Programmacorrectheid

Wim H. Hesselink, 24 november 2006
http://www.cs.rug.nl/~wim

1 Inleiding

Bij Programmacorrectheid gaat het om de ontwikkeling en het gebruik van formele methoden om de correctheid van computerprogramma's te bevorderen of vast te stellen. Het vakgebied wordt daarom ook aangeduid als "Formal Methods".

Het doel van dit verhaal is jullie de plaats en betekenis van programmacorrectheid binnen de informatica te schetsen, en je te laten zien dat dit een uitdagend en nuttig vakgebied is. Ik geef een aantal verwijzingen naar de oude wetenschappelijke literatuur waarin je kunt zien hoe de groten in ons vakgebied vroeger geworsteld hebben om inzichten te verwerven, die wij tegenwoordig aan beginnende studenten kunnen uitleggen. Recentere verwijzingen over het belang van dit gebied zijn bv. [Woo06,Vaa06]. Literatuurverwijzingen staan aan het einde van dit stuk.

Wat betekent de correctheid van een computerprogramma? We kunnen een computerprogramma opvatten als een opdracht aan de computer. Of de opdracht correct gesteld is en begrijpelijk voor de computer, dat kan de compiler zelf wel controleren. Hier gaat het dus om iets anders.

Het programma is namelijk niet zo maar een opdracht, nee, het is een opdracht met een *doel*. Het doel van het programma wordt weergegeven in de *specificatie*.

Een programma is correct als het voldoet aan zijn specificatie.

Het zou vanzelfsprekend moeten zijn, dat het programma aan zijn specificatie voldoet. Net 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 bepaald niet altijd waar is. Mensen maken fouten, of het nu programmeurs zijn of kassières.

Er zijn wel verschillen. Bij het bedienen van de kassa moet je telkens opnieuw één klant bedienen, de boodschappen aanslaan en de kosten correct verrekenen. De transacties met verschillende klanten zijn onderling onafhankelijk (als we afzien van de klant die later terug komt met de klacht dat hij te weinig geld heeft teruggekregen). De programmeur echter werkt aan een programma dat misschien 1000, misschien 100000 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, het klinkt zo eenvoudig, dat het programma aan zijn specificatie moet voldoen. Hier zitten 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.

Het is dus wel begrijpelijk dat programmeurs fouten maken. Er zijn tal van afschrikwekkende verhalen over feitelijk opgetreden fouten in computerprogramma's (b.v. de mislukte eerste lancering van de Ariane 5 en de floating point operaties van de Pentium 3). Ik pleeg dergelijke verhalen zo vlug mogelijk te vergeten. Eén verhaal wil ik jullie niet onthouden.

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.

Opgave 1. Software projecten kunnen op allerlei wijzen falen: ze komen soms nooit of veel te laat klaar, of er zijn budgetoverschrijdingen, of het product voldoet niet aan de verwachtingen, of de afgeleverde software vertoont incidenteel dramatische fouten. Rond 1970 sprak men van een software crisis. Deze bestond in 1994 nog steeds [Gib94]. Onderzoek mbv. het internet of de situatie sindsdien verbeterd is. Vermeld je bronnen en maak een inschatting van hun betrouwbaarheid.

2 Geschiedenis

Na een eerste aanzet van Turing [Tur49] is de aandacht voor programmacorrectheid pas goed opgekomen aan het eind van de zestiger jaren, bv. [Nau66]. In de zelfde tijd kwam ook het idee op van multiprogramming: dit is het ontwerpen van een verzameling programma's met één gemeenschappelijk geheugen, die min of meer gelijktijdig op één computer verwerkt worden. Dit is het begin van de huidige vakgebieden *operating systems*, *parallel systems* en *thread programming*. De nederlander Dijkstra (1930-2002) heeft hierbij een belangrijke rol gespeeld, bv. [Dij68].

Het is geen toeval, dat programmacorrectheid en multiprogramming tegelijk opkwamen. De correctheid van gewone (dwz. sequentiële) programma's is vaak redelijk gemakkelijk in te zien en vrij goed te testen. Wanneer verschillende programma's tegelijk het gemeenschappelijk geheugen kunnen raadplegen en wijzigen, wordt het veel moeilijker om de correctheid in te zien, terwijl betrouwbaar testen soms vrijwel onmogelijk is. Het zelfde geldt voor het vakgebied van de *gedistribueerde systemen*, waarin verschillende computers met elkaar communiceren (bv. bij web-services).

