Programmatuur zonder Logica

Rede

uitgesproken bij de aanvaarding van het ambt
van hoogleraar in de Toegepaste Logica
aan de Faculteit der Wijsbegeerte
van de Rijksuniversiteit te Utrecht
op woensdag 10 maart 1993

door

dr.ir. C.A. Middelburg

Utrecht
1993

ISBN 90-393-0325-8 (ISSN 0927-3395)

Geachte Rector, Dames en Heren,

Ik wil het met u hebben over toegepaste logica en programmatuurkunde. Binnen de vakgroep Theoretische Filosofie van de Universiteit Utrecht zal ik mij vooral bezighouden met onderzoek op het gebied van de programmatuurkunde waarbij de logica een vitale rol speelt. Overeenkomstig de bedoeling van een oratie zal ik u een indruk geven van mijn opvattingen en ideeën op dit terrein. Ik zal van deze gelegenheid gebruik maken om ook een aantal opmerkingen te maken over datgene waardoor ik mij in de eerste plaats laat motiveren bij mijn onderzoek.

Als programmatuurkunde noch toegepaste logica uw belangstelling heeft dan vraagt u zich misschien af waarom u eigenlijk naar mij bent komen luisteren. Het lijkt immers allemaal zo ver verwijderd van de kleine zaken die ons allemaal dagelijks bezighouden, zoals familie, vrienden, gezondheid en geld, en de grote wereldproblemen, zoals oorlog, honger en milieu. Maar wellicht blijkt in de loop van deze voordracht wel dat iets ervan u toch ook op één of andere wijze aangaat. Hoewel sommige details misschien alleen door vakgenoten goed zijn te volgen, zal ik proberen het voor iedereen enigszins begrijpelijk te maken.

De term programmatuurkunde verdient misschien enige toelichting. Programmatuurkunde is het vakgebied dat zich bezighoudt met de ontwikkeling van produkten waarbij de wetenschappelijke inzichten die er bestaan met betrekking tot programmatuur essentieel zijn. De beste Engelse term hiervoor is natuurlijk "sofware engineering", maar die term gebruik ik hier liever niet. Het wordt in het algemeen gebruikt waar "software management" meer op zijn plaats zou zijn. Het is zelfs zo dat een aanzienlijk deel van de Nederlandse "software engineers" het vakgebied programmatuurkunde als theoretische informatica beschouwt. Tijdens mijn voordracht zal ik onder meer proberen duidelijk te maken waarom de logica van belang is voor de programmatuurkunde.

Het maken van programmatuur

In grote lijnen ligt de manier waarop programmatuur wordt gemaakt voor de hand. Eerst wordt uitgezocht wat er gemaakt moet worden, daarna wordt een ontwerp gemaakt (mogelijk in een aantal stadia waarin steeds meer ontwerpbeslissingen worden verwerkt) en vervolgens wordt het ontwerp geprogrammeerd. Een centrale vraag in de programmatuurkunde is: "Hoe kan hierbij de juiste werking van het programmatuurprodukt worden gegarandeerd?". De vakterm is hier programma-correctheid. Dit is een term die allerlei filosofische discussies heeft opgeroepen waarop ik nu niet nader zal ingaan. Wel wil ik er alvast op wijzen dat het moeilijk is vast te stellen of een programmatuurprodukt juist werkt zonder nauwkeurige beschrijvingen van wat ervan wordt verwacht en hoe het ontwerp tot stand is gekomen.

De voorbeelden die worden gebruikt om het belang van programma-correctheid aan te tonen zijn meestal spectaculair. In de betreffende voorbeelden wordt altijd een scenario geschetst waarin een fout in een kritisch besturingsprogramma leidt tot grote rampen. Ik zal echter een minder spectaculair voorbeeld geven.

Als een geldautomaat wordt gebruikt om geld op te nemen van een giro- of bankrekening zijn er verscheidene programmatuurcomponenten betrokken bij de totstandkoming van de bijbehorende afschrijving. Een kleine tekortkoming van één van die componenten kan leiden tot een onjuiste afschrijving zoals een afschrijving van de verkeerde rekening. Dergelijke tekortkomingen zijn meestal een gevolg van het feit dat er tijdens het maken van de programmatuur bepaalde details over het hoofd zijn gezien. De regels betreffende het gebruik van geldautomaten die de banken hebben opgesteld zijn eigenlijk alleen acceptabel voor de gebruiker als de garantie kan worden gegeven dat in alle omstandigheden de betrokken programmatuurcomponenten correct zullen werken. De vraag is echter of de makers van de

betreffende programmatuur in staat zijn dit aan te tonen.

Vergelijkbare situaties doen zich vaker voor. Ik had bijvoorbeeld ook de totstandkoming van de telefoonrekening als voorbeeld kunnen nemen. Ik denk dat u nu beseft dat de deugdelijkheid van programmatuurprodukten ook voor u van enig belang is. Ik ben van mening dat veel programmatuurprodukten op dit punt onnodig te kort schieten.

