Specificatie van berekening

Hesselink, W.H.

Specificatie van Berekening

Dr. W. H. Hesselink

Rede uitgesproken bij de aanvaarding van het ambt
van bijzonder hoogleraar in de
Leer van de Programmacorrectheid
aan de Rijksuniversiteit Groningen
op 4 april 1995

Een citaat uit Hartmanis [Har93]:

> *One of the defining characteristics of computer science is the immense difference in scale of the phenomena computer science deals with. From the individual bits of programs and data in the computers to billions of operations per second to the highly complex operating systems and various languages in which the problems are described, the scale changes through many orders of magnitude.*
>
> *... The computer scientist has to create many levels of abstractions to deal with these problems. He has to create intellectual tools to conceive, design, control, program, and reason about the most complicated of human creations. Furthermore, this has to be done with unprecedented precision. The underlying hardware which executes the computations are universal machines and therefore they are chaotic systems: the slightest change in their instructions can result in arbitrary large differences in the results.*
>
> *... This is achieved by successive layers of implemented abstraction wrapped around the chaotic universal machines which help to bridge the many orders of magnitude in the scale of things.*

Tussen vierkante haakjes staan literatuurverwijzingen; men zie hiervoor de literatuurlijst aan het eind van deze brochure. Getallen tussen ronde haakjes verwijzen naar de aantekeningen die daarvoor staan.

Specificatie van Berekening

Mijnheer de Rector Magnificus,
Dames en Heren,

> *Ik draag deze rede op aan de nagedachtenis van Jan van de Snepscheut, mijn leermeester in de informatica, die de vakgroep Informatica in Groningen van 1984 tot 1989 met enthousiasme en visie geleid heeft, die vervolgens is vertrokken naar Californië, waar hij in februari 1994 op tragische wijze om het leven is gekomen.*

Zeergewaardeerde toehoorders,

De titel van deze voordracht is "Specificatie van berekening". Dit is namelijk het centrale begrip in de informatica. Specificatie is de beschrijving van wat een computerprogramma (maar het zou ook een hardware-component kunnen zijn) moet doen. Berekening is de uitvoering hiervan. Ik wil over de uitvoering kort zijn. Het programma is een berekeningsvoorschrift, d.w.z. een gestructureerde rij opdrachten. Deze rij wordt omgezet in een veel langere rij van voor de mens onleesbare opdrachten aan de machine. Deze laatste rij opdrachten wordt mechanisch of liever electronisch uitgevoerd door de processor in het inwendige van de computer. Wij laten als informatici de natuurkundige processen in de processor rustig over aan de natuurkundigen en de electrotechnici. De echte uitvoering van de berekening is dus niet ons terrein. Wij zijn verantwoordelijk voor de inhoud of de betekenis van de berekening. Dit komt erop neer, dat wij op allerlei niveaus opdrachten moeten uitwerken, d.w.z. opdelen in rijen van kleinere opdrachten. Op elk van deze niveaus kunnen we de uit te werken opdracht zien als de specificatie en de rij van kleinere opdrachten als het berekeningsvoorschrift. Het gaat dus op elk van deze niveaus om *specificatie* van berekening.

Ik wil u in deze voordracht kennis laten maken met de informatica als wetenschap (1). Ik zal dit doen door enkele kernbegrippen uit de informatica met u te bespreken, om daarna in te gaan op de plaats die de informatica tussen de andere wetenschappen inneemt.

Informatica is een wetenschap met duidelijke doelen, methoden en concepten (2). Het is niet meer dan dat: wij kunnen u geen steen der wijzen bieden. Het is niet minder: wij hebben inmiddels technieken en concepten tot onze beschikking die een geinteresseerde natuurkundige niet op een achternamiddag bedenkt. Het is meer dan een ambacht, hoewel het vak inderdaad een grote ambachtelijke component heeft.

U moet van de computer niet de oplossing voor alle maatschappelijke problemen verwachten. Reeds in Marten Toonder's boek "De Klonters" [Too57] werd hiermee effectief de draak gestoken. We horen daar het mechanisch superbrein geschapen door Professor Sickbock, het parkeerprobleem in Rommeldam als volgt oplossen: "Er rinkelde iets, ergens klonk een fluittoon en tenslotte begonnen er een paar lampen te branden. Daarop opende hij de mond 'Oplossing parkeerprobleem Rommeldam', sprak hij met metalen stem, 'Huizen in binnenstad afbreken, puinruimen, pleinen maken.'" Hoewel onze superbreinen tegenwoordig misschien verfijnder zijn dan Marten Toonder toen kon voorzien, moet u ook van de huidige computer geen goede oplossingen voor dit soort problemen verwachten.

Mijn bespreking van de informatica is noodzakelijkerwijze eenzijdig. Ik zal u niet over mijn schouder mee laten kijken, terwijl ik een computerprogramma ontwerp, of terwijl ik een bewijs tracht te vinden voor de gelijkwaardigheid van twee computerprogramma's, of terwijl ik een stelling tracht te bewijzen, die voor alle mogelijke computerprogramma's geldt. Ik zal mij in plaats daarvan concentreren op begripsvorming.

1 Programmacorrectheid

Mijn leeropdracht is de leer van de programmacorrectheid. Dit betreft de ontwikkeling en het gebruik van formele methoden om de correctheid van computerprogramma's te bevorderen of vast te stellen. Dit is een hele mond vol. Ik zal deze begrippen daarom één voor één bespreken, en wel in omgekeerde volgorde. Een programma is (in dit verband) een opdracht aan een computer, doorgaans een samengestelde opdracht in de trant van

```
doe A ;
zolang P geldt doe B ;
doe tenslotte C .
```

Of iets dergelijks. Wat betekent correctheid van een dergelijke opdracht nu? We bedoelen met correctheid niet, dat de opdracht goed gesteld is en dat de computer de opdracht kan interpreteren en uitvoeren. Deze eis kan namelijk door de computer zelf nagegaan worden (daar heb je dus geen hoogleraar voor nodig). Elke opdracht is, voorzover hij op zichzelf staat, correct, maar als u wel eens met een computer gewerkt heeft, weet u, dat je het apparaat wel eens een opdracht geeft die je niet bedoelde.

Het programma is namelijk niet zomaar een opdracht, nee, het is een *berekeningsvoorschrift* met één of ander *doel*. Een voorbeeld van een programma waar de meesten van u wel eens mee gewerkt hebben, is een tekstverwerker. Dit is inderdaad een programma, een berekeningsvoorschrift, en het is ooit geschreven door een programmeur of een team van programmeurs. Het heeft het doel, dat de gebruiker een tekst kan invoeren, aanvullen, opslaan, raadplegen, wijzigen en afdrukken. De afgedrukte tekst hoort dan overeen te stemmen met de tekst, zoals die in één of meer zittingen is ingevoerd, en zoals die er uiteindelijk uit bleek te zien. Dit is een correctheidseis, en uit het berekeningsvoorschrift blijkt meestal niet direct, of aan deze eis voldaan wordt.

Een ander voorbeeld. Staartdelen, weet u nog wel? Dat is ook een berekeningsvoorschrift. Toen u op de lagere school leerde staartdelen, hebt u zich de vraag misschien niet gesteld, maar het is bepaald niet vanzelfsprekend, dat bij het staartdelen van een getal x door de deler y (die > 0 is) een quotient q en een rest r verkregen worden, waarvoor geldt dat $x = q \cdot y + r$ en dat $0 \leq r < y$. Dit is echter wel de specificatie voor deling met rest. De vraag of het staartdelingsalgoritme altijd aan deze specificatie voldoet, is een vraag van programmacorrectheid.

Correctheid spreekt niet vanzelf

Correctheid houdt dus in, dat het programma aan zijn specificatie voldoet. U zult misschien vragen: spreekt dat niet vanzelf? Inderdaad, zeergewaardeerde toehoorders, het zou vanzelf moeten spreken, dat het programma aan zijn specificatie voldoet. Evenzeer, als het vanzelf moet spreken, dat als je een middag achter de kassa van een supermarkt hebt gezeten, de som van de kosten van de afgerekende boodschappen gelijk is aan het verschil tussen de hoeveelheid geld in de kas na afloop en van tevoren. In beide gevallen geldt, dat het vanzelf zou moeten spreken, maar helaas bepaald niet altijd waar is. Mensen maken fouten, of het nu programmeurs zijn of kassières.

Nu ik deze vergelijking eenmaal gemaakt heb, moet ik u echter ook op enige verschillen wijzen. Bij het bedienen van de kassa moet u telkens opnieuw één klant bedienen, de boodschappen aanslaan en de kosten correct verrekenen. De transacties met verschillende klanten zijn onderling onafhankelijk (ik ga even voorbij aan de klant die even later terug komt met de klacht dat u hem te weinig geld hebt teruggegeven). De programmeur echter werkt aan een programma dat misschien duizend, misschien honderdduizend regels lang is. Elke wijziging in één programmaregel kan op allerlei andere plaatsen in het programma effect hebben. De programmeur moet dus voortdurend rekening houden met een groot aantal details uit het

hele programma.

Een tweede verschil is, dat het programma vermoedelijk vele malen zal worden uitgevoerd. Die ene programmeerfout kan dus telkens weer tot grote ergernis of ongelukken aanleiding geven.

