stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IN 25/83 AUGUSTUS

P. KLINT

HET SOFTWARE HOUSE - EEN SIMULATIESPEL

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, Kruislaan 413, Amsterdam, The Netherlands.

The Mathematical Centre, founded 11 February 1946, is a non-profit institution for the promotion of pure and applied mathematics and computer science. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

HET SOFTWARE HOUSE -- EEN SIMULATIESPEL

door

Paul Klint

SAMENVATTING

Deze notitie beschrijft het simulatiespel "het software house" behorende bij het college Software Engineering te geven in het studiejaar 1983/1984 aan de Universiteit van Amsterdam. Dit spel heeft tot doel om op kleine schaal allerlei aspekten van programmatuurontwikkeling te demonstreren:

- ontwerpen, implementeren, testen en documenteren van een niet-triviaal programma,
- het definiëren van interfaces tussen programma's,
- onderhandelen over en beoordelen van programmatuur die door anderen gemaakt is, en, tenslotte,
- het wijzigen van door anderen gemaakte programmatuur.

1. INLEIDING

Deze notitie beschrijft het simulatiespel "het software house" behorende bij het college Software Engineering te geven in het studiejaar 1983/1984 aan de Universiteit van Amsterdam. Dit spel heeft tot doel om op kleine schaal allerlei aspekten van programmatuurontwikkeling te demonstreren:

- ontwerpen, implementeren, testen en documenteren van een niet-triviaal programma,
- het definiëren van interfaces tussen programma's,
- onderhandelen over en beoordelen van programmatuur die door anderen gemaakt is, en, tenslotte,
- het wijzigen van door anderen gemaakte programmatuur.

Globale opzet is dat de studenten in groepjes van circa 3 personen verdeeld worden. Ieder groepje vormt een "software house". Tijdens het werkcollege dient een projekt voltooid te worden dat te groot is voor ieder software house afzonderlijk. Daarom wordt het projekt in twee delen gesplitst: modulen A en B. Ieder software house implementeert een van beide modulen. Halverwege begint er een koop/verkoop ronde waarin iedereen probeert het zelf geïmplementeerde moduul (misschien wel een paar maal) te verkopen en bovendien van twee andere software houses de modulen A en B koopt. In de volgende fase gaat ieder software house deze (niet door henzelf geschreven) modulen A en B combineren tot een werkend programma. Dit werkende programma wordt weer verkocht. Tenslotte worden de specifikaties van het projekt iets gewijzigd en moet iedereen een (volgens de originele specificatie) werkend programma daaraan aanpassen.

2. SPELREGELS

START

Een groep van tenminste drie en hooguit vijf personen vormt een software house.

FASE I (4 weken)

Ieder software house ontwerpt, implementeert, test, documenteert en maakt anderzins voor verkoop gereed hetzij een moduul A of een moduul B als beschreven in de bijgaande specificatie (zie sectie 3).

FASE II (2 weken)

Ieder software house integreert modulen A en B die ze gekocht hebben van twee andere software houses, en maakt het resulterende systeem geschikt voor verkoop.

FASE III (3 weken)

Ieder software house koopt een systeem (dat niet het door henzelf geschreven moduul mag bevatten) van een ander software huis, wijzigt dit volgens nader te verstrekken gewijzigde specificaties, en maakt het herziene systeem geschikt voor verkoop.

Prijsverhoudingen

De bank betaalt de volgende (fictieve) prijzen voor het succesvol voltooien van iedere fase op de gestelde datum:

zeer goed	Fl 1000
goed	Fl 750
voldoende	Fl 500
onvoldoende	Fl 250
zeer onvoldoende	Fl 0

Te late voltooiing wordt gestraft met Fl 50 per dag. Deze bedragen worden opgeteld bij de winst of verlies van ieder software house aan het einde van fasen I en II zoals blijkt uit transacties die bij de bank geregistreerd zijn.

Merk op dat deze regels allerlei zaken niet of niet expliciet regelen. Het is, bijvoorbeeld, ongunstig als het ene software house zich onrechtmatig programmatuur van een ander software house toeëigent: dit laatste verliest daardoor immers een potentiële klant. Ieder software house zal er dus naar streven zijn programmatuur te beschermen. Verder zijn er allerlei verschillende overeenkomsten denkbaar op het gebied van leveringsvoorwaarden, garantiebepalingen, samenwerkingsovereenkomsten, enz.

3. HET PROJEKT

Het projekt omvat de dataflow-analyse van programma's in de programmeertaal Algol/0.

