van een vierde produkt. Enzovoort. Ik meen dat het hier van strategisch belang is om goed te kunnen specificeren, redeneren en testen. Zolang de ontwikkelaars binnen een bedrijf nog een soort interoperabiliteit-spelletje met elkaar spelen, al of niet bewust, lacht de concurrentie in haar vuistje.

## Communicatieprotocollen

Interoperabiliteitsproblemen doen zich ook voor bij communicatieprotocollen en ze zijn dan extra vervelend. Dat komt omdat een protocol een gedistribueerd programma is, dat wil zeggen, het bestaat uit meerdere deelprogramma's die tegelijk actief zijn, maar als verschillende processen, elk in een eigen omgeving en met een eigen snelheid. Die deelprogramma's communiceren dan regelmatig, via een onderliggende communicatiedienst (die zelf ook vaak weer een protocol bevat). Als elk van die deelprogramma's zich in 100 verschillende toestanden kan bevinden, en we beschouwen een stelsel met vijf delen, dan zijn er $10^{10}$ toestanden denkbaar. Bovendien zijn er vaak verschillende leveranciers voor die delen.
Uitputtend testen van zo'n systeem is meestal ondoenlijk. Het alternatief is om te zorgen dat het protocol correct is en dat elk van de componenten precies voldoet aan de specificatie die bij de correctheidsanalyse voor de component is aangenomen. Testen van zo'n component is dan wel te doen, (dat heet dan conformance-testing), maar let op de veronderstellingen. Die zijn: 1) correctheid van het protocol en 2) een specificatie per component. Dus toch weer formele technieken, of anders iets wat daar heel veel op lijkt.

Nu is onderzoek betreffende het specificeren en redeneren over parallelle programma's pas later op gang gekomen dan dat over sequentiële programma's en over data-typen. Van bijzonder belang is de calculus van communicerende systemen (CCS), die in 1980 door Milner werd voorgesteld [10]. Een aantal wetmatigheden wordt hier expliciet gemaakt met vergelijkingen. Ook in Nederland is hieraan sinds het begin van de jaren tachtig fundamenteel en toegepast onderzoek gedaan, waarbij het algebraïsche karakter van de theorie sterk benadrukt is. Met name bedoel ik het werk van Bergstra, Klop en Baeten betreffende de algebra van communicerende processen (ACP) [11,12]. Zeker ook vermeldenswaard is de taal LOTOS, waaraan vanuit Nederland met name Brinksma veel heeft bijgedragen;

deze taal, die verwant is aan CCS en ACP heeft reeds de status van internationale standaard en wordt ook in de praktijk gebruikt [13].

## Onderzoeksthema's

Terwijl er een enorme schat aan fundamenteel inzicht op het gebied van parallellisme werd opgebouwd, werd in diezelfde tijd in het COLD project besloten om parallellisme nog niet in de taal op te nemen, en eigenlijk deden de groepen die met VDM en Z werkten dat ook. Daar was ook een goede reden voor, namelijk dat deze talen eigenlijk al behoorlijk complex waren geworden, zelfs zonder parallellisme. In het geval van COLD is dat gemis later opgevangen door zekere voorzieningen in de SPRINT-methode. En nog steeds is het combineren van bijvoorbeeld modularisering, parallellisme, data-typen en programmeervariabelen een moeilijke kwestie. Er is reeds gepionierd op dit onderzoeksgebied, maar helemaal begrepen en goed in kaart gebracht is het niet.