Bovendien, ik zei zo eenvoudig, dat het programma aan zijn specificatie moet voldoen. Hier zitten echter grote problemen. Het programma heeft vaak geen volledige en eenduidige specificatie. Of, als het al een goede specificatie heeft, is dat vaak een lijvig document, dat de programmeur niet uit zijn hoofd kan kennen. Tenslotte, zelfs bij een eenvoudige specificatie en een kort programma kan het al heel lastig zijn om de correctheid aan te tonen of te weerleggen.

Ik heb u aangegeven, dat het begrijpelijk is wanneer een programmeur fouten maakt. Ik zou u nu een aantal afschrikwekkende verhalen kunnen vertellen over feitelijk opgetreden fouten in computerprogramma's (zie b.v. [Gib94]) . Ik pleeg dergelijke verhalen echter zo vlug mogelijk te vergeten. Eén verhaal wil ik u niet onthouden.

Een aantal jaren geleden ontwierp men een systeem voor vliegtuignavigatie als uitbreiding van de automatische piloot. Dit systeem bevatte een kleine fout: als het vliegtuig de evenaar passeerde, draaide het zich om en trachtte op de rug verder te vliegen. Ik weet niet, of het verhaal historisch is, maar ik zou dat niet kunnen uitsluiten. Het verhaal illustreert duidelijk het belang van programmacorrectheid, althans voor mensen die in een vliegtuig de evenaar willen passeren.

Correctheid en formele methoden

Nu we inzien, dat de correctheid van programma's niet vanzelf spreekt maar wel belangrijk is, komen we bij de vraag hoe we de correctheid van programma's kunnen bewerkstelligen of bevorderen.

Het eerste punt is het opstellen van een volledige, eendui-

dige en toch heldere (bij voorkeur korte) specificatie voor het programma. Hoewel het opstellen van de specificatie in het algemeen veel voeten in aarde heeft, en een eenmaal gemaakte specificatie ook wel weer gewijzigd zal worden, wil ik voor de verdere bespreking ervan uitgaan, dat de specificatie gegeven is. De vraag is dan, hoe de correctheid van het programma te bevorderen of vast te stellen.

Als het programma ook reeds gemaakt is, kunnen we het natuurlijk gaan testen om te zien of het doet wat ons beloofd wordt. Zo'n test is heel geschikt (en nodig) om fouten in het programma te vinden. Omdat we het programma echter niet onder alle mogelijke omstandigheden kunnen testen, kan een test nooit de afwezigheid van fouten aantonen (dit is een opmerking van Dijkstra uit 1972, zie [DDH72] p. 6). Voor elk product zijn tests in de praktijk nodig om te zien of het product aan de verwachtingen voldoet. Voor computerprogramma's helpen tests echter niet om correctheid vast te stellen. Wat dan?

Ik sprak hierboven over formele methoden om de correctheid van programma's te bevorderen of vast te stellen. Formele methoden zijn wiskundige technieken gebaseerd op het idee dat specificaties en programma's wiskundige formules zijn, d.w.z. gestructureerde objecten opgebouwd uit wiskundige symbolen. Het ideaal is, dat het programma en de specificatie allebei wiskundige formules zijn en dat je dan kunt uitrekenen of het programma aan de specificatie voldoet. We gaan er hierbij van uit, dat de toepasser van een formele methode zich niet laat leiden door zijn inzicht in de betekenis van de specificatie of het programma, maar alleen door de wiskundige structuur van zijn formules (3).

Dit uitgangspunt is formeler dan wat in de wiskunde zelf gebruikelijk is. De reden hiervan is, dat de formules die wij hanteren doorgaans ingewikkelder zijn dan bij de wiskundigen zelf (ik wees er reeds op, dat er computerprogramma's van meer dan honderdduizend regels zijn). Hiermee hangt samen, dat wij vaak minder abstracties tot onze beschikking hebben dan

de wiskundigen, zodat we in een chaotischer landschap moeten werken waar we minder op de intuïtie kunnen vertrouwen (4).

Een tweede verschil is, dat de formules voor de wiskundigen meestal slechts een hulpmiddel zijn om een relatie uit de drukken tussen de objecten waarin zij geïnteresseerd zijn. Voor de informaticus is de formule (d.w.z. het programma of de specificatie) het centrale object van studie. Hij heeft concretere vragen, wil van het gegeven programma weten, dat het inderdaad aan de gegeven specificatie voldoet. Hij kan zich daarbij geen rekenfout veroorloven, en achteraf zeggen, dat het idee van zijn programma wel goed is en dat zijn programma voor bijna elke invoer aan de specificatie voldoet ("Wie wil er nu ook in een vliegtuig over de evenaar vliegen?").

Het onderwijs in formele methoden stelt ons voor een didactisch probleem. De formele methoden zijn nodig om het vertrouwen in de correctheid van programma's te vergroten. Deze noodzaak doet zich pas voelen als men te maken heeft met gecompliceerde programmeerproblemen. We kunnen echter bij het onderwijs in de formele methoden niet met ingewikkelde problemen beginnen. Het handwerk moet met eenvoudige probleempjes geleerd worden. We moeten de studenten dus dwingen de formele methoden te leren hanteren bij programmeerproblemen waar de correctheid ook operationeel eenvoudig is in te zien. Dit werkt vaak demotiverend, en leidt soms tot hypocrisie en lippendienst.

Het is vergelijkbaar met de behandeling van een lui oog: om te voorkomen, dat een kind een bepaald oog niet gebruikt, wordt het andere oog afgeplakt. Op dezelfde wijze moeten wij studenten tijdelijk verbieden om operationeel inzicht te gebruiken. We komen dan later weer voor het integratieprobleem te staan: hoe te bevorderen dat de studenten hun beide ogen open houden en gebruiken, en niet uitsluitend vertrouwen op hetzij operationele intuitie, hetzij formele methoden.

Op het gevaar af de vergelijking in het absurde te trekken kan ik nog toevoegen, dat het gebruik van twee ogen het veel

makkelijker maakt perspectief en diepte te zien.

Er is trouwens een tweede didactisch probleem: de huidige generatie studenten weet niet meer wat een wiskundig bewijs is. Sommige ouderen onder u herinneren zich nog hoe je zo'n bewijs moest opzetten: eerst het gegeven, dan wat er te bewijzen was, en daarna kwam het bewijs. Het onderwijs hierin op de middelbare school is helaas verloren gegaan. Hierdoor is een ernstig manco ontstaan bij de voorbereiding op een aantal universitaire studies. Dit gemis speelt ook in sterke mate bij de informatica. Enerzijds hebben wij juist grote behoefte aan strikte wiskundige bewijzen; anderzijds moet een representatieve en oriënterende propedeuse in de informatica ook veel andere onderwerpen bevatten, zodat we aan strikte bewijsvoering niet alle aandacht kunnen besteden. Ik acht het derhalve van groot belang, dat het leveren van wiskundige bewijzen weer wordt opgenomen in de wiskunde-eindexamenstof van het VWO (5).

Parallelle verwerking

Terug naar de programmacorrectheid. Na een eerste aanzet van Turing zelf in 1949 (zie [Tur49]) is de zorg en aandacht voor programmacorrectheid pas goed opgekomen aan het eind van de zestiger jaren (zie b.v. [Nau66, Flo67]), tegelijk met de opkomst van wat in die tijd multiprogramming genoemd wordt: dit is het ontwerpen van een verzameling programma's met één gemeenschappelijk geheugen, die min of meer tegelijkertijd verwerkt worden.

Het is geen toeval, dat programmacorrectheid en multiprogramming tegelijk opkwamen. De correctheid van een gewoon programma is in veel gevallen vrij gemakkelijk aan te tonen, d.w.z. met een bewijs waarvan de lengte min of meer evenredig is met de lengte van het programma zelf. Wanneer verschillende programma's tegelijk het gemeenschappelijke geheugen kunnen raadplegen of zelfs wijzigen, wordt het veel moeilij-

ker om de correctheid aan te tonen. Voor samenwerkende programma's geldt b.v. vaak, dat de lengte van het correctheidsbewijs evenredig is met het product van de lengtes van de afzonderlijke programma's. U moet deze bewering vooral niet letterlijk nemen, maar het is een redelijke vuistregel en het geeft een indicatie, dat multiprogramming echt moeilijker is en dus meer behoefte heeft aan volledige correctheidsbewijzen dan gewoon programmeren (6).

Het woord multiprogramming is nooit aangeslagen. We spreken tegenwoordig van concurrency of parallellisme, of mogelijk van gedistribueerde of verkavelde berekeningen. Ik zeg niet, dat dit allemaal precies hetzelfde is (de meningen daarover zijn verdeeld), maar vanuit het oogpunt van het ontwerp en de correctheid van programma's zijn de verschillen tussen deze begrippen van ondergeschikte betekenis.

Begrensd wachten

Ik wil u in het verband met parallellisme een voorbeeld geven waar ik zelf aan gewerkt heb.