In de cursus programmacorrectheid beperken we ons voor de eenvoud tot sequentiële programma's, met methoden die ontwikkeld zijn door Hoare [Hoa69] en Dijkstra [Dij76], en waarvoor naar mijn smaak het boek [Kal90] nu het beste leerboek is. Het gaat er hierbij niet om de correctheid van een gegeven programma te bewijzen, maar om bij een gegeven specificatie een correct en efficiënt programma te ontwerpen. Ik gebruik deze methoden zelf ondermeer bij het ontwerpen van programma's voor beeldbewerking (bv. [MRH00]).

3 Oogkleppen

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 bij gecompliceerde programmeerproblemen. We kunnen echter bij het onderwijs niet met ingewikkelde problemen beginnen. Het handwerk moet met eenvoudige probleempjes geleerd worden. We moeten jullie dus dwingen de formele methoden te leren hanteren bij programmeerproblemen waar de correctheid ook operationeel eenvoudig is in te zien.

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 jullie tijdens het vak programmacorrectheid tijdelijk verbieden om operationele argumenten te gebruiken. We komen dan later weer voor het integratieprobleem te staan: na de cursus moet je leren beide ogen open te houden en te gebruiken, en niet uitsluitend te vertrouwen op hetzij operationele intuïtie, hetzij formele methoden.

4 Afleiding, verfijning of verificatie

De ervaring bij sequentiële programma's is, dat het meestal moeilijker is om een gegeven programma correct te bewijzen, dan om vanuit de specificatie opnieuw te beginnen en het programma tegelijk met zijn correctheidsbewijs uit de specificatie af te leiden (waarbij het bewijs iets voorop loopt).

Een voorbeeld hiervan: een maand geleden kwam Arnold Meijster, jullie docent Imperatief Programmeren, bij me met een programma om de grootste gemene deler van twee gehele getallen uit te drukken als gehele lineaire combinatie van de betreffende getallen. Of ik de correctheid daarvan kon aantonen? Mijn antwoord was: "Nee, maar ik kan je het programma wel afleiden" (ik had het eerder gezien).

4.1 Een afleiding van een while-programma uit de postconditie

Gegeven zijn twee positieve gehele getallen x en y. Gevraagd wordt om de grootste gemene deler van x en y uit te drukken als gehele lineaire combinatie van x en y. We gebruiken daarbij het Euclidisch algoritme om de *ggd* te bepalen, dat jullie al bij Imperatief Programmeren gezien hebben.

De postconditie is dus $ggd(x, y) = p * x + q * y$ voor te berekenen gehele getallen p en q. In het Euclidisch algoritme wordt de *ggd* berekend door x en y steeds te veranderen, totdat uiteindelijk één van de twee de *ggd* is. Omdat de oorspronkelijke waarden van x en y in de postconditie staan, houd ik ze constant en voer ik variabelen a en b in, die bij x en y beginnen, en waarvan één van de twee de *ggd* wordt. Omdat ik nog niet weet welke, ga ik a en b allebei uitdrukken als gehele lineaire combinaties van x en y.

```
int a = x , b = y , p = 1 , q = 0 , r = 0 , s = 1 ;
```

Er geldt nu $ggd(x, y) = ggd(a, b)$ en $a = p * x + q * y$ en $b = r * x + s * y$.

In het begin zijn a en b positief, maar ik wil deling met rest toepassen en daarbij zou één van beide wel eens nul kunnen worden. Ik kies ervoor om toe te laten dat b ooit nul wordt. We hebben dus

$$J: \qquad a > 0 \ \wedge \ b \geq 0 \ \wedge \ ggd(x, y) = ggd(a, b) \\ \wedge \ a = p * x + q * y \ \wedge \ b = r * x + s * y \ .$$

We gaan b dus kleiner maken, terwijl we de betrekking J invariant houden. We ontwerpen daartoe een herhaling met een body die ervoor zorgt, dat J aan het eind van de body weer waar is, en dat b kleiner is geworden. Als dit lukt moet $b = 0$ een keer waar worden, maar dan geldt $ggd(a, b) = a$ en dus

$$ggd(x, y) = ggd(a, b) = a = p * x + q * y \ ,$$

wat de gewenste postconditie impliceert. Het programma wordt dus

```
while (b > 0) {  // J geldt hier
    body         // J geldt hier weer
}
```

