

Demonisch en angeliek nondeterminisme: keuze en aanbod

Een tekst voor het Jongerejaarscolloquium van 11 januari 2007
Wim H. Hesselink

1 Mijn Onderzoek

Ik ben in de wiskunde gepromoveerd, en heb daarna nog meer dan 5 jaar wiskundig onderzoek gedaan. In 1982 had ik voor mijn wiskundig onderzoek behoefte aan wat tegenwoordig computeralgebra zou heten. Dat bestond toen nog niet, dus ik heb gewoon leren programmeren. Van het een kwam het ander en rond 1985 ben ik naar de Informatica overgestapt.

Mijn uitgangspunt was, dat ik geen wiskunde in de informatica zou gaan gebruiken die niet echt nuttig was. Ik heb me daar aan gehouden. Ik heb me in eerste instantie bezig gehouden met het onderzoeken van de betekenis van gewone sequentiële programma's, en met de regels die gebruikt kunnen worden om de correctheid daarvan uit te drukken en te bewijzen. Dit valt onder de vakgebieden: semantiek van programmeertalen, programmeermethodologie en programmacorrectheid. Ik heb dit rond 1992 afgerond door er een boek [Hes92] over te schrijven.

Ik ben daarna vooral bezig geweest met de correctheid van concurrent programma's. Concurrency is het verschijnsel dat verschillende processen, processoren of computers gelijktijdig rekenen en communiceren en daarmee de globale toestand van het systeem veranderen. Het gaat hier om een spectrum van mogelijkheden. Het varieert van multithreaded programma's voor een specifiek doel tot communicatieprotocollen.

Doordat de executievolgorde van de acties van de verschillende processen onvoorspelbaar is, is het lastig er goed over te redeneren, terwijl het vaak vrijwel onmogelijk is om zulke systemen uitputtend te testen. Er zijn drie soorten van onderzoeksprojecten in dit gebied. Soms proberen we de correctheid van een gegeven concurrent programma aan te tonen (dit komt soms neer op het aantonen van de incorrectheid, hoewel het meestal meevalt). Soms proberen we bij een gegeven probleem een bewezen correct algoritme te ontwerpen. En er zijn ook projecten, waarin gestreefd wordt naar een algemene ontwerpmethodologie die altijd bewezen correcte oplossingen oplevert. Anderhalf jaar geleden is Gao Hui bij mij gepromoveerd op een proefschrift [Gao05] met een paar voorbeelden van het tweede type. We gebruiken stellingbewijzers om de correctheid van specifieke programma's aan te tonen, en model checkers om de incorrectheid aan te tonen. Gao Hui heeft trouwens alleen stellingbewijzers gebruikt.

Ik houd me daarnaast bezig met programma's voor beeldbewerking, samen met Meijster, Roerdink en Wilkinson (bv. [MRH00]) en met een grafentheorie voor onverzadigde polycyclische koolwaterstoffen, samen met Renardel en enkele natuur- en scheikundigen.

Ik wil het nu verder hebben over nondeterminisme. Toen ik naar Informatica overstapte, was dit het eerste onderwerp waar ik onderzoek in ben gaan doen. Het speelt ook een belangrijke rol bij concurrency, dus het is me blijven bezig houden.

2 Nondeterminisme

Als er in de studiegids staat dat de student Vectorcalculus moet doen of Wiskunde B, dan kan dat twee dingen betekenen

1. Beide vakken worden aangeboden en de student mag kiezen.

2. De staf ziet zich gedwongen slechts één van de twee vakken aan te bieden (er is nog niet beslist welk), en de student zal dat vak voor lief moeten nemen.

Het gaat in beide gevallen om meerdere alternatieven, dus om nondeterminisme. In het ene geval mag de afnemer of gebruiker (hier de student) kiezen, in het andere geval de aanbieder of het systeem (hier de staf). Nondeterminisme in het “voordeel” van de afnemer heet *angeliek nondeterminisme*. Nondeterminisme in het “voordeel” van de aanbieder heet *demonisch nondeterminisme*.