Stel u voor dat een luchtvaartmaatschappij een data-base heeft waarin staat welke vliegtuigstoelen bij welke vluchten bezet zijn en dat men bij verschillende reisbureaus stoelen kan reserveren. Dit betekent, dat verschillende computers de data-base min of meer gelijktijdig moeten kunnen raadplegen en wijzigen. De data-base moet echter betrouwbaar zijn: het moet niet kunnen voorkomen, dat twee verschillende reisbureaus tegelijkertijd één en dezelfde lege stoel waarnemen en reserveren. We zouden het natuurlijk zo kunnen regelen, dat er altijd maar één reisbureau tegelijk geholpen wordt. Dit heeft echter het risico, dat de computer van dat éne reisbureau tijdens een transactie uitvalt zonder dat aan de data-base te kunnen melden. In dat geval zouden alle andere reisbureaus misschien wel heel lang op dat ene bureau moeten wachten. We hebben dus behoefte aan parallelle verwerking zodanig, dat het logische ge-

drag van de data-base is alsof de transacties in één of andere volgorde plaats vinden, terwijl toch elk van de reisbureaus voor elke transactie een begrensde wachttijd heeft.

Ik heb u het probleem uitgelegd voor vliegtuigstoelen, maar het is een heel algemeen en fundamenteel probleem om meerdere processen gelijktijdig gemeenschappelijke data te laten raadplegen en wijzigen.

Ik kwam het probleem tegen, toen we een jaar of drie geleden in onze wekelijkse werkgroep een artikel ([Her91]) van Herlihy bestudeerden. De schrijver gaf een oplossing van het probleem, waarvan we echter zelfs na zorgvuldige bestudering niet konden inzien of deze al dan niet correct was. Zoiets is natuurlijk een uitdaging. Ik heb er lang aan gewerkt en uiteindelijk, geïspireerd door Herlihy's programma, een oplossing gevonden waarvan ik de correctheid wel kon aantonen. Het is dan een programma van ongeveer 32 enkelvoudige opdrachten. De moeilijkheid is dat het programma bij elk van de reisbureaus tegelijk loopt. Er worden dus parallel met elkaar even veel versies van het programma uitgevoerd als er reisbureaus zijn.

Op grond van de bovengenoemde vuistregel moeten we er rekening mee houden (zelfs als er maar twee reisbureaus zijn), dat het bewijs een lengte kan hebben evenredig met 32^2, dat is ongeveer duizend. Inderdaad, mijn bewijs was zo lang en bevatte zoveel gevalsonderscheidingen (zie [Hes94b]), dat ik elke keer als ik mijn bewijs weer eens langsliep, aan het eind gekomen niet meer wist of ik in het begin wel met alle mogelijkheden rekening gehouden had. Zo'n bewijs is voor andere mensen nauwelijks te lezen, en dus niet echt overtuigend.

Ik heb het bewijs daarom aan een mechanische stellingbewijzer voorgelegd (zie [Hes95]). Een mechanische stellingbewijzer is een computerprogramma, dat eenvoudige wiskundige stellingen kan bewijzen en dat je moeilijker wiskundige stellingen kunt laten bewijzen door hem de goede aanwijzingen te geven. Ik zal dadelijk meer ingaan op het gebruik van stellingbe-

wijzers, maar laat ik eerst vertellen hoe het met de correctheid van mijn programma afgelopen is.

De opdracht die ik aan de stellingbewijzer gegeven heb, bevatte uiteindelijk het programma en het correctheidsbewijs ervan. Dit bewijs is opgebouwd uit een aantal hoofdstellingen (de specificatie van het programma) en een groot aantal hulpstellingen (om de stellingbewijzer te helpen). Het totaal was ongeveer 6500 regels lang, met misschien vijf woorden per regel. Ik heb door het gebruik van de stellingbewijzer één foutje in het bewijs gevonden. Dit foutje kon echter eenvoudig hersteld worden. Ik heb er een aantal maanden werk in zitten, maar de winst is, dat ik nu zelf overtuigd ben van de correctheid van het programma en dat ik de stellingbewijzer heb leren hanteren.

Het gebruik van stellingbewijzers

Het gebruik van stellingbewijzers roept vaak nog al gemengde reacties op. Allereerst zijn er twee fundamentele problemen. Het eerste is de vraag: "Wie bewijst de correctheid van de stellingbewijzer?" Dit is een serieuze vraag, waarop ik de volgende twee argumenten pleeg aan te voeren.

1. Ik heb gebruik gemaakt van de stellingbewijzer NQTHM van Boyer en Moore uit Austin (zie [BoM88]) die een zeer goede naam heeft.

2. Zelfs als NQTHM een fout zou bevatten, is het heel onwaarschijnlijk, dat het mij zou lukken om deze fout uit te buiten voor het bewijzen van een onjuiste stelling.

Een tweede probleem is de vraag of datgene wat ik de stellingbewijzer heb laten bewijzen, wel overeenstemt met de bewering dat mijn programma het bovengenoemde probleem oplost. Ook dit is bepaald niet vanzelfsprekend. De stellingbewijzer bewijst namelijk alleen stellingen die in zijn taal geschreven zijn. We hebben hier dus een vertaalprobleem: komen de bewezen stellingen (geschreven in de formele taal van de stellingbewijzer) overeen met de gegeven probleemstelling (die in het ne-

derlands of engels geschreven is). Dit kan alleen beoordeeld worden door een mens die zowel de natuurlijke taal als de taal van de stellingbewijzer kent. In beide gevallen geldt: absolute zekerheid bestaat niet in het leven, maar een mens kan ervoor kiezen zich door argumenten te laten overtuigen.

Het gebruik van een stellingbewijzer roept ook vaak emotionele bezwaren op. Vooralsnog hoeft de doorsnee–wiskundige niet bang te zijn voor zijn baan. NQTHM is één van de intelligentste stellingbewijzers, maar zijn vaardigheid in het bewijzen van stellingen is niet echt groter dan van een eerstejaars student. De grootste verschillen zijn, dat NQTHM niet gevoelig is voor intimidatie en dat hij heel goed in staat is bewezen stellingen te onthouden en toe te passen. Anderzijds is elke middelbare scholier al oneindig veel creatiever en flexibeler dan NQTHM.

Men noemt ook wel het bezwaar, dat je om de stellingbewijzer te overtuigen vaak eerst zelf het bewijs heel goed door moet hebben om vervolgens nog vrij veel werk te hebben in de communicatie met de stellingbewijzer. U kunt dit natuurlijk ook zien als een voordeel (waarbij het ieder vrij staat om de communicatie met de stellingbewijzer alsnog achterwege te laten). De stellingbewijzer was voor mijn toepassing nodig om de administratie van de bewezen stellingen sluitend te houden.

Het grote belang van bewijzen geverifieerd met een stellingbewijzer is, dat men het bewijs van een stelling kan controleren door de bewijsverplichtingen te controleren en de aan de stellingbewijzer geleverde aanwijzingen opnieuw in te voeren. Op deze wijze is de tekst voor de stellingbewijzer dus een certificaat van correctheid. Aangezien in de toekomst aantoonbare correctheid van software ongetwijfeld een veelvoorkomend onderwerp van rechtzaken zal zijn, kan een dergelijk certificaat grote practische waarde hebben.

Een real-time kernel

Eén van de frappante aspecten van de informatica is het grote practische belang van abstracte en algemeen toepasbare constructies. Het proefschrift [Tol95] van Ronald Tol, b.v., dat nu bijna voltooid is, behandelt het ontwerp van een real-time kernel.

Een real-time systeem is een computersysteem dat processen waarneemt en regelt. Men kan denken aan het regelen van een spoorwegovergang. Als er een trein aankomt, moeten de bomen tijdig naar beneden. Als de trein voorbij is, moeten de bomen weer omhoog, tenzij er nog een trein komt. Er zijn echter tal van andere real-time systemen, in wasmachines, kerncentrales, videorecorders, vliegtuigen, enz.

Een real-time kernel is het centrale deel dat al deze systemen gemeen hebben. Het is het programma dat regelt dat alle inkomende signalen verwerkt worden en dat alle taken van het systeem tijdig (d.w.z. elk vóór zijn eigen deadline) opgestart worden.

Omdat elk goed ontworpen real-time systeem een dergelijke kernel moet hebben, loont het de moeite om een goedgespecificeerde, algemeen toepasbare real-time kernel te ontwerpen. De algemene toepasbaarheid vergt echter abstractie: we kunnen b.v. niet praten in termen van treinen, spoorbomen en passerende auto's, maar we moeten praten over gebeurtenissen en taken. Het vergt dan een soort abstract inlevingsvermogen om criteria aan te geven en te verifiëren, die behelzen dat de kernel correct en tijdig reageert, terwijl we niet weten in wat voor applicatie de kernel wordt ingezet.

Het zal duidelijk zijn hoe belangrijk de correctheid van de real-time kernel is. Als de kernel niet correct is, is alles wat erop gebouwd is, van vliegtuig tot wasmachine, onbetrouwbaar.

2 Nondeterminisme in de informatica

Ik wil nu de problemen van de begripsvorming in de informatica illustreren aan de hand van een begrip, dat op verschillende plaatsen in de informatica opduikt en dat ook verschillende verschijningsvormen heeft. Het gaat mij om het begrip *nondeterminisme*, letterlijk *onbepaaldheid*, of misschien beter *onderbepaaldheid*.