3.1 Algol/0

Algol/0 is een zeer simpele programmeertaal. Het enige datatype in de taal zijn integers. Een programma bestaat uit een serie deklaraties voor variabelen en procedures.

Variabele-deklaraties op het buitenste niveau introduceren globale variabelen, die vanuit alle procedures toegankelijk zijn. Variabele-deklaraties die binnen een procedure-deklaratie voorkomen introduceren lokale variabelen die alleen toegankelijk zijn vanuit die betreffende procedure.

Een procedure-deklaratie bestaat uit de procedure-naam, optionele variabele-deklaraties en een rij statements. Procedures hebben geen parameters, maar ze mogen wel recursief zijn. Een procedure, met de naam

"main", is het feitelijke programma, met andere woorden, deze procedure wordt bij het begin van de programma-executie aangeroepen.

Statements zijn assignment-statements, if-statements, while-statements en call-statements. Een assignment-statement kent de waarde van de expressie aan zijn rechterkant, toe aan de (lokale of globale) variabele aan zijn linkerkant. Een if-statement berekent eerst de waarde de test tussen if en then. Als de test true oplevert dan wordt de rij statements tussen then en fi uitgevoerd. Een while-statement berekent herhaaldelijk zijn test en voert daarna, als het resultaat true is, de rij statements uit. Een test die false oplevert beeindigt de executie van het while-statement. Een call-statement roept de procedure met de betreffende naam aan.

In expressies mogen de voor de hand liggende operatoren met standaard prioriteiten voorkomen. Operanden zijn integers of variabelen. In tests mogen de bekende rij relationele operatoren voorkomen.

De lexicale conventies van Algol/0 zijn alsvolgt. Identifiers bestaan uit een letter gevolgd door een optionele rij letters of cijfers. De volgende keywords zijn gereserveerd en mogen niet als identifier gebruikt worden: "begin", "call", "do", "end", "if", "fi", "od", "proc", "then", "var", "while". De identifier "none" is ook gereserveerd om redenen die later duidelijk zullen worden. Een number bestaat uit een of meer cijfers. Commentaar staat tussen accolades en mag niet langer zijn dan een regel.

De grammatica van Algol/0 ziet er alsvolgt uit:

```
<program>    ::= { (<var-decl> | <proc-decl>) ';' }* .
<var-decl>   ::= 'var' { <id> ',' }+ ';' .
<proc-decl>  ::= 'proc' <id> ';' <block> ';' .
<block>      ::= 'begin' [ <var-decl> ] <series> 'end' .
<statement>  ::= <asg-stat> | <call-stat> |
                 <if-stat> | <while-stat> .
<series>     ::= { <statement> ';' }* .
<asg-stat>   ::= <id> ':=' <expr> .
<call-stat>  ::= 'call' <id> .
<if-stat>    ::= 'if' <test> 'then' <series> 'fi'.
<while-stat>::= 'while' <test> 'do' <series> 'od'.
<test>       ::= <expr> <rel-op> <expr> .
<expr>       ::= <term> ( <add-op> <term> )* .
<term>       ::= <factor> ( <mul-op> <factor> ) * .
<factor>     ::= <id> | <number> | '(' <expr> ')' .
<rel-op>     ::= '=' | '!=' | '<' | '>' | '<=' | '>=' .
<add-op>     ::= '+' | '-' .
<mul-op>     ::= '*' | '/' .
```

In de hier gebruikte notatie betekent:

[component] een optionele component.

non-terminal* nul of meer herhalingen van de non-terminal (postfix + betekent een of meer herhalingen).

{ non-terminal separator }+
een lijst van (een of meer) non-terminals gescheiden door separators (postfix + duidt een lijst van een of meer elementen aan).

(componenten) heeft verder geen betekenis maar wordt gebruikt voor groeperen van componenten van een regel.

Een Algol/0 programma dat onder andere de faculteit funktie berekent is:

```
var a, b, c, d, e, f, x, y, z;

{ faculteit: input parameter x; resultaat: z }

proc fac;
begin var n;
      n := x;
      if n = 1 then z := 1 fi;
      if n > 1 then x := n - 1; call fac; z := z * n fi
end;

{ De volgende procedures p1, p2, p3 en p4 demonstreren }
{ gebruik en wijziging van globale variabelen.         }

proc p1;
begin
    var tmp1, tmp2;
    tmp1 := 2 * b;
    tmp2 := tmp1 * tmp1;
    a := 5 * tmp2 + 15;
    call p2;
    call p3
end;

proc p2; begin c := 3 * d end;
proc p3; begin e := 4; call p4 end;
proc p4; begin f := 5 end;

proc main; begin x := 4; call fac; call p1 end;
```

3.2. Doelstellingen van het projekt

Het is de bedoeling om over Algol/0 programma's de volgende informatie te berekenen:

- Welke globale variabelen gebruikt iedere procedure direkt (d.w.z in de procedure zelf) of indirekt (d.w.z. variabelen die gebruikt worden in procedures die door deze procedure aangeroepen worden).