Het toepassen van wetenschappelijke kennis

De theoretische informatica heeft op het gebied van de grondslagen van de programmatuurbouw voldoende wetenschappelijke basis ontwikkeld voor een aantal praktisch toepasbare methoden voor programmatuurontwikkeling met een degelijke wiskundige onderbouwing. De belangrijkste aspecten van dergelijke methoden, ook wel formele methoden genoemd, zijn formele specificatie en geverifieerd ontwerp.

Formele specificaties zijn beschrijvingen van programmatuurprodukten in een notatie die aan de wiskunde is ontleend. Op die manier kunnen abstracties van te ontwikkelen produkten formeel worden gemaakt. Een dergelijke notatie is meestal een wiskundige notatie die is uitgebreid om specificatie van programmatuur te vereenvoudigen. Aan de uitbreidingen wordt een nauwkeurig gedefinieerde betekenis toegekend in termen van algemeen aanvaarde begrippen uit de wiskunde. Verschillende interpretaties van een specificatie zijn hiermee uitgesloten. Belangrijk is verder dat bij gebruik van een dergelijke notatie wiskundig kan worden geredeneerd over specificaties.

Formele methoden gaan er van uit dat er eerst een formele specificatie kan worden gemaakt van wat er wordt verwacht van de programmatuur en dat daarna het ontwerp kan worden opgedeeld in stappen waarvan de rechtvaardiging te overzien is. Bij geverifieerd ontwerp zijn er een aantal formele bewijzen te leveren om een ontwerpstap te rechtvaardigen. Men spreekt hier

in het algemeen over het vervullen van de bewijsverplichtingen die een ontwerpstap met zich mee brengt.

Computerondersteuning is zeer gewenst bij het vinden van de bewijzen die nodig zijn voor het vervullen van de bewijsverplichtingen. De betreffende bewijzen hebben veelal weinig weg van bewijzen zoals men die vaak in de wiskunde tegenkomt. De bewijzen bij programmatuurontwikkeling zijn hoofdzakelijk bedoeld om op een nauwkeurige manier na te gaan of er geen details in een ontwerp over het hoofd zijn gezien, maar zij zijn meestal lang en nogal saai. Juist daarom is computerondersteuning daarbij van groot belang.

De huidige formele methoden bieden samenhangende technieken voor formele specificatie en geverifieerd ontwerp. Misschien ten overvloede wil ik opmerken dat zij die aspecten van programmatuurontwikkeling niet dekken waarvan de wiskundige onderbouwing momenteel nog onvoldoende is ontwikkeld. Om dezelfde reden zijn zij ook niet toepasbaar op alle soorten programmatuur. Dit betekent echter niet dat we formele methoden niet moeten gebruiken waar dat wel mogelijk is. Zij maken het mogelijk dat met details van specificatie en ontwerp van vele programmatuurprodukten wordt omgegaan op een wiskundige manier. Dat vergroot de kans om tot deugdelijke produkten te komen.

Andere methoden doen geen alternatieven aan de hand om te ontwikkelen programmatuurprodukten voldoende nauwkeurig en ondubbelzinnig te specificeren en om ze daarna systematisch geheel overeenkomstig de gemaakte specificaties te ontwikkelen. Gebrek aan nauwkeurigheid en systematiek leidt tot programmatuur zonder logica. Dat vergroot de kans op tekortkomingen.

Het verband met logica

Bij gebruik van een notatie die aan de wiskunde is ontleend kunnen niet alleen relevante abstracties van te ontwikkelen pro-

grammatuurprodukten formeel worden gemaakt maar kunnen bovendien claims betreffende die abstracties – zoals de eerdergenoemde bewijsverplichtingen – worden geformuleerd op een wiskundig nauwkeurige manier en in principe formele bewijzen worden geconstrueerd om die claims te rechtvaardigen. Het laatste vereist wel dat er bijbehorende formele redeneerregels zijn. De notatie tezamen met de redeneerregels noem ik een specificatieformalisme.

Een specificatieformalisme wordt dus gebruikt om (1) abstracties van programmatuurprodukten formeel te maken en (2) bewijzen van claims betreffende die abstracties te construeren. In de logica zijn grofweg dezelfde punten aan de orde. Logica wordt gebruikt om (1) wiskundige structuren formeel te maken en (2) bewijzen van eigenschappen van die structuren te construeren. Het verschil ligt hoofdzakelijk daarin dat een specificatieformalisme zich concentreert op wiskundige structuren die als relevante abstracties van programmatuurprodukten worden beschouwd.

De punten die bij een specificatieformalisme aan de orde komen zijn uitgebreid bestudeerd in de logica. Daarom ben ik van mening dat het ontwerp van een specificatieformalisme beschouwd kan worden als toegepaste logica. Een soortgelijk standpunt wordt ingenomen door Jos Baeten (1992) in zijn oratie in Eindhoven. Bij deze opvatting past het idee dat ik nu in grote lijnen zal schetsen.

De semantiek die voor de meeste specificatietalen wordt gegeven, beschrijft de betekenis van specificaties in termen van modellen die voldoen aan de specificaties. De betreffende modellen zijn gebaseerd op bekende begrippen uit de wiskunde zoals verzamelingen, functies en relaties. Het is de vraag of een dergelijke semantiek het meest geschikt is voor specificatietalen.