Onder nondeterminisme verstaat men in de informatica het verschijnsel, dat een berekening bij één en dezelfde invoer verschillende resultaten kan opleveren (7). Dit lijkt op het eerste gezicht misschien ongewenst. Bij nadere beschouwing blijkt echter dat nondeterminisme noodzakelijk is als we programma's of systemen willen specificeren, ontwerpen of analyseren zonder te verdrinken in een zee van irrelevante details. Eén voorbeeld: wanneer u als gebruiker van een computersysteem een bestand naar een schijf wegschrijft, wilt u er doorgaans geen weet van hebben, waar het bestand op de schijf terecht komt. Voor een enigszins abstracte specificatie van de schrijfopdracht zal de plaats op de schijf dus een nondeterministisch resultaat zijn.

De introductie van nondeterminisme hangt samen met de noodzaak een ingewikkeld systeem in een aantal stadia te ontwerpen. Het is dan verstandig om ontwerpbeslissingen zo lang mogelijk uit te stellen. De noodzaak tot een keuze kan zich namelijk in een vroeg stadium opdringen, terwijl de redenen waarom een bepaalde keuze de voorkeur verdient pas veel later bekend zijn. In dat geval doet de ontwerper er goed aan een nondeterministische keuze te gebruiken, die in een later stadium misschien weer door een deterministische keuze vervangen wordt.

Nondeterminisme is dus het verschijnsel, dat bij gegeven begintoestand verschillende eindtoestanden mogelijk zijn. Laten we nu aannemen, dat de programmeur een zeker doel met zijn programma had, d.w.z. een beoogde eindconditie. Door het nondeterminisme komen we dan voor de vraag te staan of

alle mogelijke eindtoestanden aan dit doel beantwoorden, of alleen sommige, of misschien zelfs geen van de eindtoestanden.

Als de programmeur de taak heeft te zorgen dat $alle$ mogelijke eindtoestanden aan de beoogde eindconditie voldoen, spreken we van demonisch nondeterminisme. We kunnen de nondeterministische keuze dan immers gerust aan een "kwaadwillende demon" overlaten. In specificaties is deze vorm van nondeterminisme het meest gebruikelijk.

Een geheel andere vorm is angeliek nondeterminisme: dit drukt uit, dat tenminste één van de mogelijke eindtoestanden aan de beoogde eindconditie voldoet. We laten de nondeterministische keuze dan als het ware over aan onze "persoonlijke beschermengel". Vandaar de toevoeging "angeliek". Angeliek nondeterminisme wordt bij specificaties gebruikt om een keuze, die we nog niet kunnen of willen vastleggen, uit te stellen (8).

Angeliek nondeterminisme heeft –vreemd genoeg– in de informatica de oudste rechten. Het werd in 1959 ingevoerd door Rabin en Scott in hun theorie van nondeterministische eindige automaten (zie [RaS59]). Het belang van demonisch nondeterminisme voor specificatie en programmacorrectheid werd voor het eerst duidelijk naar voren gebracht in Dijkstra's boek "A Discipline of Programming" uit 1976 ([Dij76]).

Er valt over nondeterminisme veel meer te vertellen (het is een fascinerend onderwerp, waar ik me uitgebreid mee bezig gehouden heb), maar ik zal dat in verband met de tijd nu niet doen (9).

3 Informatica tussen de andere wetenschappen

Ik wil nu ingaan op de plaats van de informatica tussen de andere wetenschappen. Vanwege onze positie in de Faculteit van de Wiskunde en de Natuurwetenschappen is misschien één van de eerste vragen of de informatica een natuurwetenschap is. In

een recente rede gaf Hartmanis hierop een duidelijk antwoord. Hij stelde (ik citeer [Har93], (10)):

> Het is duidelijk dat informatica geen natuurwetenschap is. Toch neemt men vaak aan, dat zij grote overeenkomsten met de natuurwetenschappen zal vertonen en soortgelijke onderzoeksparadigma's zal hebben met betrekking tot de relatie tussen theorie en experiment. Dit is ten onrechte. Theorie en experiment spelen in de informatica een geheel andere rol dan in de natuurwetenschappen.

De vraag is dan dus eerst:

Is informatica een experimentele wetenschap?

Ik heb u aan de hand van de begrippen correctheid en nondeterminisme een indruk gegeven van de vraagstellingen in de informatica. Het is u daarbij misschien al opgevallen, dat ik geen enkel experiment genoemd heb. Een gemaakt programma zal natuurlijk getest moeten worden, maar als er dan een programmeerfout gevonden wordt, is dat geen wetenschappelijk resultaat, en als er geen fout gevonden wordt, betekent dat wetenschappelijk ook niets.

We hebben namelijk in de informatica geen wetenschappelijke hypothesen die door een experiment gefalsifieerd kunnen worden. Een voorbeeld dat er wel dichtbij komt, is de uitspraak: "Het is niet mogelijk een schaakprogramma te maken dat de wereldkampioen verslaat." Je kunt nu met wat goede wil de beslissende schaakpartij een experiment noemen (11). Voor de informatica zou die partij echter niet interessant zijn, maar het programma misschien wel. Het betreffende programma zou als een demonstratieobject kunnen dienen.

Men versta mij niet verkeerd: hoewel de informatica dus geen experimentele wetenschap is, moeten wij wel veel practisch werk verrichten. Het is misschien illustratief de compu-

terpractica van onze studenten te vergelijken met het natuurkundepracticum, dat ik zelf dertig jaar geleden in Utrecht deed. Er zijn grote uiterlijke overeenkomsten: er is een welomschreven opdracht, er is een stuk theorie dat bestudeerd moet worden, en de opdracht moet uitmonden in een verslag dat de uitvoering van de opdracht en de resultaten ervan weergeeft. De essentie van de opdracht is echter totaal verschillend. De kern van de natuurkundeproef bestaat uit de metingen en de interpretatie daarvan. De kern van het computerpracticum is het ontwerp, b.v. van een programma, en de implementatie daarvan. Natuurlijk moet het programma ook getest worden, maar dat is een noodzakelijke *nevenactiviteit*.

Ook ons wetenschappelijk werk bevat een grote practische component. Dit kan b.v. bestaan uit het ontwerpen van een computerprogramma of het leveren van een correctheidsbewijs met behulp van een mechanische stellingbewijzer. Men kan hierbij echter niet spreken van experimenten. Het practisch werk is namelijk niet gericht op beter inzicht in de werkelijkheid, maar soms op producten en anders op betere methoden om complexe systemen te ontwerpen.

Is informatica dan een technische wetenschap?

Mijn antwoord op deze vraag is zonder meer: "Ja." Jan van de Snepscheut drukte het indertijd kernachtig uit in de titel van zijn inaugurele rede [vdS85]: "Wij zijn ontwerpers". Bij een technische wetenschap gaat het volgens mij namelijk om het ontwerp van dingen en om de methoden van productie van deze dingen, waarbij gebruik gemaakt wordt van de mogelijkheden die de natuur ons biedt en rekening gehouden wordt met de beperkingen die de natuur ons oplegt. Dit alles is duidelijk van toepassing op de informatica.

Het is echter wel zo, dat de informatica zich in veel sterkere mate onafhankelijk van de natuurwetenschappen kan en moet opstellen dan de meeste andere technische wetenschappen. Ons

hoofdmateriaal is namelijk informatie, een abstractum, dat in de practijk natuurlijk altijd een fysische representatie moet hebben, maar waarvan het gedrag toch in hoge mate onafhankelijk is van de representatie. De mogelijkheden en beperkingen die de natuur ons biedt dan wel oplegt, betreffen vooral de snelheid en de betrouwbaarheid of onbetrouwbaarheid van de informatiestromen. Dit zijn zeker belangrijke dingen, maar zij zijn onafhankelijk van de betekenis van de informatie.

Laat ik de informatica vergelijken met b.v. de technische mechanica. In de technische mechanica zou men een brug kunnen ontwerpen. De mogelijkheden en beperkingen van de natuur komen dan tot uiting in de eigenschappen van de te gebruiken materialen en in de krachten die op de brug zullen gaan werken. Men zal ervoor moeten zorgen, dat de brug niet door de wind of door marcherende soldaten zodanig in beweging komt, dat hij breekt. Het ontwerp van de brug is dus sterk afhankelijk van een groot aantal kwantitatieve aspecten van natuurwetenschappelijke oorsprong.

Bij het programmeren daarentegen zijn kwantitatieve aspecten van natuurwetenschappelijke oorsprong heel vaak afwezig. Zelfs als het gaat om communicatie tussen computers onderling kan men de fysische aspecten isoleren en door een electrotechnicus laten verzorgen, terwijl de informaticus zich kan en moet concentreren op de behandeling van de informatiestroom zelf.

Ik beschouw de informatica dus als een technische wetenschap, die welhaast evenzeer onafhankelijk is van de natuurwetenschappen als de psychologie onafhankelijk is van de biologie en van de medische wetenschap.

Informatica als onderdeel van de wiskunde?