Je kan ook niet zeggen, stop nu maar eens met het specificeren van simplistische data-typen zoals 'stacks' of met het rekenen aan het alternerende-bitprotocol. Omdat we nog niet alles begrijpen van de logica en de wiskunde van specificeren en redeneren, is het belangrijk dat het fundamentele onderzoek doorgaat. En daar hoort ook bij om kleine voorbeelden te analyseren. We zijn nog niet in staat om alle verschijnselen die we goed willen specificeren en waarover we goed willen kunnen redeneren, te vangen in onze talen. Misschien is het ook wel niet haalbaar om een ideale specificatietaal te hebben die alles kan. Ik acht dat voor de komende tien jaar in elk geval niet haalbaar. En dus moeten we in de industriële praktijk werken met combinaties van talen. Er dient ook rekening gehouden te worden met wat gebruikelijk is en met standaarden. Talen als SDL, COLD, PSF, TTCN, LOTOS, Z, message sequence charts (MSC) [14], maar ook programmeertalen als Prolog, Gofer, C++ enzovoort zullen in allerlei wisselende combinaties gebruikt moeten worden. Hoe moet dat? Hoe gaan we om met de soms subtiele syntactische en semantische verschillen tussen deze talen? Proberen in de praktijk, formuleren van zinvolle betrekkingen tussen de talen en uitzoeken van de theoretische vragen die hieruit voorkomen. Ik meen dat er veel belangrijk werk te doen ligt, en ik zal proberen hieraan bij te dragen.

## De kracht van logische specificaties

Graag wil ik nog eens proberen uit te leggen wat de kracht en het voordeel is van een beschrijving in logica ten opzichte van een program-

ma. Er zijn vele vormen van logica, zoals equationele logica, propositielogica, predikatenlogica, temporele logica en dynamische logica.
Pre- en post-condities kunnen gezien worden als onderdeel van dynamische logica. Een zin in een logica is een bewering die, voor een gegeven stand van zaken of voor een gegeven opeenvolging van gebeurtenissen en toestanden, of *waar* of *niet waar* is. Bijvoorbeeld als ik praat over een rij R van namen (rij ter lengte N), dan is er een zin in de predicatenlogica die zegt dat die rij gesorteerd is:
VOOR ALLE i $((0 < i < N) \Rightarrow (R[i] \leq R[i+1]))$.
Hierbij is R[1] de eerste naam in de rij, R[2] de tweede, enzovoort.
Neem even aan dat namen bestaan uit een of meer voorletters en een achternaam en dat de $\leq$ relatie betrekking heeft op de alfabetische volgorde van de achternamen. Nu kan ik praten over een programma P dat toekenningen kan doen aan een variabele rij R en beweren dat na uitvoering van P geldt dat R nog dezelfde namen bevat als tevoren en dat bovendien R gesorteerd is. Voor sommige keuzen van P is deze bewering waar; zulke programma's P heten sorteerprogramma's. Andere keuzen voor P maken deze bewering niet waar. Bijvoorbeeld een programma dat louter nullen invult in R. Tot zover de logica, nu de programma's.

Er zijn vele vormen van programma's, zoals machine-taalprogramma's, imperatieve programma's, logische programma's, functionele programma's, parallelle programma's en allerlei combinaties daarvan. Een zin in een programmeertaal is een samengestelde opdracht aan een automaat. Voor bepaalde invoergegevens voert de automaat de opdracht uit en komt dan tot een bepaald resultaat.
Voor sommige programma's, de reactieve programma's, moeten we dit wat ruimer interpreteren en zeggen dat alle acties van de omgeving invoer zijn en alle reacties van het programma uitvoer.

Als een programma gespecificeerd is door een logische bewering, dan zijn er ofwel nul ofwel zeer veel programma's die aan die specificatie voldoen. Bijvoorbeeld mijn eerdere specificatie van een sorteerprogramma laat oneindig veel implementaties toe. We kennen inderdaad vele sorteerprogramma's, bijvoorbeeld bubble-sort en quick-sort. Maar kan ik de zaak ook omdraaien? Kan ik bijvoorbeeld een aantal invoer-uitvoercombinaties laten zien en tegen een programmeur zeggen: „zo, nu heb je het gezien, dat was de specificatie van het programma P"?
Bijvoorbeeld, de invoer is de rij van twee namen: J. Jansen, P. Pieterse en de uitvoer is hetzelfde, J. Jansen, P. Pieterse.
Moet de programmeur het nu weten? Welnee, hij kan bijvoorbeeld