Zoals ik eerder heb geschetst, is een belangrijk aspect van het gebruik van een specificatietaal dat formele bewijzen kunnen worden geconstrueerd om claims betreffende beschreven abstracties van programmatuurprodukten te rechtvaardigen. Het

ligt daarom voor de hand om als betekenis van een specificatie een verzameling van logische uitdrukkingen te nemen die de beschreven abstractie volledig karakteriseert. Dit leidt tot een semantiek die de uitdrukkingen van de specificatietaal vertaalt in logische uitdrukkingen.

Mijn ervaringen met de VDM-notatie, zoals die wordt gebruikt door Jones (1990), zijn dat een dergelijke semantiek betrekkelijk eenvoudig is te geven (zie Middelburg, 1993). De gebruikte logica was in dit geval de klassieke predikatenlogica met gelijkheid. Om recursieve definities aan te kunnen, werden aftelbaar oneindige disjuncties toegestaan. Het gemak waarmee de bijbehorende redeneerregels op basis van deze semantiek kunnen worden gerechtvaardigd in de klassieke predikatenlogica is verrassend.

In overeenstemming hiermee houd ik een specificatieformalisme voor een logica in vermomming. Het gebruik ervan verkleint de kans op programmatuur zonder logica.

Voordat ik u een uitgebreidere indruk geef van mijn opvattingen over het gebruik van formele methoden voor programmatuurontwikkeling, zal ik eerst dieper ingaan op twee strijdpunten onder sommigen van mijn vakgenoten. Het eerste strijdpunt betreft twee verschillende manieren van specificeren, namelijk model-georiënteerd specificeren en eigenschap-georiënteerd specificeren. Het tweede strijdpunt betreft het gebruik van klassieke tweewaardige logica's of niet-klassieke driewaardige logica's. Dit gedeelte is waarschijnlijk minder interessant voor niet-vakgenoten.

Model- of eigenschap-georiënteerd specificeren

Vaak wordt er een onderscheid gemaakt tussen model-georiënteerde specificatie en eigenschap-georiënteerde specificatie. In een model-georiënteerde specificatie wordt beschreven wat er van een programmatuurprodukt wordt verwacht in termen van

algemene abstracte wiskundige begrippen – zoals verzamelingen en functies – en wiskundig onderbouwde abstracties van begrippen die worden gehanteerd bij de bouw van programmatuur – zoals toestanden en operaties die een toestand raadplegen en/of wijzigen. Voorbeelden van dergelijke specificatietalen zijn de VDM-notatie (zie Jones, 1990 of Jones en Shaw, 1990) en Z (zie Spivey, 1988, 1989).

Zo beschrijft men in de VDM-notatie wat er wordt verwacht van een te ontwikkelen produkt in termen van de operaties die het moet kunnen uitvoeren. Operaties kunnen resultaten opleveren die afhangen van een toestand en die toestand veranderen. Operaties komen in het algemeen overeen met deelprogramma's (zoals subroutines, procedures, etc.) van het uiteindelijke produkt. Met een pre-conditie worden de omstandigheden waaronder het produkt een operatie met succes moet uitvoeren begrensd en met een post-conditie worden de mogelijke effecten van de uitvoering afgebakend. Pre- en post-condities zijn logische uitdrukkingen. De soorten objecten die worden gemanipuleerd door de operaties, worden beschreven in termen van natuurlijke, gehele en rationale getallen, eindige verzamelingen, afbeeldingen en rijen, etcetera.

Ik heb reeds vermeld dat het betrekkelijk eenvoudig is om een semantiek voor een dergelijke model-georiënteerde specificatietaal te geven die de betekenis van een specificatie in de taal beschrijft als een verzameling van logische uitdrukkingen die het beschrevene volledig karakteriseert.

In een eigenschap-georiënteerde specificatie wordt een programmatuurprodukt beschreven in termen van zijn gewenste eigenschappen. Deze eigenschappen moeten in het algemeen worden gegeven in de vorm van (conditionele) vergelijkingen. Men spreekt dan ook van algebraïsche specificatietalen. Voorbeelden van dergelijke specificatietalen zijn Clear (zie Burstall en Goguen, 1981), ACT ONE (zie Ehrig en Mahr, 1985), de Larch Shared Language (zie Guttag en Horning, 1986) en ASF (zie Bergstra, Heering en Klint, 1989).

In een algebraïsche specificatie worden de toestanden niet expliciet beschreven. De toestanden worden impliciet beschreven door vergelijkingen die de operaties met elkaar in verband brengen.

Vergelijkingen zijn in wezen eenvoudige logische uitdrukkingen. Dit betekent dat een algebraïsche specificatie eigenlijk al een karakteriserende verzameling van logische uitdrukkingen is. Het is daarom triviaal om een semantiek voor een algebraïsche specificatietaal te geven die de betekenis van een specificatie in de taal beschrijft als een karakteriserende verzameling logische uitdrukkingen.