- Welke globale variabelen wijzigt iedere procedure direkt en indirekt.

- Welke procedures roept iedere procedure direkt en indirekt aan.

- Welke procedures zijn direkt of indirekt rekursief.

Het verzamelen van dergelijke informatie is van praktisch belang, bijvoorbeeld bij het onderhouden van grote programmapakketten. Het veranderen of verwijderen van deklaraties wordt erdoor vereenvoudigd; men krijgt bovendien inzicht in mogelijkheden om de modularisering van het programma te verbeteren. Bovendien kan met behulp van dataflow-analyse abnormaal gebruik van variabelen opgespoord worden. Analyse van de struktuur van procedure-aanroepen brengt soms onverwachte, of ongewenste, verbanden tussen procedures aan het licht.

Het formaat van de output van de dataflow-analyse wordt door de volgende grammatica beschreven:

```
<output>       ::= <rule>* ;
<rule>         ::= 'proc' <id> <descriptor> ';' .
<descriptor>   ::= <dir-use> | <indir-use> |
                   <dir-upd> | <indir-upd> |
                   <dir-call> | <indir-call> .
<dir-use>      ::= 'directly uses' <id-list> .
<indir-use>    ::= 'indirectly uses' <id-list> .
<dir-upd>      ::= 'directly updates' <id-list> .
<indir-upd>    ::= 'indirectly updates' <id-list> .
<dir-call>     ::= <recursive> 'directly calls' <id-list> .
<indir-call>   ::= <recursive> 'indirectly calls' <id-list> .
<recursive>    ::= ['is recursive and'] .
<id-list>      ::= 'none' | {<ident> ','}* .
```

Voor iedere procedure moeten precies zes <rule>s voorkomen, voor ieder type descriptor één. De volgorde van de diverse <rule>s en van de identifiers in <id-list>s is verder niet vastgelegd. Ook in de output staat kommentaar tussen accolades. Extra informatie dient dus als commentaar afgedrukt te worden. De output dient te beginnen met commentaar dat in elk geval de naam van het betreffende software house bevat! Het resultaat van de analyse van het voorbeeld programma uit sectie 3.1 kan er bijvoorbeeld alsvolgt uitzien:

```
{ Software House: Klint Data b.v. -- date: 17 augustus 1983 }

{--------------- fac ------------}
proc fac directly uses x, z;
proc fac indirectly uses none;
proc fac directly updates x, z;
proc fac indirectly updates none;
proc fac is recursive and directly calls fac;
proc fac indirectly calls none;

{--------------- main ------------}
proc main directly uses none;
proc main indirectly uses b, d, x, z;
proc main directly updates x;
proc main indirectly updates a, c, e, f, z;
proc main directly calls fac, p1;
proc main indirectly calls p2, p3, p4;

{--------------- p1 ------------}
proc p1 directly uses b;
proc p1 indirectly uses d;
proc p1 directly updates a;
proc p1 indirectly updates c, e, f;
proc p1 directly calls p2, p3;
proc p1 indirectly calls p4;

{--------------- p2 ------------}
proc p2 directly uses d;
proc p2 indirectly uses none;
proc p2 directly updates c;
proc p2 indirectly updates none;
proc p2 directly calls none;
proc p2 indirectly calls none;

{--------------- p3 ------------}
proc p3 directly uses none;
proc p3 indirectly uses none;
proc p3 directly updates e;
proc p3 indirectly updates f;
proc p3 directly calls p4;
proc p3 indirectly calls none;

{--------------- p4 ------------}
proc p4 directly uses none;
proc p4 indirectly uses none;
proc p4 directly updates f;
proc p4 indirectly updates none;
proc p4 directly calls none;
proc p4 indirectly calls none;
```

De analyse van een programma wordt in twee delen gesplitst:

- Moduul A (het "front-end") parseert het programma en konstrueert een abstracte syntaxboom;

- Moduul B (het "back-end") voert de feitelijk dataflow-analyse uit op grond van de syntaxboom.