In Nederland is het aanvragen van een kenteken een kwestie van demonisch nondeterminisme: het systeem kiest, de gebruiker heeft geen invloed (hetzelfde geldt dikwijls voor hotelkamers). In de informatica komen beide vormen van nondeterminisme voor.

3 Nondeterminisme in de Informatica

Onder nondeterminisme verstaat men in de informatica het verschijnsel dat een berekening bij één en dezelfde invoer verschillende resultaten kan opleveren. Dit lijkt op het eerste gezicht ongewenst. Bij nadere beschouwing blijkt echter, dat nondeterminisme noodzakelijk is als we programma's of systemen willen specificeren, ontwerpen of analyseren zonder te verdinken in een zee van irrelevante details. Eén voorbeeld: wanneer je als gebruiker van een computersysteem een bestand naar een schijf wegschrijft, wil je er doorgaans geen weet van hebben, waar het bestand op de schijf terecht komt. Voor een enigszins abstracte specificatie van de schrijfpdracht 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 duidelijk worden. In dat geval doet de ontwerper er goed aan een nondeterministisch keuze te gebruiken, die in een later stadium misschien weer door een deterministische keuze vervangen wordt.

Nondeterminisme is dus het verschijnsel, dat bij gegeven beginconditie verschillende eindtoestanden mogelijk zijn. Laten we nu aannemen, dat de programmeur een zeker doel met zijn programma heeft, dwz. een beoogde postconditie. Door het nondeterminisme komen we dan voor de vraag te staan, of alle mogelijke eindtoestanden aan dit doel beantwoorden (*demonisch*), of alleen sommige (*angeliek*).

4 Demonisch nondeterminisme

Als de programmeur de taak heeft te zorgen dat *alle* mogelijke eindtoestanden aan de beoogde postconditie 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.

Het sequentiële deel van gewone imperatieve programmeertalen als Pascal of Java bevat alleen demonisch nondeterminisme, en dan nog alleen in ongeïnitieerde variabelen en onbekende pointerwaarden. Specificatietalen moeten *wel* in staat zijn demonisch nondeterminisme uit te drukken, dwz. uit te drukken, dat een bepaalde opdracht een zekere conditie waar maakt, terwijl andere effecten onbepaald zijn.

Demonisch nondeterminisme voor sequentiële programma's is ingevoerd door Dijkstra in 1975 in [Dij75]. Het nondeterminisme wordt daar gerechtvaardigd door de behoefte aan een symmetrische oplossing van het probleem om het maximum van x en y te bepalen:

```

if  $x \geq y$   $\rightarrow$   $m := x$ 
 $\parallel$   $y \geq x$   $\rightarrow$   $m := y$ 
fi .

```

Een uitgebreider voorbeeld: het demonisch binair zoeken van een stijgend punt. Gegeven is een array van getallen $a[0 \dots n]$ met $0 < n$ en $a[0] < a[n]$. Gevraagd: een algoritme ter bepaling van een index i met $0 \leq i < n$ en $a[i] < a[i+1]$. Laat daarbij zo veel mogelijk vrijheid aan het systeem (demonisch nondeterminisme).

```

(1)   int  $p := 0, q := n$  ; // invariant  $0 \leq p < q \leq n \wedge a[p] < a[q]$ 
       while  $p + 1 \neq q$  do
         kies int  $m$  met  $p < m < q$  ; // demonische keuze
         if  $a[p] < a[m]$   $\rightarrow$   $q := m$  // demonische keuze
          $\parallel$   $a[m] < a[q]$   $\rightarrow$   $p := m$ 
         fi
       end .

```

Ga na, dat in beide gevallen de keuze niet leeg is!