Algebraïsche specificatie van programmatuur is lastig. Dit is niet verwonderlijk als men bedenkt dat er wordt uitgegaan van een zeer elementaire notatie die op geen enkele manier is aangepast om de specificatie van programmatuur te vereenvoudigen. Wel kunnen betrekkelijk eenvoudig algebraïsche specificaties worden gegeven van de soorten objecten die in bijvoorbeeld de VDM-notatie worden gebruikt om de toestanden van een systeem te modelleren, met andere woorden van de data types. Dit gaat ook op voor andere soorten objecten die nuttig of nodig zijn voor één of ander soort toepassingen. Zulke algebraïsche specificaties leiden in het algemeen tot vereenvoudiging van de formele bewijzen die nodig zijn voor het vervullen van bewijsverplichtingen.

Model-georiënteerde specificatie en eigenschap-georiënteerde specificatie kunnen elkaar dus goed aanvullen. Algebraïsche specificatietechnieken lijken daarom het best tot hun recht te komen in een specificatietaal waarin algebraïsche specificatie van soorten objecten met model-georiënteerde specificatie van toestandsgebaseerde systemen kan worden gecombineerd. Uit het voorafgaande mag duidelijk zijn dat een dergelijke taal semantisch geen fundamentele problemen oplevert.

Zo'n taal is bijvoorbeeld COLD-K (zie Jonkers, 1989 of Feijs en Jonkers, 1992). Deze taal kan worden beschouwd als een algebraïsche specificatietaal die is uitgebreid met speciale voor-

zieningen voor het beschrijven van toestandsgebaseerde systemen. De semantiek die er voor wordt gegeven weerspiegelt dit gezichtspunt heel duidelijk. Het is waarschijnlijk de eerste semantiek die de betekenis van een model-georiënteerde specificatie beschrijft als een verzameling van logische uitdrukkingen die het beschrevene volledig karakteriseert. COLD-K en een formele methode voor programmatuurontwikkeling rondom deze specificatietaal werden ontwikkeld op het Philips Research Laboratorium in Eindhoven.

De zojuist geschetste visie op model-georiënteerde specificatie en eigenschap-georiënteerde specificatie is niet origineel. Het punt is dat vanuit de eerder geschetste benadering van de semantiek van specificatietalen het idee van de combinatie zich eigenlijk direct aan je opdringt. Bovendien is deze visie nog altijd het vermelden waard omdat de discussie over welke van deze twee stijlen beter is steeds weer opduikt.

Klassieke logica versus niet-klassieke logica

Zowel in model-georiënteerde specificaties als in eigenschap-georiënteerde specificaties worden functies gespecificeerd. Deze functies zijn in het algemeen partiële functies, met andere woorden functies die niet altijd een resultaat opleveren. Partiële functies geven aanleiding tot ongedefinieerde uitdrukkingen, dit wil zeggen uitdrukkingen die geen objecten van het bedoelde soort aanduiden. Dit maakt het redeneren over partiële functies problematisch in de klassieke predicatenlogica. Er zijn verschillende benaderingen om dit probleem op te lossen (zie ook Cheng en Jones, 1990 en Middelburg en Renardel de Lavalette, 1991).

Eén benadering blijft binnen het gebied van klassieke tweewaardige logica's: atomaire formules waarin ongedefinieerde uitdrukkingen voorkomen zijn logisch onwaar. Op die manier behoeft de aanname van de uitgesloten derde niet te worden opgegeven. Met andere woorden, als een formule niet kan worden

geclassificeerd als waar, wordt het onverbiddelijk geclassificeerd als onwaar. Een verder onderscheid wordt niet gemaakt. Deze benadering wordt toegewezen aan Scott (1967). Het is ook gevolgd in, bijvoorbeeld, MPL_ω door Koymans en Renardel de Lavalette (1989).

Een andere benadering, die niet binnen het gebied van klassieke tweewaardige logica's blijft, is gevolgd in de logica LPF: atomaire formules waarin ongedefinieerde uitdrukkingen voorkomen kunnen logisch waar-noch-onwaar zijn. Daarmee wordt de aanname van de uitgesloten derde opgegeven. Toch worden de klassieke voorwaarden voor waarheid en onwaarheid van formules overgenomen. De betreffende formule wordt geclassificeerd als waar-noch-onwaar dan en slechts dan als het niet kan worden geclassificeerd als waar of onwaar aan de hand van deze voorwaarden. Vele andere driewaardige logica's voor partiële functies blijken nauw verwant te zijn aan LPF (zie Konikowska, Tarlecki en Blikle, 1988 of Gavilanes-Franco en Lucio-Carrasco, 1990).

Barringer, Cheng en Jones (1984) behandelen een ongetypeerde versie van LPF. Een getypeerde versie van LPF wordt gebruikt als de basis voor formele specificatie en geverifieerd ontwerp in VDM. Om bruikbaar te zijn bij programmatuurontwikkeling is deze versie uitgebreid voor de types die worden gebruikt in VDM, recursieve functie definities, etc. Het uiteindelijke resultaat is eigenlijk de VDM-notatie tezamen met de bijbehorende redeneerregels.

Cliff Jones en ik hebben onlangs het manuscript van een artikel voltooid waarin naast de gebruikelijke model-theoretische rechtvaardiging van de redeneerregels van deze getypeerde versie een logische rechtvaardiging wordt gegeven door middel van een vertaling naar klassieke predicatenlogica die zodanig is dat wat kan worden bewezen hetzelfde blijft na vertaling. Hiermee tonen we aan hoe deze niet-klassieke logica klassiek kan worden gereconstrueerd. Klassieke logica wordt dus meta-logisch gebruikt. Het levert een klassieke uitleg van LPF die verhel-

derend is voor al die mensen die deze logica gebruiken maar eigenlijk klassiek denken. Deze aanpak vertoont grote overeenkomsten met de eerder geschetste benadering van de semantiek van specificatietalen.