Het interface tussen beide modulen is vastgelegd in de include-file "treenodes.h" die gemeenschappelijke deklaraties bevat. N.B. de precieze naam wordt op het college bekend gemaakt. Merk op dat de hier gehanteerde splitsing met als interface een volledige syntaxboom voor het huidige probleem wel wat zwaar is: een eenvoudiger interface zou kunnen volstaan. Bij intregratie van dataflow-analyse in een programmeeromgeving ligt het echter voor de hand dat programma's al in de vorm van een syntaxboom beschikbaar zijn. De hier gebruikte verdeling heeft ook als voordeel dat moduul A later eventueel nog met een totaal ander back-end gekombineerd kan worden; denk hierbij bijvoorbeeld aan een pretty printer. De globale struktuur van het totale programma is:

```
program algol0(input, output)

 ... gemeenschappelijke deklaraties voor syntaxboom ...

 ... Moduul A ...

 ... Moduul B ...

begin   { program algol0 }
        { roep Moduul A aan; resultaat in "tree" }
        { roep Moduul B aan met "tree" als input }
end.
```

3.3. Moduul A

Moduul A heeft tot taak om een Algol/0 programma om te zetten in een abstracte syntax boom. De types en vorm van de verschillende knopen in de boom zijn van te voren gegeven als Pascal type-deklaraties. Er zijn globaal twee verschillende strategieën voor ontwerp/implementatie van dit moduul denkbaar:

- Gebruik een recursive descent parser.

- Gebruik een parsergenerator (zoals Yacc en Lex).

Beide alternatieven hebben hun voor- en nadelen en de keuze hangt in hoge mate van ieders interesse en voorkennis af. Bovendien zijn Yacc en Lex alleen maar geschikt om in C geschreven parsers te genereren: bij gebruik van deze hulpmiddelen moet dus ook de problematiek van het mengen van C

en Pascal programma's opgelost worden.

Er zijn nog allerlei toeters en bellen te verzinnen die moduul A kunnen verfraaien (en daardoor de marktwaarde kunnen verhogen). Enkele voorbeelden:

- Laat input van meerdere files toe.
- Produceer optioneel een listing van de input.
- Probeer errorrecovery toe te passen als de invoer niet syntactisch correct is.
- Controleer of alle variabelen wel gedeklareerd zijn.

Bedenk echter dat het bijbouwen van opties tijd kost!

3.4. Moduul B

Moduul B heeft tot taak om, uitgaande van een syntax boom de hierboven geschetste informatie te verzamelen. Ook voor ontwerp/implementatie van moduul B zijn er ruwweg twee benaderingen:

1) Konstrueer per procedure (bit)vektoren die aangeven welke procedures direkt aangeroepen en welke globalen direkt gebruikt of gewijzigd worden. Bereken vervolgens van de verzameling vectoren voor alle procedures in het programma de transitieve afsluiting: dit levert de gewenste informatie op.

2) Doorloop de syntaxboom iteratief en propageer de informatie over aanroepen, gebruik en wijziging zolang dat nog mogelijk is. Zodra dit proces termineert (onder welke voorwaarden is dit het geval?) is ook de gewenste informatie berekend.

Deze beide methodes zijn niet wezenlijk verschillend: methode 1 is efficiënter, maar methode 2 is misschien iets eenvoudiger te implementeren.

Er zijn allerlei verfraaiingen denkbaar:

- Druk, naast de verplichte output, tabellen af die de verschillende dataflow relaties in meer leesbare vorm weergeven.
- Sorteer de identifiers die in de verschillende <id-list>s in de output voorkomen.
- Verzamel extra informatie: spoor b.v. niet gebruikte lokale variabelen op, of houdt per globale variabele bij welke procedures deze variabele gebruiken en/of wijzigen.

4. DIVERSEN

In het voorafgaande is al in grote lijnen vastgelegd hoe het programma voor de dataflow-analyse van Algol/0 programma's georganiseerd moet worden. Er zijn echter nog open vragen waarover kollektief afspraken gemaakt moeten worden:

- Hoe ziet het interface tussen de beide modulen eruit?
- Hoe weet Moduul A wat zijn invoerfiles zijn, hoe wordt de parsetree opgeleverd?
- Hoe komt Moduul B aan de parsetree waarop de dataflow-analyse uitgevoerd moet worden?
- Moeten beide modulen in hetzelfde formaat foutmeldingen geven? Zo ja, hoe ziet de gemeenschappelijke procedure voor foutmeldingen eruit?
- enz.