Ook hier is een vergelijking met technische mechanica op zijn plaats. Technische mechanica wordt hier in Groningen gerekend tot de wiskunde en is zelfs onderdeel van de betreffende

vakgroep. De informatica is geen onderdeel van de vakgroep Wiskunde maar wel aan wiskunde gekoppeld in het Instituut voor Wiskunde en Informatica. Ik vind dit de verhoudingen wel aardig weergeven. Laat ik u trachten duidelijk te maken waarom.

De informatica gebruikt net als de wiskunde formules, abstracties en axiomatisering, strikte definities en sluitende bewijsvoeringen. In zekere zin houdt het daarbij op. De informatica gebruikt maar een heel klein deel van alle wiskundige theorie die in de laatste eeuwen en met name de laatste vijftig jaar verzameld is.

Bij de technische mechanica gaat men uit van delen van de natuurkunde waarin de fundamentele wetten bekend zijn. Deze wetten zijn kwantitatieve wiskundige vergelijkingen, waarvan de oplossing voor een gegeven ontwerp in het algemeen onbekend is. Hier ligt dan een wiskundig probleem, dat men kan trachten op te lossen met methoden uit de analyse of de numerieke wiskunde. Vandaar de nauwe relatie tussen technische mechanica en wiskunde.

De wiskundige methoden die in de technische mechanica gebruikt worden, vormen geaccepteerde onderdelen van de wiskunde, waarin ook voor wiskundigen interessante vragen zitten. Bij informatica ligt dit anders. Informatica gebruikt dergelijke wiskunde ook wel, zij het in mindere mate. Informatica gebruikt echter vooral delen van de wiskunde die de wiskundigen in mindere mate als wiskunde ervaren.

We hebben in de informatica van tijd tot tijd behoefte aan wiskundige theorie die door de wiskundigen zelf nog niet bedacht is en die verder af staat van de hoofdstroom van de wiskunde dan wat de natuurkundigen inbrengen. Bij natuurkundige problemen gaat het immers over kwantitatieve aspecten van de natuur en dus over continue of differentieerbare functies. In de informatica gaat het meestal over kwalitatieve aspecten van informatie, dus over functies met als waarden ja of nee. We noemen zulke functies Boole'se functies naar de

wiskundig–logicus Boole, die de algebra van de waarheidswaarden ontwikkeld heeft. Dit voert ons tot de vraag:

Is informatica dan soms toegepaste logica?

Ik kan deze vraag moeilijk ontkennen. Het lijkt me echter, dat alle wetenschappen als toegepaste logica beschouwd kunnen worden. Het feit dat er in de informatica meer met waarheidswaarden gerekend wordt dan in andere wetenschappen, maakt het niet specifieker toegepaste logica. Immers ook in de logica zelf wordt in het algemeen niet met waarheidswaarden *gerekend*. Onze rekenpartijen met waarheidswaarden zijn voorts alleen daarom interessant, omdat ze samenhangen met de complexiteit van de informatie. De logica helpt ons niet deze complexiteit te beheersen. We hebben hiervoor wiskunde nodig, maar in het algemeen is dit wiskunde die we zelf nog moeten ontwikkelen.

Het belangrijkste deel van de wiskunde dat de informatici gebruiken, is echter de wiskundige taal. Om te redeneren over de dingen die zij interessant vonden, hebben de wiskundigen een wiskundige taal van formules en dergelijke ontwikkeld. De wiskundigen gingen dan verder en gebruikten die taal om mooie stellingen over mooie wiskundige strukturen te bewijzen. De informatica gebruikt nu diezelfde wiskundige taal om informatiesystemen of programma's te specificeren. We kunnen de mooie wiskundige stellingen doorgaans niet gebruiken, we gebruiken alleen het formalisme. Voor wiskundigen is dat echter geen wiskunde, alleen maar taal.

Informatica een onderdeel van de taalkunde?

Inderdaad, hier valt wel wat voor te zeggen. Een informaticus moet zich voortdurend vergewissen van de relaties tussen woorden en hun betekenissen. Een informaticus moet verschillende soorten computertalen beheersen. Het is moeilijk vergelijken,

maar ik denk, dat de verschillen tussen programmeertalen als Pascal en Prolog groter zijn dan die tussen natuurlijke talen als nederlands en hebreeuws. De informaticus heeft voorts nog te maken met specificatietalen en met talen waarmee hij een operating system of een data-base kan aanspreken. Ditzelfde geldt natuurlijk voor iedereen die met een computer werken moet, maar de informaticus heeft de verantwoordelijkheid voor het ontwerp van zulke talen.

Dit laatste brengt ons bij een opvallend verschil met de taalkunde. De informaticus kan misschien talen ontwerpen, de taalkundige doet dit vrijwel nooit. Voor de taalkundige is de taal een object van studie. De informaticus zou de taal graag zien als een goed hanteerbaar instrument, maar is vooralsnog zo vaak ontevreden over de werkstukken van zijn collega's, dat hij maar weer een nieuwe taal ontwerpt. De taal is voor ons inderdaad een instrument, net als de wiskunde. Het is niet het object van studie (12).

Mijn conclusie

Ik wil ons vakgebied niet indelen naar de aard van zijn instrumenten, maar naar de aard van zijn doelstellingen en objecten. Het is daarom, dat ik informatica beschouw als een *ontwerpwetenschap*, een technische wetenschap die niet hoort bij de natuurwetenschappen en ook geen onderdeel is van de wiskunde, de logica of de taalkunde. Ik beschouw deze laatste drie vakgebieden wel als de meest verwante vakgebieden, en de verwantschap met de wiskunde is het grootste.

Laat ik besluiten met een opmerking over de plaats van de informatica in het geheel van de wetenschappen. Door het onderzoeken van de mogelijkheden en de beheersbaarheid van informatieverwerking komen we in de buurt van de bron van de kennis, en dat is iets dat ons allen in de universiteit ter harte gaat.

4 Slotwoorden

Mijnheer de Rector, zeergewaarde toehoorders.

Tot slot dan enige persoonlijke woorden.

Allereerst dank ik het Bestuur van het Groninger Universiteitsfonds, het College van Bestuur van deze universiteit en alle andere universitaire instanties die mijn benoeming mogelijk gemaakt hebben.

Ik ben twintig jaar geleden gepromoveerd in de zuivere wiskunde. Tien jaar geleden ben ik overgestapt naar de informatica. Deze stap sloot aan bij mijn wetenschappelijke ontwikkeling, maar bleek wel groter dan ik had voorzien. Het was een moeilijke beslissing, maar ik heb er geen spijt van gehad.

Ik heb in de loop der jaren van veel wiskundigen en veel informatici veel dingen geleerd. Het aantal wiskundigen en informatici dat aan mijn vorming heeft bijgedragen is echter te groot om hen allen te noemen.

Dames en heren studenten. Formele methoden zijn er niet om u te plagen, maar om u te helpen scherper te redeneren en uw gedachten en ontwerpbeslissingen ondubbelzinnig vast te leggen. Het is echter mijn streven u niet alleen de zin maar ook de schoonheid van formele methoden te doen inzien.

Leden van de vakgroep Informatica. Wij staan voor de moeilijke taak een opkomende wetenschap te ontwikkelen en te onderwijzen in een maatschappij die grote behoefte heeft aan snelle vruchten van deze wetenschap, maar die er weinig begrip voor heeft wanneer te-vroeg geplukte vruchten soms zuur blijken te zijn. We moeten de ongekende mogelijkheden van de computer exploreren en in kaart brengen, en tegelijk methoden ontwikkelen en uitdragen om de complexiteit van deze maaksels te beheersen. Ik weet dat u voor deze taak allen uw best zult doen. Het is mij een eer thans uw voorzitter te mogen zijn.

Hooggeleerde Bron, Petkov, Spaanenburg en Nieuwenhuis. Beste Coen, Nikolay, Ben en Bart. Jullie houdt je

bezig met het ontwerp van programma's en systemen, waarbij correctheid één van de belangrijke factoren is. Samenwerking met jullie zal nodig zijn om de leer van de programmacorrectheid te doen groeien en vrucht te doen dragen.

Hooggeleerde Renardel, beste Gerard. Ik ben je bijzonder dankbaar voor de loyale wijze waarop je je jegens mij hebt opgesteld, vanaf het moment, nu drieeneenhalf jaar geleden, dat je als hoogleraar in de Logica en de Theoretische Informatica in Groningen werd aangesteld. Ik hoop, dat wij naast een vruchtbare collegiale en bestuurlijke samenwerking ook tot nauwere wetenschappelijke samenwerking op het gebied van specificatie en correctheid zullen kunnen komen.

Dames en Heren. Ik wil u tot slot allen uitnodigen voor de receptie hier beneden. Ik dank u voor uw aandacht.

Aantekeningen

(1) Wat is informatica? Het wordt heel kernachtig uitgedrukt in de titel van de intreerede [vdS85] van Van de Snepscheut: "Wij zijn ontwerpers". Maar dan rijst de vraag: "ontwerpers waarvan?" Het antwoord moet misschien luiden: "van alle dingen die je een computer kunt laten doen". U zou nu kunnen denken, dat ik de vraag probeer te ontwijken. Dat is echter niet zo. Het is juist één van de basisvragen van de informatica: wat kun je de computer allemaal laten doen?