De vertaling van LPF naar klassieke predicatenlogica werpt natuurlijk ook de vraag op wat eigenlijk het bestaansrecht is van LPF en verwante driewaardige logica's. De vertaling suggereert dat het bestaansrecht in de eerste plaats moet worden gezocht in de mogelijkheden om eigenschappen betreffende partiële functies korter te kunnen formuleren en de mogelijkheden om kortere bewijzen van dergelijke eigenschappen te kunnen geven. In die mogelijkheden ligt voor mij het bestaansrecht van LPF.

Hiermee wil ik de behandeling van enkele specialistische onderwerpen uit mijn vakgebied afsluiten. Ik ga nu over op mijn opvattingen over de wiskundige benadering van formele methoden voor de praktijk van programmatuurontwikkeling.

De praktijk en ik

Hoewel het misschien niet erg gebruikelijk is tijdens een oratie, lijkt het mij nuttig om nu eerst enkele opmerkingen over mijzelf te maken. Mijn hoofdfunctie ligt niet bij de Universiteit Utrecht maar bij PTT Research, waar ik al sinds eind 1971 werk. Gedurende mijn loopbaan bij PTT Research heb ik niet alleen onderzoek op informaticagebied gedaan.

Aanvankelijk heb ik vooral systeemprogrammatuur ontwikkeld. Het begon met een TRAC "macro processor" en een "runtime system" voor de programmeertaal BCPL. Daarna heb ik onder meer gewerkt aan de ontwikkeling van een experimenteel "data base management system" en aan de ontwikkeling van een "compiler" voor de programmeertaal CHILL. Mijn belangrijkste bijdrage aan de ontwikkeling van die compiler is het ontwerp van de vertaalfase met behulp van VDM geweest (zie Middelburg, 1980). Ook heb ik regelmatig automatiseringsprojecten binnen het PTT-bedrijf geadviseerd over het toepassen van re-

sultaten van recente ontwikkelingen op informatica-gebied en ook sommige projecten daarbij begeleid.

Het is mijn overtuiging dat de wiskundige benadering van formele methoden voor programmatuurontwikkeling in vele gevallen noodzakelijk is om op een technisch verantwoorde manier programmatuur te maken. Deze overtuiging komt voor een groot deel voort uit mijn eigen ervaringen in de praktijk. Misschien zijn er voorstanders die vanwege hun gebrek aan praktische ervaring in de ontwikkeling van programmatuur beschuldigd kunnen worden van wereldvreemdheid, maar dat lijkt in mijn geval toch niet terecht. Ik ga ook niet de tegenstanders van de wiskundige benadering van formele methoden beschuldigen van wereldvreemdheid. Wel wil ik mijn standpunten in deze duidelijk naar voren brengen.

Een formele methode is geen wondermiddel

In formele methoden voor de ontwikkeling van programmatuur staat het maken van wiskundige beschrijvingen centraal. Sommige argumenten die voor en vooral tegen het gebruik van formele methoden worden aangevoerd komen voort uit de verkeerde opvatting dat het doel van het maken van een wiskundige beschrijving het creëren van een exacte copie van de werkelijkheid is. Overigens wordt men tegenwoordig al op het VWO nadrukkelijk gewaarschuwd tegen deze opvatting.

Met een wiskundige beschrijving kan men in het algemeen de werkelijkheid slechts benaderen. Het betreft een abstractie van de werkelijkheid. Dit betekent dat een wiskundige beschrijving in zekere zin meestal onjuist is. Toch wordt het succes in technische disciplines in sterke mate bepaald door het gebruik van een wiskundige aanpak vergelijkbaar met de eerdergenoemde formele methoden voor de ontwikkeling van programmatuur.

Het punt is dat men er in de praktijk altijd weer in slaagt de werkelijkheid te benaderen wat betreft relevante punten. Dit

maakt een betrouwbare analyse van verscheidene aspecten van een produkt in verschillende ontwerpstadia mogelijk. Men kan zo vooraf niet met volledige zekerheid vaststellen of het uiteindelijke produkt aan de gestelde eisen zal voldoen, maar men kan wel zoveel zekerheid verkrijgen als de huidige stand van de wetenschap toelaat.

Zo zijn er ook aspecten van programmatuurprodukten waarvoor de theoretische informatica nog onvoldoende wiskundige onderbouwing heeft geleverd, zoals bijvoorbeeld de "real-time" aspecten. Voor de functionele aspecten van een grote klasse van programmatuurprodukten ligt de situatie echter anders. Door de wiskundige benadering die ik heb geschetst kan die wetenschappelijke kennis met vrucht worden toegepast om betere programmatuur te maken en complexere problemen op te lossen.