denken dat het voor alle invoerrijen de bedoeling is dat ze naar de uitvoer gecopieerd worden (de invoerrij was namelijk toevallig al gesorteerd).
Goed, dan nog een invoer-uitvoercombinatie: invoer P. Pieterse, J. Jansen, uitvoer J. Jansen, P. Pieterse. Moet de programmeur het nu weten? Nee, nog steeds niet, hij kan bijvoorbeeld denken dat het de bedoeling is om te sorteren volgens de lengte van de namen ('Pieterse' is namelijk toevallig langer dan 'Jansen'). Ik kan dit gedachtenexperiment erg lang voortzetten. Ik doe dat nu niet maar concludeer alvast, dat een programma oneindig veel gedragingen kan hebben en dat 'voordoen' dus geen sluitende specificatietechniek is.

Kan ik het misschien anders doen? Kan ik een voorbeeldprogramma nemen, bubble-sort en tegen een programmeur zeggen:
„hier heb je een prototype, zo moet het, maar dan in het echt". 'In het echt' betekent dan: op een andere computer en sneller. De programmeur kan nu zelf de programmatekst inspecteren en met zekerheid vaststellen dat er gesorteerd wordt. Maar hij stelt nog meer vast, bijvoorbeeld dat het sorteerprogramma stabiel is. Dat wil zeggen dat van de vele Jansen's die er zijn, de onderlinge volgorde niet gewijzigd wordt. Als de Jansen's bijvoorbeeld al gesorteerd zijn op voornaam, dan zal het prototype, zijnde bubble-sort, dat niet verstoren.
Nu heeft de programmeur een probleem: was dat stabiel zijn nu onderdeel van de specificatie, of was het slechts toeval. Als het onderdeel van de specificatie is, dan is een snel sorteerprogramma als quicksort uitgesloten, anders niet. Was de vaststelling dat er gesorteerd moest worden eigenlijk wel zeker, of was dat ook toeval?
Misschien is het voorbeeld wat gezocht, maar zodra we te maken hebben met interactieve programma's is het probleem echt serieus. Ik concludeer dat een programma weliswaar zichzelf specificeert, maar dat het niet de uitdrukkingskracht heeft van een logische bewering, die oneindig veel implementaties toe laat. Prototypering is wel zinvol, maar meestal is het geen sluitende specificatietechniek.

Het idee dat het mogelijk is om onder deze gedachtengang uit te komen, wordt vaak naar voren gebracht onder de kreet 'executeerbare specificaties'. Het is een mooie combinatie van woorden, zoiets als 'gratis geld', maar of het echt werkt behoeft een nauwkeuriger analyse. Zo een executeerbare specificatie is uitvoerbaar, dat wil zeggen het is een samengestelde opdracht aan een automaat. Het is dus gewoon een programma, alleen wellicht in een mooie taal van hoog niveau. Misschien is de executeerbare specificatie ook te duiden als een logische bewering, waar dus oneindig

veel programma's aan voldoen, maar dat is dan toch een andere duiding (een andere semantiek) dan die als een opdracht aan een automaat.
Ik concludeer dat een enkele zin best wel tegelijk een programma en een echte specificatie kan zijn, maar dan toch onder twee verschillende semantieken. Ik meen deze situatie te herkennen rondom functionele talen, die enerzijds geacht worden echte wiskundige functies te betekenen, anderzijds efficiënte programma's, soms zelfs reactieve programma's.
Ik herken het ook bij PROTOCOLD en bij Prolog.
Begrijp mij goed, ik ben voorstander van programmeertalen van hoog niveau, maar we moeten goed weten wat we bedoelen als we het hebben over 'executeerbare specificaties'.