Ik geef er echter de voorkeur aan ons vak niet zo direct door het apparaat te laten definiëren. Laat ik daarom de definitie geven, die Bron in zijn intreerede [Bro74] in Twente gegeven heeft: "Informatica is de verzameling kennis en kunde op het gebied van verwerven, vastleggen, opslaan, verwerken, en verschaffen van informatie, in het bijzonder met behulp van rekenautomaten." Deze definitie is nu (20 jaar later) nog even geldig als toen. Alleen het woord "rekenautomaat" doet hopeloos verouderd aan. De computer heeft zich in deze 20 jaar een vaste plaats in onze samenleving verworven.

(2) Drie jaar geleden stelde Baeten in zijn intreerede [Bae92] in Eindhoven: "dat de informatica nog een lange weg te gaan heeft, voordat het zich als volwaardige wetenschap kan presenteren." Inderdaad, de informatica staat voor de geweldige uitdaging om de complexe machines die we zelf ontwerpen beheersbaar te houden. Ik beschouw de informatica echter wel als een volwaardige wetenschap, want er zijn belangrijke vragen gesteld en voor belangrijke problemen zijn goede oplossingen of oplossingsmethoden ontwikkeld.

(3) Er zijn verschillende soorten van formele methoden voor programmacorrectheid. We gebruiken bij ons informaticaonderwijs de zogenaamde inductieve assertionele methode. Het gaat hierbij om asserties (beweringen) over de toestand van het programma, die inductief bewezen worden. De methode bestaat er dus uit, dat men elk punt van het programma van een bewering voorziet die op dat punt geldig is. Deze geldigheid wordt inductief bewezen, namelijk gebruik makend van de geldigheid van de beweringen op alle voorgaande punten. Een apart argument is nodig om eindiging van het programma aan te tonen.

Ik wil u hiervan een voorbeeld geven, een programma voor de berekening van x^n voor gehele $n \geq 0$. Zoals u weet, ontstaat x^n door het getal x n–keer met zichzelf te vermenigvuldigen. Het grootste gevaar in een programma hiervoor is nu, dat dit één keer te veel of te weinig gebeurt. Mijn programma gaat alsvolgt:

$k := 0$; $y := 1$;
zolang $k \neq n$ **doe**
($k := k + 1$; $y := y \cdot x$) .

Om te laten zien dat dit programma correct is, ga ik het annoteren. We krijgen aldus een programma met een bewijs volgens de inductieve assertionele methode:

$\{ 0 \leq n \}$
$k := 0$; $y := 1$; $\{ J : \quad k \leq n$ **en** $y = x^k \}$
zolang $k \neq n$ **doe**
($\{k + 1 \leq n$ **en** $y \cdot x = x^{k+1}\}$
$k := k + 1$; $y := y \cdot x$ $\{J\}$) ;
{ na afloop geldt J **en** $k = n$, en dus $y = x^n$ } .

In het begin weten we hier alleen $0 \leq n$. Als we in de eerste regel k en y de waarden 0 en 1 gegeven hebben, geldt de bewering J, want $x^0 = 1$ (dat is een afspraak). We zorgen er nu voor, dat de bewering J steeds weer geldig wordt, door zolang $k \neq n$ is, telkens k één te verhogen en y met x te vermenigvuldigen. Zo'n bewering J heet een invariant. Het programma eindigt, omdat k niet kleiner dan n kan blijven. Na afloop geldt de bewering J weer. Omdat dan tevens $k = n$ geldt, hebben we $y = x^n$, wat de bedoeling was.

Ik heb het u hierboven overigens verkeerd verteld. Het is niet realistisch om zomaar een programma op te schrijven en dan te denken dat je dat correct zou kunnen bewijzen. Het ontwerpen van een bewijsbaar correct programma gebeurt door het bewijs en het programma tegelijk te bedenken,– waarbij het bewijs vaak iets voorop loopt, omdat het een indicatie geeft hoe het programma verder moet.

De besproken bewijsmethode voor de herhaling is afkomstig van Hoare [Hoa69]. Dijkstra heeft in zijn boek [Dij76] uit 1976 de methodologie aangegeven om dit soort herhaalprogramma's tezamen met het bewijs uit de specificatie af te leiden. Deze me-

thodologie is vooral in Eindhoven verder ontwikkeld, en heeft zijn voorlopig hoogtepunt gevonden in Kaldewaij's boek [Kal90].

Analoge bewijsregels voor recursieve procedures zijn al door Hoare in 1971 gegeven (zie [Hoa71]). De situatie is hier echter ingewikkelder door het optreden van diverse soorten van parameters en door de scheiding tussen de declaratie van de procedure, waar hij gemaakt wordt, en zijn aanroep waar hij wordt toegepast. Ook het eindigingsargument ligt ingewikkelder. De goede bewijsregels zijn inmiddels wel duidelijk (zie b.v. [Hes93]), maar het is niet duidelijk of een echte methodologie voor het ontwerpen van recursieve procedures mogelijk is.

(4) Ook speelt een rol, dat de ingewikkelde concepten in de wiskunde door topspecialisten zijn ingevoerd, terwijl die programma's van ons vaak zo ingewikkeld zijn geworden, omdat de gebruikers meer wilden en de programmeurs geen tijd hadden om iets eenvoudigers te bedenken.

(5) Zie het rapport [Lan94]. Ik zou nog iets verder willen gaan. De leerlingen moeten niet alleen een goede redenering kunnen opzetten, maar daarbij ook onderscheid kunnen maken tussen wat gegeven is en wat als vanzelfsprekend aanvaard kan worden.

(6) Deze behoefte wordt heel duidelijk uitgesproken in Dijkstra's artikel [Dij68a] over het Eindhovense systeem voor multiprogramming. Hij spreekt daarin over zijn "bitter experience with programming errors" en zegt dan: "As a result I was terribly afraid", om tenslotte trots aan te kondigen: "We shall have proved the correctness of the system with a rigor and explicitness that is unusual for the great majority of mathematical proofs". Het is verrassend, dat Dijkstra zich toen reeds realiseerde, dat multiprogramming de inspiratiebron zou kunnen worden waar de correctheid van gewone programma's veel baat bij zou hebben. Hij besluit zijn andere baanbrekende artikel uit dat jaar ([Dij68b]) namelijk met de zin: "And may we not hope that a confrontation with the intricacies of Multiprogramming gives us a clearer understanding of what Uniprogramming is all about?" Deze hoop van Dijkstra is duidelijk uitgekomen.

Het gaat bij parallellisme om de werking van een stel programma's, die min of meer tegelijk uitgevoerd worden en die met elkaar communiceren door middel van boodschappen of een ge-

meenschappelijk geheugen. Men zou het geheel natuurlijk op dezelfde wijze kunnen specificeren als een gewoon programma, maar dan vergt het correctheidsbewijs van het geheel een analyse van alle programma's uit die verzameling. De enige manier om een groot probleem op te lossen, is het in stukjes te hakken. We hebben dus behoefte aan specificaties van de afzonderlijke programma's, zodanig dat daaruit kan worden afgeleid wat het gedrag is van het geheel bij parallelle verwerking. We moeten dus een manier hebben om programma's zo te specificeren, dat de specificaties samengesteld kunnen worden op een manier die overeenkomt met parallelle verwerking van de programma's. Men spreekt dan van compositionele specificaties. Er zijn inmiddels inderdaad specificatieformalismen voorgesteld die dit mogelijk maken (zie b.v. [PaJ91]), maar er is nog nauwelijks ervaring mee. Het staat dus nog te bezien of dit soort formalismen echt iets oplossen, of dat zij alleen de hoeveelheid te verrichten werk op een andere manier indelen.

Ik moet in dit verband ook een ander aspect van concurrency noemen, waarop Harel en Pnueli [HaP85] gewezen hebben. Parallelle verwerking is in wezen niet meer dan een implementatiedetail,– belangrijker is, dat de processen in een parallel programma doorgaans reactief zijn: dit wil zeggen, dat ze in beginsel oneindig lang kunnen doorgaan in een geregelde communicatie met gebruikers of andere processen. Dit heeft tengevolge, dat we het proces niet meer kunnen specificeren door de relatie tussen de begintoestand en de eindtoestand. Er is immers geen eindtoestand. Het gewenste gedrag bestaat uit communicaties die afhangen van de hele geschiedenis die het proces doorlopen heeft.

We kunnen b.v. ook het tekstverwerkingsprogramma beter als een reactief programma beschouwen dan als een transformationeel programma waarvan alleen begin– en eindtoestand van belang zijn. Dat we daar vroeger geen problemen mee hadden, komt omdat het tekstverwerkingsprogramma een relatief eenvoudig communicatiepatroon heeft.

(7) De wiskundige behandeling van nondeterminisme hoeft niet moeilijk te zijn. Men kan de verzameling mogelijke resultaten beschouwen als een functie van de invoer, of men kan het hebben over de binaire relatie tussen de invoer en de mogelijke resultaten. Deze beschouwingswijzen zijn wiskundig natuurlijk gelijkwaardig.