Terugkomend op de real-time aspecten van programmatuurprodukten wil ik nog vermelden dat er in Nederland veel belangwekkend onderzoek op dit gebied wordt gedaan. Zo hebben Groote (1990) en Baeten en Bergstra (1991) interessante bijdragen geleverd aan de wiskundige onderbouwing. Toetenel (1992) heeft in zijn proefschrift onderzocht of de bestaande wiskundige onderbouwing kan worden gebruikt om de VDM-notatie uit te breiden tot de basis van een praktisch toepasbare formele methode die ook de real-time aspecten dekt.

In eerste instantie zal men in de praktijk veelal moeten afzien van geverifieerd ontwerp en zich beperken tot formele specificatie. Dat is ook nuttig want formele specificatie dwingt een preciese stijl van formuleren af. Dat komt de helderheid van onderliggende concepten en het begrip van de aard van problemen ten goede.

Verificatie moet vaak nog op een te elementair niveau plaatsvinden en is daardoor voor omvangrijke programmatuurprodukten voorlopig niet praktisch haalbaar. Hier is "boot-strapping" nodig. Uit de kennis van applicatiegebieden zullen geleidelijk modellen en theoriën over de applicatiegebieden moeten worden opgebouwd om verificatie praktisch haalbaar te maken. Formele

specificatie van programmatuur speelt hierbij een centrale rol.

Voor de duidelijkheid wil ik nog opmerken dat programma's als zij geheel los worden gezien van hun omgeving – waartoe onder meer de apparatuur behoort waarop zij worden uitgevoerd – zich in principe wel lenen voor een volledige wiskundige beschrijving. Met andere woorden, zij moeten wiskundig volledig begrepen kunnen worden. Dat komt omdat programma's ook niet meer dan abstracties van de werkelijkheid kunnen zijn. Dit wordt volgens mij maar al te vaak vergeten. Men realiseert zich veelal pas na verrassingen met een voltooid programma dat het de werkelijkheid slechts benadert.

Een formele methode is ook niet alles

Dikwijls worden tegen het gebruik van formele methoden argumenten aangevoerd die niet ter zake doen. Hoewel dergelijke argumenten in principe aanvechtbaar zijn, wekken ze bij veel mensen foutieve voorstellingen en zijn ze moeilijk te weerleggen.

Zo wordt als argument om formele methoden niet te gebruiken vaak aangevoerd dat je bij opdrachtgevers niet hoeft aan te komen met formele specificaties van programmatuur. Dit argument is natuurlijk niet overtuigend. Het is niet realistisch om te verwachten dat opdrachtgevers overweg moeten kunnen met de notaties die de makers van programmatuur gebruiken. Het duidt op een onderschatting van het vakgebied of een schromelijke overschatting van de gemiddelde opdrachtgever. Vanzelfsprekend is een goede communicatie met opdrachtgevers essentieel en formele specificaties kunnen nuttig zijn in het stadium waarin wordt uitgezocht wat er gemaakt moet worden. Maar het is duidelijk dat formele specificaties niet bedoeld zijn om direct te communiceren met opdrachtgevers.

Als argument om formele methoden niet te gebruiken wordt ook nogal eens aangevoerd dat je er niet alle problemen die zich kunnen voordoen bij de ontwikkeling van programmatuur

mee kunt oplossen. Ook dit argument is zwak. Het is een feit dat sommige aspecten van programmatuurontwikkeling, zoals bijvoorbeeld prestatie-analyse en het waarborgen van hanteerbaarheid voor menselijke gebruikers, niet of nauwelijks worden ondersteunt door formele methoden. Dat komt in het algemeen omdat zij nog niet goed worden begrepen en er nog onvoldoende wiskundige onderbouwing voor is. Zoals elders zullen dergelijke situaties ook bij programmatuurbouw altijd blijven bestaan. Wie blijft wachten op een panacee is dan ook weinig reëel.

Het argument dat we met een formele specificatie de werkelijkheid slechts kunnen benaderen is reeds ter sprake gekomen. Dit argument wordt vaak gehanteerd door tegenstanders van formele methoden die mij er van proberen te overtuigen dat juist zij zich met de meest relevante aspecten van programmatuurontwikkeling bezighouden, dat zij die aspecten heel goed begrijpen, maar dat die aspecten waarschijnlijk niet formeel zijn te maken en dat dit ook geen praktisch nut heeft. Door vast te houden aan een dergelijk standpunt wordt natuurlijk weinig bijgedragen aan de vooruitgang van het vakgebied die nodig is om betere programmatuur te kunnen maken en complexere problemen te kunnen oplossen. Er wordt ook geheel aan voorbij gegaan dat het soms op zijn minst wenselijk is dat de juiste werking van een programmatuurprodukt kan worden aangetoond.

Vele tegenstanders van de wiskundige benadering van formele methoden blijken zichzelf software engineer te noemen. Dat is verwonderlijk omdat engineering staat voor het gebruik van wiskunde en wetenschap om op een technisch en economisch verantwoorde manier problemen op te lossen. Daarom meen ik nogeens uiting te moeten geven aan mijn opvatting dat de echte software engineer in zijn opleiding ervaring behoort te hebben opgedaan met het gebruik van tenminste één formele methode voor programmatuurontwikkeling.