Voor alle duidelijkheid: specificeren is dus iets anders dan programmeren. Specificeren is nodig voor het beheerst omgaan met de afspraken over wat componenten (deelprogramma's) doen.
Specificeren is nodig in verband met interoperabiliteitsproblemen. Dat zijn problemen van een eigenaardig soort. Als twee deelprogramma's niet goed samenwerken, is er meestal geen echt fundamenteel probleem; ze zouden best kunnen samenwerken, als het ene programma de parameters maar op de goede plaats had neergezet voor het andere, en als niet enkele bitjes per ongeluk verkeerd waren geïnterpreteerd. Natuurlijk kan je zeggen: „stom ja, zet die bitjes dan ook gewoon goed". Maar er zijn zoveel van die bitjes.
Er zijn natuurlijk nog programma's van enkele duizenden bits, maar de programmacode van veel embedded software beloopt al gauw vele miljoenen bits. Als ik een CD beschouw, een veelgebruikte drager van software, dan gaan op een schijfje ongeveer vijf miljard bits. Daarin kun je niet zomaar op zoek gaan naar enkele bits die verkeerd staan. Als een boek 50 miljoen bits bevat, dan is het schijfje te vergelijken met een berg van honderd boeken. Stelt U zich een berg van honderd boeken voor, waarvan misschien sommige ook met losse bladzijden en ga dan eens op zoek naar een woordje dat verkeerd staat.

## Kansen in de toekomst

Nederland heeft een sterke positie in onderwijs en onderzoek naar formele methoden en ruimer, in onderzoek en onderwijs op het gebied van de kern-informatica. Een aantal voorbeelden van sterk onderzoek heb ik al genoemd, maar er zijn er nog veel meer die ik hier tot mijn spijt niet allemaal kan noemen.
De benodigde investeringen in de software-industrie zijn aanzienlijk. Hoewel een prototype soms vlug

gemaakt is, blijkt telkens weer dat het goed ontwikkelen en onderhouden van een software-produkt veel tijd kost, en zoals meestal gaan ook hier de kosten voor de baten uit.
Er is veel werk te doen. Het is namelijk niet zo dat na invoering van formele specificaties alles vanzelf gaat. Dat is slechts een hulp-techniek om goed met interoperabiliteitsproblemen om te kunnen gaan.
De efficiëntieproblemen blijven aandacht vragen, maar tegelijk dienen nieuwe technologieën zich aan: neurale netten, spraaksynthese en -herkenning, uitbreiding van de gebruikelijke randapparaten van computers met geluid- en videoweergevers, netwerken met hoge capaciteit enzovoort.
Dat zijn allemaal kansen, in de positieve zin van het woord.
Ik zie de formele methoden hierbij als dijken. Ze zijn onmisbaar bij het veroveren van nieuwe gebieden. Ze zijn geen substituut voor de pompen, maar ze zijn wel nodig om een ordelijke toestand te creëren en te handhaven.

## Dankwoord

Tenslotte wil ik enkele woorden van dank uitspreken.
Het idee om dijken en formele methoden op één lijn te plaatsen is afkomstig van J. Kroon van KPN-research.
Graag wil ik de collega's en groepsleiders bij Philips bedanken met wie ik de afgelopen jaren heb kunnen samenwerken, collega's in de produktdivisies en in het Natuurkundig Laboratorium.
Ik kan hier niet alle namen noemen, maar één uitzondering wil ik toch maken:
mijn collega Jonkers van het Natuurkundig Laboratorium, met wie ik vele jaren heb mogen samenwerken en van wie ik bijzonder veel geleerd heb.
Graag ook bedank ik de directie bij Philips, met name Valster, Bosma, Nijman en Bourgonjon, die mij steunden en ruimte gaven om te werken aan mijn wetenschappelijke ambities.

Mijn ouders en mijn vrouw Liesbeth wil ik graag bedanken voor hun hulp en hun steun.

Bijzondere dank gaat ook naar professor Kruseman Aretz, mijn eerste promotor, voor zijn steun en vriendelijke begeleiding bij mijn promotiewerk en ook daarna.
En naar professor Bergstra, mijn tweede promotor, aan wie ik veel verschuldigd ben en in wie ik een groot vertrouwen heb.
Graag ook wil ik hen bedanken die zich hebben ingezet voor mijn benoeming; ik zal ervoor ijveren het in mij gestelde vertrouwen niet te beschamen.
Tenslotte dank ik de andere hoogleraren van de vakgroep informatica, Aarts, Backhouse, Baeten, Hammer, Van Hee, Hilbers,

Paredaens en Rem, die mij vriendelijk in hun midden hebben opgenomen.