Deze lus eindigt het snelste, als we m steeds ongeveer halverwege tussen p en q kiezen, maar voor de correctheid maak het niet uit. Waarom willen we dit? Zoals gezegd: om de ontwerpbeslissingen zo lang mogelijk uit te stellen, bv. omdat we in de praktijk willen testen welke oplossing beter voldoet. Het is dan nuttig om de implementator nog zo veel mogelijk vrijheid te laten.

We kunnen het programma (1) dus opvatten als een *specificatie* waaraan verschillende programma's kunnen voldoen. Twee verschillende invullingen voor de body van de lus (0) zijn bv.

```

(a)   int  $m := p + 1$  ;
       if  $a[p] < a[m]$  then  $q := m$  else  $p := m$  end .

(b)   int  $m := (2 \cdot p + q + 1) / 3$  ; // integer deling
       if  $a[m] < a[q]$  then  $p := m$  else  $q := m$  end .

```

Dit is het idee van “stepwise refinement” van programma's, zie [Dij70,Wir71].

Vraag uit het publiek: “Waar is die +1 in het geval (b) voor nodig?”

Antwoord: “Anders is het geen verfijning en hoeft de lus niet te eindigen.”

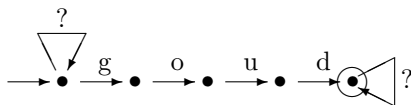
5 Angeliek nondeterminisme

Een geheel andere vorm is angeliek nondeterminisme. Dit drukt uit, dat tenminste één van de eindtoestanden aan de beoogde postconditie voldoet. We laten de 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.

Angeliek nondeterminisme heeft –vreemd genoeg– de oudste rechten. Het is het nondeterminisme van de nondeterministisch automaten, ingevoerd in 1959 door Rabin en Scott [RS59]. Ook het nondeterminisme in NP-volledige problemen is angeliek. NP volledigheid komt van Cook [Coo71].

Een nondeterministische eindige automaat is een gerichte graaf met één startknoop, waarvan de pijlen gelabeld zijn met bv. characters en sommige knopen omcirkeld zijn. De automaat *accepteert* een string met de achtereenvolgende characters a_1, \dots, a_n als er een gericht pad in de graph is die bij de startknoop begint, waarvan de stappen achtereenvolgens gelabeld zijn met a_1, \dots, a_n , en waarvan de laatste knoop omcirkeld is. Het label “?” kan elk character matchen.

Example. De hier getekende automaat accepteert elke string die de deelstring *goud* bevat.



Probeer nu de string voor *geen gouden stuiver*. Het nondeterminisme is hier angeliek: bij de *g* van *geen* moet de automaat bij de startknoop blijven, terwijl hij bij de *g* van *goud* naar rechts moet. De automaat accepteert de string *gebouwd* niet, want na de *g* kan hij niets met *eb*. Hij moet dus bij de startknoop blijven, en bereikt dus nooit de omcirkelde knoop. \square

6 Begrensd of onbegrensd nondeterminisme

Demonisch nondeterminisme ontstond in de praktijk, toen men rond 1965 op één computer meerdere processen naast elkaar begon te laten lopen. Het is dan niet van te voren duidelijk welk proces het eerst aan de beurt komt en dus ook niet in welke toestand dit proces het systeem aantreft. Dijkstra was daar vanaf het begin bij betrokken, vandaar dat hij nondeterminisme (voor zover ik weet) altijd demonisch heeft opgevat. Dijkstra beperkt zich in zijn boek “A discipline of programming” [Dij76] uitdrukkelijk tot begrensd nondeterminisme. Dit betekent dat het systeem op elk moment slechts kan kiezen uit een eindig aantal mogelijkheden (hij is hier later op teruggekomen [DS84]). Begrensd nondeterminisme is ook het uitgangspunt van het boek [dB80].