Hoe moet `body` er nu uit gaan zien? Dit vraagt wat lagere wiskunde (letteralgebra). We gebruiken de operatoren **div** voor geheeltallige deling met rest en **mod** voor de rest. Dan geldt:

$$a = (a \ \mathbf{div} \ b) * b + a \ \mathbf{mod} \ b \quad \wedge \quad 0 \leq a \ \mathbf{mod} \ b < b \ .$$

Elke gemeenschappelijke deler van a en b is dus ook een deler van a **mod** b, en elke gemeenschappelijke deler van b and a **mod** b is een deler van a. Dus $ggd(b, a \ \mathbf{mod} \ b) = ggd(a, b)$.

Ik ga nu a, b, p, q, r en s veranderen en noem de nieuwe waarden a', b', enz. Omdat ik b kleiner wil maken, stel ik dus

$$a' = b \quad \wedge \quad b' = a \ \mathbf{mod}\ b \ \text{(NB gelijkheid, niet: toekenning).}$$

Dan is inderdaad $a' > 0 \ \wedge \ b' \geq 0 \ \wedge \ ggd(x,y) = ggd(a',b')$. Wat moet er met p, q, r en s gebeuren? Wel, we hebben $a' = b = r * x + s * y$ en voor b' vinden we:

$$\begin{aligned} b' &= a - (a \ \mathbf{div}\ b) * b = p * x + q * y - (a \ \mathbf{div}\ b) * (r * x + s * y) \\ &= (p - (a \ \mathbf{div}\ b) * r) * x + (q - (a \ \mathbf{div}\ b) * s) * y \ . \end{aligned}$$

We vinden dus dat J ook geldig is als we bijna overal accenten boven zetten en definiëren: $p' = r$, $q' = s$, $r' = p - (a \ \mathbf{div}\ b) * r$ en $s' = q - (a \ \mathbf{div}\ b) * s$. Als we vervolgens de accenten weglaten, geldt J weer! Om er java van te maken, vervangen we de accenten door het achtervoegsel "`a`". We krijgen voor `body` aldus:

```
int aa = b , ba = a % b ,  pa = r , qa = s ;
int ra = p - (a / b) * r , sa = q - (a / b) * s ;
a = aa ; b = ba ; p = pa ; q = qa ; r = ra ; s = sa ;
```

Als je er een methode van maken wil, kan dit als volgt:

```
int p, q ;
void euclides(int a, int b) {
    p = 1 ; q = 0 ;
    int r = 0 , s = 1 ;
    while (b > 0) {
        int aa = b , ba = a % b ,  pa = r , qa = s ;
        int ra = p - (a / b) * r , sa = q - (a / b) * s ;
        a = aa ; b = ba ; p = pa ; q = qa ; r = ra ; s = sa ;
    }
}
```

Opgave 2. Ontwerp zelf een recursieve methode met dezelfde functionaliteit. Deze methode blijkt iets eenvoudiger te zijn dan de iteratieve versie.

4.2 Stapsgewijze verfijning

Als we een array `a[N]` van integers moeten sorteren, is een deel van de postconditie dat het array opklimmend is in de zin dat

P: $\quad \forall\, i,j : 0 \leq i \leq j < N \quad \Rightarrow \quad a[i] \leq a[j]$.

De notatie is hier misschien iets anders dan je bij het vak Inleiding Logica gehad hebt, maar daar moet je doorheen leren kijken.

Er zijn nu twee bekende sorteeralgoritmen: insertion sort en selection sort, die beide een variatie van P als invariant hebben. Insertion sort gaat volgens

```
k := 1 ;  //  a[0..k-1] is gesorteerd
while (k < n) {
    zet a[k] op de goede plek in a[0..k-1] en
    schuif alles rechts van deze plek een positie naar rechts ;
    k = k+1 ;
}   //  a[0..n-1] is gesorteerd
```

Het alternatief selection sort gaat net iets anders:

```
k := 0 ;  //   a[0..k-1] is gesorteerd en alle elementen ervan
          //   zijn <= de elementen van a[k..n-1]
while (k < n-1) {
    kies j met k <= j < n en a[j] minimaal ;
    verwissel a[k] en a[j] ;
    k = k+1 ;
}   //  a[0..n-1] is gesorteerd
```

In beide gevallen moeten we de deeltaken in de body van de lus nog "verfijnen" met behulp van een binnenlus (maar dan doen we nu niet).