Binnen elk software house moeten ook allerlei beslissingen genomen worden:

- Wat is de taakverdeling binnen het software house? Wie is verantwoordelijk voor het ontwerp, implementeren, testen, documenteren, onderhandelen met andere software houses, enz.
- Welke globale spelstrategie wordt er gevolgd?
- Afspraken over gemeenschappelijke beprekingen, e.d.
- Het is bijzonder aan te bevelen om van meet af aan een "testplan" te ontwikkelen, d.w.z. al tijdens het ontwerp van het moduul wordt nagedacht over de manier waarop submodulen getest kunnen worden. Hoe ziet dit eruit?

5. APPENDIX: PROGRAMMASKELET VOOR ALGOL/0 DATAFLOW-ANALYSE

```
{-------------------------------------------------------------}
{ Gemeenschappelijke declaraties voor Algol/0 analyse         }
{-------------------------------------------------------------}

program algol0(input, output);
const
        IDLEN           = 10;   { max length of identifiers }
type
        alfa            = packed array [1..IDLEN] of char;
        pid             = ^id;
        pnumber         = ^number;
        pfactor         = ^factor;
        pterm           = ^term;
        plistterm       = ^listterm;
        plistfactor     = ^listfactor;
        pexpr           = ^expr;
        plistaddop      = ^listaddop;
        plistmulop      = ^listmulop;
        ptest           = ^test;
        pasgstat        = ^asgstat;
        pcallstat       = ^callstat;
        pifstat         = ^ifstat;
        pwhilestat      = ^whilestat;
        pstatement      = ^statement;
        pseries         = ^series;
        pblock          = ^block;
        pvardecl        = ^vardecl;
        plistvardecl    = ^listvardecl;
        pprocdecl       = ^procdecl;
        plistprocdecl   = ^listprocdecl;
        pa0program      = ^a0program;

        id      = record name:  alfa; end;
        number  = record value: integer; end;
        factorkind =
                  (fkundef, tyid, tynumber, tyexpr);
        factor  = record case variant:  factorkind  of
                        tyid:           (id:    pid);
                        tynumber:       (number:pnumber);
                        tyexpr:         (expr:  pexpr);
                  end;
        listfactor =
                  record factor:pfactor;
                        next:  plistfactor;
                  end;
        addop   = (aoundef, add, subtract);
        mulop   = (moundef, multiply, divide);
```

```
relop    = (roundef, equal, notequal, less, lesseq,
            greater, greatereq);
listaddop =
           record op:     addop;
                  next:   plistaddop;
           end;
listmulop =
           record op:     mulop;
                  next:   plistmulop;
           end;
term     = record listfactor:    plistfactor;
                  listmulop:     plistmulop;
           end;
listterm =
            record term: pterm;
                   next: plistterm;
            end;
expr     = record listterm:      plistterm;
                  listaddop:     plistaddop;
           end;
test     = record op:            relop;
                  expr1, expr2:  pexpr;
           end;
asgstat  = record id:    pid;
                  expr:  pexpr;
           end;
callstat =
           record id:    pid; end;
ifstat   = record test:          ptest;
                  series:        pseries;
           end;
whilestat =
           record test:          ptest;
                  series:        pseries;
           end;
statkind = (skundef, tyifstat, tywhilestat,
             tycallstat, tyasgstat);
statement =
            record case variant: statkind of
                   tyifstat:     (ifstat:       pifstat);
                   tywhilestat:  (whilestat:    pwhilestat);
                   tycallstat:   (callstat:     pcallstat);
                   tyasgstat:    (asgstat:      pasgstat);
            end;
series = record statement:      pstatement;
                next:           pseries;
         end;
block    = record vardecl:      pvardecl;
                  series:       pseries;
           end;
```

```
        vardecl = record id:    pid;
                         next:  pvardecl;
                  end;
        listvardecl =
                  record vardecl:       pvardecl;
                         next:          plistvardecl;
                  end;
        procdecl =
                  record id:    pid;
                         block: pblock;
                  end;
        listprocdecl =
                  record procdecl:      pprocdecl;
                         next:          plistprocdecl;
                  end;
        a0program =
                  record listvardecl:   plistvardecl ;
                         listprocdecl:  plistprocdecl;
                  end;

{-------------------------------------------------------------}
{                 Moduul A                                    }
{-------------------------------------------------------------}

        { ... }

{-------------------------------------------------------------}
{                 Moduul B                                    }
{-------------------------------------------------------------}

        { ... }

{-------------------------------------------------------------}
{       Het hoofdprogramma algol0                             }
{-------------------------------------------------------------}
begin
        {roep Moduul A aan; resultaat in "tree"}
        {roep Moduul B aan met "tree" als input}
end.
```