Ik denk, dat Park [Par80] in 1980 de eerste is, die demonische keuzes uit oneindig veel alternatieven toestaat. Hij beschouwt het parallelle programma:

$$(2) \quad \begin{array}{l} x := 0 ; y := 1 ; \\ (x := 1 \quad || \quad \mathbf{while} \ x = 0 \ \mathbf{do} \ y := y + 1 \ \mathbf{end}) . \end{array}$$

De twee verticale strepen betekenen parallelle compositie. Het linker en rechter deelprogramma worden uitgevoerd door verschillende threads.

Park heeft het hier over fair (eerlijk) parallelisme en eist dus, dat de toekenning $x := 1$ ooit uitgevoerd wordt. Omdat we niet kunnen weten wanneer dit gebeurt, is het aantal mogelijke eindwaarden voor y oneindig (we werken hier op een computer met willekeurig grote integers). Het programma is dus equivalent met

$$(3) \quad x := 1 ; y := \mathit{anyposint} .$$

Park legt vervolgens uit (blz 514), dat het nondeterminisme in (3) “loose” (vrijzinnig) bedoeld is: het is niet van belang of de implementatie meer dan een resultaat kan opleveren. De enige beperking is, dat elk opgeleverde resultaat aan de specificatie voldoet. Elke implementatie van $y := 1$ is dus ook een implementatie van $y := \mathit{anyposint}$. Programma (3) is een vorm van onbegrensd demonisch nondeterminisme.

7 Het modelleren van nondeterminisme

Het basisidee voor het modelleren van nondeterminisme is het gebruik van relaties. We beschouwen een verzameling X van alle mogelijke begintoestanden en een verzameling Y van alle mogelijke eindtoestanden. Een nondeterministisch commando wordt dan gemodelleerd als een verzameling R van paren (x, y) , waarbij $(x, y) \in R$

betekent dat het commando aangeroepen in een toestand x de eindtoestand y mag/kan hebben. Let wel: dit is een kwestie van specificatie.

Laten we bv. het programma beschouwen:

```
(4)   real x ;
       if x ≤ 6 → x := x + 1
       || x ≥ 1 → x := x - 1
       fi .
```

De bijbehorende relatie is de verzameling paren:

$$R = \{(x, y) \mid (x \leq 6 \wedge y = x + 1) \vee (x \geq 1 \wedge y = x - 1)\} .$$

We kunnen daar gemakkelijk een grafiek van maken; het is een “functie” die op het segment $[1, 6]$ tweewaardig is.

De relationele beschrijving zegt echter nog niet, of we het determinisme demonisch of angeliek opvatten. We kunnen dat wel uitdrukken met Hoare tripels [Hoa69]. Neem aan, dat we de postconditie $x < 4$ willen hebben. Wat is dan de benodigde preconditione bij het demonische commando dat hoort bij de relatie R ? En van het angelieke commando?

$$\begin{aligned} \{x < 3\} \text{ dem.} R \{x < 4\} & ; \\ \{x < 5\} \text{ ang.} R \{x < 4\} & . \end{aligned}$$

Het programma (4) staat overigens voor het demonische commando. De operator \parallel wordt namelijk gebruikt voor de demonische nondeterministische keuze. Ik gebruik \diamond voor de angelieke nondeterministische keuze.

Dijkstra [Dij75] heeft *weakest preconditions* ingevoerd om de bepaling van Hoare tripels eenvoudiger te maken. We hebben dat vroeger bij het vak programmacorrectheid onderwezen, maar het is eruit gesloopt, omdat het voor de programmeerpraktijk niet echt nodig was.

Voor een commando S en een postconditie Q is de zwakste preconditione $wp.S.Q$ is het zwakste predicaat P dat voldoet aan $\{P\} S \{Q\}$. Er geldt dus

$$\{P\} S \{Q\} \equiv [P \Rightarrow wp.S.Q] .$$

In feite kiezen we bij Hoare tripels de preconditione meestal zo zwak mogelijk, we kiezen dus P bij voorkeur gelijk aan $wp.S.Q$.