(8) Angeliek nondeterminisme speelt ook een rol in de complexiteitsleer, de theorie waarin men bepaalt hoeveel rekentijd of geheugenruimte de oplossing van een probleem vergt. In 1971 heeft Cook (in [Coo71]) een klasse van problemen aangegeven die met angeliek nondeterminisme efficiënt oplosbaar zijn, terwijl ze met een deterministisch programma vermoedelijk niet efficiënt oplosbaar zijn. Voor de insiders: ik doel hier op de NP-volledige problemen. Ik zei "vermoedelijk", want ondanks meer dan twintig jaren van vruchtbare theorievorming is het toen opgekomen vermoeden $\mathcal{P} \neq \mathcal{NP}$ nog steeds niet bewezen.

Ik wil u een voorbeeld geven van een NP-volledig probleem. Stel u voor, dat u een verzameling legpuzzelstukjes heeft. De vraag is te beslissen of de puzzelstukjes tot één rechthoek aaneen gelegd kunnen worden. Als u voldoende tijd en geduld heeft, zult u hier wel achter kunnen komen. Als u echter een beschermengel heeft, die de stukjes op nondeterministische wijze op de tafel legt en deze engel is u welgezind en legt de stukjes zo mogelijk in een rechthoek, dan kunt u snel nagaan, dat de puzzel inderdaad goed gelegd is. Dit is veel efficiënter. De nondeterministische keuze betreft hier het willekeurig neerleggen van de stukjes op de tafel. Angeliek nondeterminisme betreft de vraag of er tenminste één goede manier van leggen is. Demonisch nondeterminisme zou hier de vraag betreffen, of *elke* manier van leggen een rechthoek oplevert (en dat is uiterst onwaarschijnlijk). Het eerder genoemde vermoeden $\mathcal{P} \neq \mathcal{NP}$ houdt nu in, dat er geen computerprogramma kan zijn, dat zonder uitputtend zoeken nagaat of de puzzelstukjes in een rechthoek te leggen zijn. Dit vermoeden lijkt aannemelijk, maar is nog steeds niet bewezen (of weerlegd).

(9) Het is niet moeilijk een programma nondeterministisch te specificeren. Iets anders is precies te beschrijven hoe het nondeterminisme van programma-onderdelen doorwerkt in het gedrag van het programma als geheel. Dit is met name lastig, omdat het onderdeel misschien wel oneindig vaak wordt uitgevooerd. Bovendien willen we geen beschrijvingsmethode die van ons vraagt, dat we de berekening stap voor stap nadoen. Laat ik drie aspecten van nondeterminisme iets nader uitdiepen.

Oneindig nondeterminisme

De beschrijving hoe het nondeterminisme van onderdelen van een programma doorwerkt in het geheel, is extra lastig, als som-

mige stappen in de berekening "oneindig nondeterministisch" mogen zijn. Een programma-onderdeel heet oneindig nondeterministisch als bij één invoerwaarde oneindig veel verschillende resultaatwaarden mogelijk zijn. Een voorbeeld hiervan is:

`geef` y `een waarde` $> x$.

Dijkstra verbood in zijn boek [Dij76] oneindig nondeterminisme. Hij eiste dus, dat bij één invoerwaarde tenhoogste eindig veel verschillende resultaten horen. Eén van de redenen hiervoor was, dat zijn formele definitie van de herhaalopdracht anders niet overeenstemde met wat je operationeel zou verwachten. Een tweede reden was de overweging, dat een mechanisme, dat in eindige tijd een keuze kan maken uit oneindig veel mogelijkheden, niet implementeerbaar zou zijn ([Dij76], p. 77). We kunnen dit laatste bezwaar (in navolging van Park, [Par80], p. 514) als een drogreden ontmaskeren: het mechanisme is immers niet gedwongen alle mogelijkheden te beschouwen, het mag (b.v.) altijd de eerste mogelijkheid kiezen.

Ook De Bakker verbood oneindig nondeterminisme in zijn standaardwerk [Bak80]. Hij ging uit van deterministische programma's en liet in tweede instantie eindig nondeterminisme toe. Toen ik in 1985 van de wiskunde naar de informatica overstapte, was ik mij er niet van bewust hoe heet dit hangijzer was. Ik stuitte bij het geven van het college Voortgezet Programmeren op datastructuren met oneindig nondeterminisme. Hieruit groeide mijn eerste informatica-publicatie [Hes88a]. Geïnspireerd door werk van Dijkstra en Scholten [DiS84] bleek mij vervolgens, dat bij een goede benadering van recursieve procedures oneindig nondeterminisme geen probleem hoeft te zijn, zie [Hes88b].

De eerst–gepubliceerde behandeling van oneindig nondeterminisme is die van Apt en Plotkin [ApP86], maar ik moet erkennen, dat ik dat artikel nooit goed heb kunnen verwerken.

Refinement calculus en angeliek nondeterminisme

Hoe het ook zij, determinisme en eindig nondeterminisme kunnen we voortaan als speciale gevallen beschouwen en oneindig nondeterminisme mag gewoon toegestaan worden. Ik moet nu echter weer terugkomen op angeliek nondeterminisme, en wel in het kader van de vraag een methode te geven voor het maken van een programma dat aan een gegeven specificatie voldoet. Er zijn hier twee benaderingen, die elkaar uiteindelijk ongetwijfeld zullen moe-

ten aanvullen. Allereerst is er de afleidingsmethode voor herhaalprogramma's die ik eerder noemde. Ik wil hier echter ingaan op de andere methode, die bekend staat onder de naam "refinement calculus".

De refinement calculus begint, voorzover ik weet, met het proefschrift van Ralph Back [Bac78]. Het idee van de refinement calculus is dat programma's en specificaties eigenlijk dezelfde soort dingen zijn. Preciezer: een programma is een specificatie, die ook nog door een computer uitgevoerd kan worden. Het ontwerp van een programma volgens de refinement methode gaat nu als volgt. We beginnen met een specificatie en gebruiken regels om specificaties stapsgewijs te verfijnen in de hoop, dat alle componenten uiteindelijk door een computer uitgevoerd kunnen worden (dat is: programma's zijn). Eén van de eerste taken is nu het vinden van goede regels om specificaties uit andere specificaties op te bouwen. Dit project van Back is een aantal jaren blijven liggen. Rond 1988 komt er weer schot in, en in 1990 stellen Back en Von Wright [BaW90] en Morgan [Mor90] onafhankelijk van elkaar een angelieke nondeterministische keuze voor, als één van de operatoren om specificaties samen te stellen.

Deze operator is slechts een hulpmiddel bij het specificeren van programma's. We spreken als het ware af, dat de machine de gewenste keuze zal maken, en bekommeren er ons vooralsnog niet om hoe de machine tot deze keuze komt.

De genoemde auteurs voerden angeliek nondeterminisme in met behulp van Dijkstra's predicaat-transformers. Het is inderdaad niet moeilijk een formele semantiek langs die lijnen te ontwikkelen.

Intuïtief gesproken vergt angeliek nondeterminisme een beschermengel die weet welke postconditie de gebruiker nastreeft. Dit is uiteraard niet op een echt computer systeem te realiseren (al zouden we dat misschien willen). Het is dus bepaald niet van te voren duidelijk, dat men aan angeliek nondeterministische programma's ook een operationele betekenis kan toekennen. Toch is het mij gelukt (in [Hes94a]) een operationele definitie te geven van de betekenis van programma's waarin angeliek nondeterminisme en demonisch nondeterminisme naast elkaar voorkomen. Deze betekenis is gebaseerd op de vraag of de engel een winnende strategie heeft in een bepaald formeel spel tegen de demon,– een spel dat

bepaald wordt door het programma en de nagestreefde postconditie.

Fairness

Ik heb u in deze bespreking van nondeterminisme tot nu toe alleen geconfronteerd met de twee uiterste mogelijkheden: met demonisch nondeterminisme waarin we aan het systeem een willekeurige keuze toelaten, en met angeliek nondeterminisme waarin we van het systeem een keuze verwachten die aan onze onuitgesproken wensen voldoet. Er is echter ook een belangrijke tussenvorm: fair nondeterminisme. Dit betreft programma's waarin we aan het systeem een onbegrensd aantal keren dezelfde keuze voorleggen. Het is maar een kleine variatie op demonisch nondeterminisme, in de zin dat het systeem elke keer een willekeurige keuze mag maken. De eis van fairness houdt alleen in, dat als er oneindig vaak gekozen kan worden, het systeem beide alternatieven ook oneindig vaak kiest. Het standaardvoorbeeld is het programma:

$x := 0$; $y := 1$;
zolang $x = 0$ **doe** ($y := y + 1$ **of** $x := 1$) .

Het woord "**of**" geeft het systeem een nondeterministische keuze. Als het systeem *altijd* de eerste mogelijkheid kiest, wordt y steeds groter gemaakt, terwijl $x = 0$ blijft; het programma eindigt dan niet. Deze berekening is echter niet *fair*, omdat de keuze oneindig vaak gemaakt wordt en de tweede mogelijkheid nooit wordt gekozen. Als de tweede mogelijkheid gekozen wordt, krijgt x de waarde 1, waarna het programma eindigt. Dit impliceert, dat het programma onder fair nondeterminisme gegarandeerd eindigt. Zoals u kunt zien heeft y dan een onbekende positieve waarde. Fair nondeterminisme impliceert hier dus tevens oneindig nondeterminisme.