Wetenschappelijk onderzoek en praktijk

Zoals reeds gezegd ligt mijn hoofdfunctie niet bij de Universiteit Utrecht maar bij PTT Research. Voordat ik mijn rede met enkele dankwoorden afsluit, wil ik nog enkele woorden wijden aan de plaats die PTT Research probeert in te nemen tussen de wetenschappelijke wereld en de werkmaatschappijen van Koninklijke PTT Nederland.

Bij PTT Research bestaat mijn taak – evenals die van velen van mijn collega's binnen de hoofdafdeling Informatica – onder meer uit het opsporen van nieuwe wetenschappelijke ontwikkelingen op het gebied van de informatica, het nader bestuderen van interessante ontwikkelingen en het schatten van de waarde ervan voor PTT en indien waardevol bevonden het bruikbaar maken voor de praktijk bij PTT. Dit vereist goede kontakten met de academische wereld.

Uit eigen ervaring weet ik dat dit heel moeilijk is als je niet als gelijkwaardig wordt beschouwd. Aan de meeste universiteiten in binnen- en buitenland wordt je slechts als gelijkwaardig beschouwd als je regelmatig publiceert in erkende wetenschappelijke tijdschriften. Heel lang was dit bij PTT Research niet mogelijk. Eén van de oorzaken was dat de aard van het onderzoek dat werd verricht zich daar vaak niet zo voor leende. Een andere oorzaak was dat er onvoldoende tijd voor werd ingeruimd omdat de inspanning die nodig is om een wetenschappelijk artikel te schrijven schromelijk werd onderschat.

Na vele magere jaren braken omstreeks 1985 zeven vette jaren aan. PTT Research onderkende dat een behoorlijk wetenschappelijk niveau noodzakelijk is voor goede wetenschappelijke kontakten en bovendien nuttig is voor het bevorderen van aanzien en goede naam bij de opdrachtgevers binnen PTT. Er werd niet alleen ruimte geschapen voor het schrijven van wetenschappelijke publicaties, maar in sommige gevallen zelfs voor het schrijven van een proefschrift. In hetzelfde kader past ook de belangstelling van PTT Research in deelname aan inter-

nationale onderzoeksprojecten waarin wordt samengewerkt met universiteiten en andere onderzoeksinstellingen.

Dit alles leidde echter tot nogal wat werk waarvan het directe belang voor PTT steeds moeilijker was aan te tonen. Dat is vanzelfsprekend niet lang vol te houden omdat de belangrijkste opdrachtgevers van PTT Research, te weten PTT Post en PTT Telecom, er op den duur natuurlijk weinig voor voelen om financieel bij te dragen aan onderzoek dat weinig of niets van doen heeft met hun korte termijn doelstellingen. Het einde van de vette jaren lijkt nabij.

Hopelijk kan ik vanuit mijn functie aan deze universiteit er toe bijdragen dat PTT Research zijn rol als intermediair behoorlijk kan blijven vervullen. Ik acht die rol ook van belang voor de wetenschappelijke wereld. Instellingen zoals PTT Research zijn nodig voor een sterke interactie tussen wetenschappelijk onderzoek en praktijk. Goede wetenschap en professionele ontwikkeling van nieuwe produkten zijn daarvan beide afhankelijk.

Besluit

Ik wil besluiten met het uitspreken van enkele dankwoorden. Ik dank het College van Bestuur van de Universiteit Utrecht voor mijn benoeming. Verder dank ik iedereen in de sectie Toegepaste Logica, in de vakgroep Theoretische Filosofie en in de faculteit Wijsbegeerte die zich voor mijn benoeming heeft ingezet. De directie van PTT Research ben ik erkentelijk voor het steunen van mijn benoeming door het beschikbaar stellen van een deel van mijn werktijd.

Ik wil ook iedereen bedanken waarmee ik de afgelopen twee decennia prettig heb kunnen samenwerken als collega bij PTT Research of anderszins. Jullie hebben allemaal op de één of andere manier bijgedragen aan mijn ontwikkeling als onderzoeker. Ik kan jullie onmogelijk allemaal met name noemen. Daarom noem ik alleen enkele mensen die volgens mij het meest van invloed zijn geweest op de wijze waarop ik tegen wetenschappelijk

onderzoek, programmatuurkunde en toegepaste logica aankijk.

Leendert Dekker, mijn begeleider tijdens het afstudeerwerk voor mijn universitaire studie natuurkunde, heeft mij er al heel vroeg op gewezen dat een wetenschappelijk onderzoeker zich moet inspannen om bekendheid te geven aan de resultaten van zijn werk. Dit is een evidentie binnen de academische wereld, maar zeker niet algemeen doorgedrongen in de industrie. Emile Nijenhuis, met wie ik in het begin van de zeventiger jaren onderzoek deed op het gebied van databases, heeft mij destijds laten zien dat een wiskundige benadering het mogelijk maakt om betere programmatuur te maken en complexere problemen op te lossen in programmatuur. Rudolf Meijer, met wie ik aan het eind van de zeventiger jaren werkte aan het ontwerp en de beschrijving van de programmeertaal CHILL alsmede de bouw van een vertaler voor deze taal, heeft mij er toe aangezet om mijn uitspraken over typering in CHILL te onderbouwen met formele bewijzen. Ik wil deze gelegenheid gebruiken om jullie hiervoor te bedanken.