De regels voor de zwakste preconditiones voor demonische en angelieke keuze van commando's S en T zijn nu:

$$\begin{aligned} wp.(S \parallel T).Q & = wp.S.Q \wedge wp.T.Q , \\ wp.(S \diamond T).Q & = wp.S.Q \vee wp.T.Q . \end{aligned}$$

Je kunt dit als volgt begrijpen. Bij de demonische keuze weten we niet of S of T wordt uitgevoerd. Om zeker te zijn van de postconditie Q , moeten we dus wel zorgen dat aan beide preconditiones voldaan wordt. Bij de angelieke keuze weten we dat de engel de postconditie zo mogelijk zal bewerkstelligen, dus het is voldoende als aan één van beide preconditiones voldaan wordt (de engel heeft daar genoeg aan).

Het commando $E = (x := 0 \diamond x := 1)$ voldoet bv. aan

$$\begin{aligned} wp.E.(x = 0) & = true , \\ wp.E.(x = 1) & = true , \\ wp.E.(x = 0 \wedge x = 1) & = false . \end{aligned}$$

Desgewenst kan de engel ervoor zorgen, dat $x = 0$ wordt of dat $x = 1$ wordt, maar je moet niet het onmogelijke van hem verwachten.

De beste voorbeelden van angeliek nondeterminisme zijn te vinden bij het afleiden van parseeralgoritmen, maar dat zou hier te ver voeren.

Hoewel we natuurlijk tevreden kunnen zijn met de wp 's of de Hoare tripels, kunnen we ook zoeken naar een wat directere modellering waarin we de eindtoestand van zo'n angeliek of demonisch commando opvatten als een soort virtuele toestand (vergelijk de complexe getallen in de wiskunde).

Ik ben hier de afgelopen weken mee bezig geweest, vandaar dat ik voor dit praatje het onderwerp nondeterminisme heb gekozen. Het blijkt inderdaad mogelijk te zijn. Het commando E hierboven heeft dan een eindtoestand waarin $x = (0 \diamond 1)$ geldt, en de eindtoestand van het commando $(x := 0 \parallel x := 1)$ voldoet aan $x = (0 \parallel 1)$. Ik kan dit hier verder nog niet toelichten.

Referenties

- [Coo71] S.A. Cook. The complexity of theorem proving procedures. In *Proc. Third Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.
- [dB80] J.W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall Int., Englewood Cliffs, 1980.
- [Dij70] E.W. Dijkstra. Notes on structured programming. Tech. Rept., Tech. Univ. Eindhoven, EWD 249, see www.cs.utexas.edu/users/EWD, April 1970.
- [Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, 1975.
- [Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [DS84] E.W. Dijkstra and C.S. Scholten. The operational interpretation of extreme solutions. Tech. Rept., Tech. Univ. Eindhoven, EWD 883, see www.cs.utexas.edu/users/EWD, 1984.
- [Gao05] Hui Gao. *Design and verification of lock-free parallel algorithms*. PhD thesis, University of Groningen, April 2005.
- [Hes92] W.H. Hesselink. *Programs, Recursion and Unbounded Choice, predicate transformation semantics and transformation rules*. Cambridge University Press, Cambridge, 1992. (Cambridge Tracts in Theoretical Computer Science 27).
- [Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–583, 1969.
- [MRH00] A. Meijster, J.B.T.M. Roerdink, and W.H. Hesselink. A general algorithm for computing distance transforms in linear time. In J. Goutsias, L. Vincent, and D.S. Bloomberg, editors, *Mathematical morphology and its applications to image and signal processing (Proc. 5th Int. Conf.)*, pages 331–340. Kluwer, 2000.
- [Par80] D. Park. On the semantics of fair parallelism. In *Abstract Software Specifications*, volume 86 of *LNCS*, pages 504–526. Springer, 1980.
- [RS59] M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res.*, 3:115–125, 1959.
- [Wir71] N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14:221–227, 1971.