Opgave 3. (a) Druk de invariant van insertion sort uit met behulp van een kwantificatie op dezelfde wijze als de postconditie P, en laat zien dat de invariant tezamen met de ontkennning van de guard de postconditie impliceert.
(b) Idem voor selection sort.

4.3 Verificatie van systemen met concurrency

Concurrency (iets algemener dan multiprogramming) is het verschijnsel dat verschillende processen of processoren gelijktijdig actief zijn en elkaars activiteiten kunnen beïnvloeden door gemeenschappelijk geheugen te raadplegen of te wijzigen, of elkaar boodschappen te sturen.

Voor lastige programma's met concurrency gebruik ik tegenwoordig de stellingbewijzer PVS of de modelchecker Spin. Dit zijn twee zeer verschillende tools, die aan de orde komen in het mastervak Automated Reasoning (vgl. bv. [Vaa06]).

Opgave 4. Zelfstabilisatie volgens [Dij74]. Er zijn N processen met nummers $p \in \{0 \ldots N-1\}$, die elk een getal $\texttt{val}[p] \in \{0 \ldots K-1\}$ bijhouden. We nemen aan dat $N \leq K$. Proces 0 is de "leider". Als $\texttt{val}[0] = \texttt{val}[N-1]$, mag proces 0 zijn getal met 1 modulo K verhogen. De andere processen zijn volgelingen: als $p > 0$ en $\texttt{val}[p] \neq \texttt{val}[p-1]$, dan mag proces p zijn getal gelijk maken aan $\texttt{val}[p-1]$. Altijd als tenminste één van de processen een stap kan doen, zal één van de processen een stap doen. Ze doen hun stappen om de beurt (niet tegelijk). Bewijs nu de volgende drie beweringen:

(a) Er is altijd tenminste één proces dat een stap kan doen.

(b) Als precies één van de processen een stap kan doen, blijft dit zo.

(c) Als het systeem vanuit een willekeurig beginsituatie voldoende stappen doet, ontstaat een situatie waarin precies één proces een stap kan doen.

Aanwijzing voor (c). Laat eerst zien dat de leider van tijd tot tijd een stap moet doen. Bewijs vervolgens met behulp van $N \leq K$, dat de leider ooit een getal krijgt die geen van de volgelingen heeft. Laat ten slotte zien, dat er ooit een situatie van het type (b) ontstaat.

Het protocol uit deze opgave kunnen we testen met de modelchecker Spin, met de volgende "Promelacode", die ik tijdens het college zal uitleggen.

```
#define Nproc 5
#define K 5
byte val[Nproc] ; // shared variable
bool stab ;       // shared variable

proctype node (byte self) { // self is the process identifier
    byte i ;
    do
    :: self == 0 ->
        atomic {
             val[self] == val[Nproc - 1] ; // await
             val[self] = (val[self] + 1) % K
        }
    :: self > 0 ->
        atomic {
            val[self] != val[self-1] ; // await
            val[self] = val[self-1] ;
```

```
            i = 0 ;    // only to verify stabilisation
            do
            :: i <= self && val[i] == val[0] -> i++
            :: else -> break
            od ;
            stab = (i == self+1)
        }   // eventually stab must always hold
    od
}

init {
    byte proc, i ;
    do
    :: proc < Nproc ->
            i = 0 ; // random choice of val[proc]
            do
            :: break
            :: i < K-1 -> i ++
            od ;
            val[proc] = i ; proc ++
    :: else -> break
    od ;
    proc = 0 ;
    atomic {
        do
        :: proc < Nproc -> run node(proc) ; proc ++
        :: else -> break
        od
    }
}  // Voor Spin, zie http://spinroot.com/spin/
```

Referenties

[Dij68] E. W. Dijkstra. The structure of the THE multiprogramming system. *Commun. ACM*, 11:341–346, 1968.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17:643–644, 1974.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[Gib94] W.W. Gibs. Software's chronic crisis. *Scientific American*, pages 72–81, Sept. 1994.

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

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

[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.

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

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

[Vaa06] F. Vaandrager. Modelgebaseerde verificatie en validatie loont. *Bits & Chips*, 28 september: 53–55, 2006.

[Woo06] J. Woodcock. First steps in the Verified Software Grand Challenge. *IEEE Computer*, 39:57–64, October 2006.