Ik dank U voor Uw aandacht.

# Literatuur


[1] C.A.R. Hoare.
An axiomatic approach for computer programming,
Communications of the ACM 12, blz. 576-580,583 (1969).

[2] E.W. Dijkstra.
Guarded commands, nondeterminacy and the formal derivation of programs,
Communications of the ACM 18, blz. 453-457 (1975).

[3] D. Bjørner, C.B. Jones.
The Vienna Development Method: the Meta-Language,
Lecture notes in computer science 61, Springer Verlag (1978).

[4] S. Mauw, G.J. Veltink.
Algebraic specification of communication protocols,
Cambridge University Press (1993).

[5] L.M.G. Feijs, H.B.M. Jonkers, C.P.J. Koymans, G.R. Renardel de Lavalette.
Formal Definition of the Design Language COLD-K,
Technical Report, ESPRIT project 432 (1987).

[6] L.M.G. Feijs, H.B.M. Jonkers, C.A. Middelburg.
Notations for software design,
Springer Verlag, FACIT serie (1994).

[7] H.B.M. Jonkers.
An overview of the SPRINT method,
in: J.C.P. Woodcock, P.G. Larsen (Eds.),
Industrial Strength Formal Methods,
Lecture notes in computer science 670, Springer Verlag, blz. 403-427 (1993).

[8] A. Joch.
How software doesn't work,
Byte 20-12, blz. 48-60 (1995).

[9] J. Browning.
Making free software pay
— the internet creates an alternative economics of innovation.
Scientific American 274-1, blz. 21 (1996).

[10] R. Milner.
A calculus of communicating systems,
Lecture notes in computer science 92, Springer Verlag, (1980).

[11] J.A. Bergstra, J.W. Klop.
Process algebra for synchronous communication,
Information and control 60, blz. 109-137, (1984).

[12] J.C.M. Baeten, W.P. Weijland.
Process algebra,
Cambridge University Press (1990).

[13] E. Brinksma.
On the design of extended LOTOS - a specification language for open distributed systems,
proefschrift, Technische Universiteit Twente (1988).

[14] S. Mauw, M.A. Reniers.
An algebraic semantics of basic message sequence charts,
The Computer Journal 37(4), pp. 269-277 (1994).

Loe Feijs is geboren in 1954 te Sittard en doorliep de lagere school te Stein en het Gymnasium te Geleen. Hij studeerde Elektrotechniek aan de TUE (toen nog TH geheten) waar hij afstudeerde in de informatie- en communicatietheorie. In 1979 werkte hij als gastonderzoeker bij CSELT in Turijn aan experimentele videocompressie en -codering.
Na vervulling van de dienstplicht trad hij in 1981 in dienst bij Philips Telecommunicatie Industrie (later APT) te Hilversum en werkte mee aan de ontwikkeling van computerbesturingen voor telefooncentrales.
Vanaf 1984 is Feijs verbonden aan het Philips Natuurkundig Laboratorium, waar hij onderzoek verricht aan onder andere formele specificatietechnieken, software architectuur en communicatieprotocollen. In 1990 promoveerde hij aan de TUE op een proefschrift betreffende specificatietechnieken en lambda calculus. Hij heeft bijdragen geleverd aan de taal COLD en meegewerkt aan een aantal industriële toepassingen van formele specificaties bij Philips. Van 1990 tot 1994 was hij clusterleider bij de groep 'Specificatie, Ontwerp en Realisatie' en sinds 1995 projectleider bij de groep 'Software-Engineering Methodologie' van de sector 'Informatie- en Softwaretechnologie (IST)'.
Vanaf september 1994 is hij aan de TUE verbonden als deeltijd hoogleraar op het gebied 'Industriële Toepassingen van Formele Methoden' bij de sectie Formele Methoden binnen de Faculteit Wiskunde en Informatica.

Vormgeving en druk:
Reproduktie en Fotografie van de CTD
Technische Universiteit Eindhoven

Informatie:
Academische en Protocollaire Zaken
Telefoon (040-247)2250/4676

ISBN 90 386 0068 2