Fair nondeterminisme wordt b.v. gebruikt bij het beschrijven van de communicatie tussen twee computers over een onbetrouwbaar kanaal. We kunnen dan afspreken, dat de zendende computer zijn boodschap herhaalt totdat hij een bevestiging van correcte onvangst van de andere gekregen heeft. Het kanaal maakt telkens een demonische keuze, of het de boodschap goed of fout doorgeeft. Fairness houdt dan in, dat de boodschap ooit wel eens goed overkomt. Dit is dan een aanname waar de electrotechnische ingenieurs voor moeten zorgen. De informaticus gebruikt deze

aanname om de samenwerking tussen de twee computers verder te analyseren.

Er zijn in feite meerdere soorten van fairness. Er is zelfs een heel boek aan dit begrip gewijd ([Fra86]). Fair nondeterminisme ligt vlak tegen demonisch nondeterminisme aan en vergt alleen bij oneindige herhaling telkens even een angelieke keuze. Mijn leerling Rutger Dijkstra heeft dit idee vorig jaar geformaliseerd voor een elegante behandeling [RDi95] van de fairness van de ontwerptaal UNITY ([ChM88]).

(10) Hartmanis [Har93]: "Clearly, computer science is not a physical science; still, very often it is assumed that it will show strong similarities to physical sciences and may have similar research paradigms in regard to theory and experiments. (...) We will argue that that is not the case and that theory and experiment in computer science play a considerably different role."

(11) Laat ik nog een falsifieerbare hypothese noemen. De zogenaamde these van Church–Turing komt erop neer, dat elke berekenbare functie berekend kan worden door een voldoend grote computer. Dit zou in beginsel gefalsifieerd kunnen worden; echter niet door een experiment maar door de invoering van een ruimer begrip van berekening. Wij hebben vooralsnog echter geen enkele aanwijzing, dat de these onjuist zou zijn. Als u een berekenbare functie hebt, die niet door een computer berekend kan worden, ligt dat misschien aan de capaciteit van de computer, maar anders vrijwel zeker aan de capaciteiten van de programmeur.

(12) De bespreking van mogelijk verwante disciplines is hiermee bepaald niet uitgeput. Er is alle reden informatica te vergelijken met informatietheorie, electrotechniek, bedrijfskunde of rechten.

Informatietheorie is een onderdeel van de wiskunde waarin begrippen als informatiedichtheid en redundantie statistisch onderzocht worden. Het gaat hierbij niet om de betekenis van de informatie. Deze wetenschap is vooral van belang bij het bestuderen en ontwerpen van betrouwbare en voldoend grote fysische informatiedragers. De relatie met informatica is vooral gelegen in gemeenschappelijke toepassingen.

De electrotechniek kan als de moeder van de informatica gezien worden. De informatica is immers groot geworden door de mogelijkheden die door de electronica geboden worden. Er is ook

een duidelijke vakinhoudelijke aansluiting. De ontwerpmethoden voor programma's en digitale hardware schijnen b.v. niet echt te verschillen ...

De relatie met bedrijfskunde loopt niet alleen via bedrijfskundige toepassingen van de informatica maar ook via het gemeenschappelijke probleem complexe informatiesystemen doeltreffend te beheersen. Ditzelfde geldt ook voor de relatie met rechtswetenschappen. Men kan programmeurs zeer wel met wetgevers vergelijken.

Referenties

[ApP86] K.R. Apt, G.D. Plotkin: Countable nondeterminism and random assignment. J. ACM **33** (1986) 724–767.

[Bac78] R.J.R. Back: On the correctness of refinement in program development. Ph.D. thesis, Report A-1978-4, Department of Computer Science, University of Helsinki, 1978.

[BaW90] R.J.R. Back, J. von Wright: Refinement calculus, Part I: Sequential Nondeterministic Programs. In: J.W. de Bakker, W.–P. de Roever, G. Rozenberg (eds.) Stepwise Refinement of Distributed Systems. Lecture Notes in Computer Science 430 (Springer, Berlin, 1990) pp. 42–66.

[Bae92] J.C.M. Baeten: Informatica als wetenschap, formele specificatie en wiskundige verificatie. Intreerede, Eindhoven 1992.

[Bak80] J.W. de Bakker: Mathematical Theory of Program Correctness. Prentice–Hall, 1980.

[BoM88] R.S. Boyer, J S. Moore: A Computational Logic Handbook. Academic Press, Boston etc., 1988.

[Bro74] C. Bron: Speelgoed of werktuig. Inaugurale Rede, Twente 1974.

[ChM88] K.M. Chandy, J. Misra: Parallel Program Design, A Foundation. Addison–Wesley, 1988.

[Coo71] S.A. Cook: The complexity of theorem proving procedures. Proc. Third Annual ACM Symposium on the Theory of Computing, pp. 151–158.

[DDH72] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare: Structured Programming. Academic Press, 1972.

[Dij68a] E.W. Dijkstra: The structure of the THE multiprogramming system. Comm. ACM **11** (1968) 341–346.

[Dij68b] E.W. Dijkstra: Co-operating sequential processes. In: F. Genuys (ed.): Programming Languages (NATO Advanced Study Institute). Academic Press, London etc. 1968, pp. 43–112.

[Dij76] E.W. Dijkstra: A Discipline of Programming. Prentice–Hall 1976.

[DiS84] E.W. Dijkstra, C.S. Scholten: The operational interpretation of extreme solutions. Tech. Rept. EWD 883, 1984.

[RDi95] R.M. Dijkstra: DUALITY: a simple formalism for the analysis of UNITY. Verschijnt in Formal Aspects of Computing.

[Fra86] N. Francez: Fairness. Springer V, 1986.

[Flo67] R.W. Floyd: Assigning meanings to programs. In: Proceedings of the Symposium on Applied Math., Vol. 19, pages 19–32, AMS, 1967.

[Gib94] W.W. Gibbs: Software's chronic crisis. Scientific American, Sept. 1994, 72–81.

[HaP85] D. Harel, A. Pnueli: On the development of reactive systems. In: K.R. Apt (ed.): Logics and models of concurrent systems, volume F13 of NATO ASI Series, pp. 477–498. Springer V., 1985.

[Har93] J. Hartmanis: Some observations about the nature of computer science. In: R.K. Shyamasundar (ed.): Foundations of software technology and theoretical computer sciencee. Springer V. (Lect. Notes in Computer Science 761) 1993, pp. 1–12 (ook Bull. EATCS 53 (1994) 170–190)

[Her91] M.P. Herlihy: Wait–free synchronization. ACM Trans. on Program. Languages and Systems **13** (1991) 124–149.

[Hes88a] W.H. Hesselink: A mathematical approach to nondeterminism in data types. ACM Transactions on Programming Languages and Systems **10** (1988), 87–117.

[Hes88b] W.H. Hesselink: Interpretations of recursion under unbounded nondeterminacy. Theoretical Computer Science **59** (1988) 211–234.

[Hes93] W.H. Hesselink: Proof rules for recursive procedures. Formal Aspects of Computing **5** (1993) 554–570.

[Hes94a] W.H. Hesselink: Nondeterminacy and recursion via stacks and games. Theoretical Computer Science **124** (1994) 273–295.

[Hes94b] W.H. Hesselink: Wait-free linearization with an assertional proof. Distributed Computing **8** (1994) 65–80.

[Hes95] W.H. Hesselink: Wait-free linearization with a mechanical proof. Verschijnt in Distributed Computing.

[Hoa69] C.A.R. Hoare: An axiomatic basis for computer programming. Com. ACM **12** (1969), 576–583.

[Hoa71] C.A.R. Hoare: Procedures and parameters: an axiomatic approach. In: Symposium on Semantics of Algorithmic Languages. (ed. E. Engeler), Springer V. (Lecture Notes in Math. 188) 1971, pp. 102–116.

[Kal90] A. Kaldewaij: Programming: the Derivation of Algrithms. Prentice Hall International, 1990.

[Lan94] J. de Lange, e.a.: Rapport van de Studiecommissie Wiskunde B VWO. Utrecht 1994.

[Mor90] C. Morgan: Programming from Specifications. Prentice Hall, Englewood Cliffs, NJ, 1990.

[Nau66] P. Naur: Proof of algorithms by general snapshots. BIT **6** (1966), 310–316

[PaJ91] P.K. Pandya, M. Joseph: P-A logic – a compositional proof system for distributed programs. Distr. Computing **5** (1991) 37–54.

[Par80] D. Park: On the semanics of fair parallelism, in: Abstract Software Specifications, Lecture Notes in Computer Science **86** (Springer, Berlin, 1980) 504–526.

[RaS59] M.O. Rabin, D. Scott: Finite automata and their decision problems. IBM J. Res. **3**:2 (1959) 115–125.

[vdS85] J.L.A. van de Snepscheut: Wij zijn ontwerpers. Inaugurale Rede, Groningen 1985

[Tol95] R.M. Tol: Formal Design of a real-time operating system kernel. Proefschrift (verwacht), Groningen 1995.

[Too57] M. Toonder: De Klonters. Bezige Bij, 1957.

[Tur49] A.M. Turing: On checking a large routine. In Report of a Conference on High–Speed Automatic Calculating Machines, University Mathematical Laboratory, Cambridge, 1949, pp. 67–69.