Verder wil ik noemen Harry van Binsbergen en Jeroen Bruijning. Jullie hebben als mijn directe chef het onderzoek dat ik in de afgelopen twintig jaren bij PTT Research heb gedaan bestuurlijk ondersteund. Ik dank jullie voor de mij geboden kansen.

Vervolgens wil ik graag noemen Jan Bergstra. Voor velen met wie jij omgaat ben je van grote waarde. Dat geldt ook voor mij. Jij hebt er onder meer aan bijgedragen dat ik mijn carriere als onderzoeker niet heb opgegeven toen ik gedurende enige tijd ten onrechte dacht dat er verder niets meer van viel te verwachten.

Tenslotte wil ik mijn vrouw en kinderen bedanken voor hun grote tolerantie als ik eigenlijk teveel begrip en steun van ze verwacht wanneer ik mijn werk mee naar huis neem.

Ik heb gezegd.

Literatuur

Baeten, J.C.M. (1992) *Formele specificatie en wiskundige verificatie*. Intreerede, Technische Universiteit Eindhoven.

Baeten, J.C.M. en Bergstra, J.A. (1991) Real time process algebra. *Formal Aspects of Computing*, **3(2)**, 142–188.

Barringer, H., Cheng, J.H. en Jones, C.B. (1984) A logic covering undefinedness in program proofs. *Acta Informatica*, **21**, 251–269, 1984.

Bergstra, J.A., Heering, J. en Klint, P. (1989) *Algebraic Specification*, ACM Press Frontier Series, Addison-Wesley.

Burstall, R.M. en Goguen, J.A. (1981) An informal introduction to specifications using Clear, in *The Correctness Problem in Computer Science* (eds. Boyer, R. en Moore, J.), hoofdstuk 4, Academic Press.

Cheng, J.H. en Jones, C.B. (1990) On the usability of logics which handle partial functions. Technical Report UMCS-90-3-1, University of Manchester, Department of Computer Science.

Ehrig, H. en Mahr, B. (1985) *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*, EATCS Monograph, Springer-Verlag.

Feijs, L.M.G. en Jonkers, H.B.M. (1992) *Formal Specification and Design*, Cambridge Tracts in Theoretical Computer Science 35, Cambridge University Press.

Gavilanes-Franco, A. en Lucio-Carrasco, F. (1990) A first order logic for partial functions. *Theoretical Computer Science*, **74**, 37–69.

Groote, J.F. (1990) Specification and verification of real time systems in ACP, in *Protocol Specification, Testing, and Verification, X* (eds. Logrippo, L., Probert, R.L. en Ural, H.), North-Holland, pp. 261–274.

Guttag, J.V. en Horning, J.J. (1986) Report on the Larch shared language. *Science of Computer Programming*, **6**, 103–134.

Jones, C.B. (1990) *Systematic Software Development Using VDM*, 2nd edition, Prentice-Hall International Series in Computer Science, Prentice-Hall.

Jones, C.B. en Shaw, R.C.F. (1990) *Case Studies in Systematic Software Development*, Prentice-Hall International Series in Computer Science, Prentice-Hall.

Jonkers, H.B.M. (1989) An introduction to COLD-K, in *Algebraic Methods: Theory, Tools and Applications* (eds. Wirsing, M. en Bergstra, J.A.), LNCS 394, Springer-Verlag, pp. 139–205.

Konikowska, B., Tarlecki, A. en Blikle, A. (1988) A three-valued logic for software specification and validation, in *VDM '88* (eds. Bloomfield, R., Marshall, L. en Jones, R.), LNCS 328, Springer-Verlag, pp. 218–242.

Koymans, C.P.J. en Renardel de Lavalette, G.R. (1989) The logic MPL_ω, in *Algebraic Methods: Theory, Tools and Applications* (eds. Wirsing, M. en Bergstra, J.A.), LNCS 394, Springer-Verlag, pp. 247–282.

Middelburg, C.A. (1980) A formal definition based design of the translation phase of a CHILL compiler, in *Proceedings CHILL Implementors/Users Meeting*, 1980.

Middelburg, C.A. (1993) *Logic and Specification – Extending VDM-SL for Advanced Formal Specification*, Computer Science Series: Research and Practice 1, Chapman & Hall.

Middelburg, C.A. en Renardel de Lavalette, G.R. (1991) LPF and MPL_ω – a logical comparison of VDM SL and COLD-K, in *VDM '91*, Volume 1 (eds. Prehn, S. en Toetenel, W.J.), LNCS 551, Springer-Verlag, pp. 279–308.

Scott, D.S. (1967) Existence and description in formal logic, in

Bertrand Russell, Philosopher of the Century (ed. Schoenman, R.), Allen & Unwin, pp. 181–200.

Spivey, J.M. (1988) *Understanding Z*, Cambridge Tracts in Theoretical Computer Science 3, Cambridge University Press.

Spivey, J.M. (1989) *The Z Notation: A Reference Manual*, Prentice-Hall International Series in Computer Science, Prentice-Hall.

Toetenel, W.J. (1992) *Model Oriented Specification of Communicating Agents.* Proefschrift, Technische Universiteit Delft.