1 ANALYSOINTI

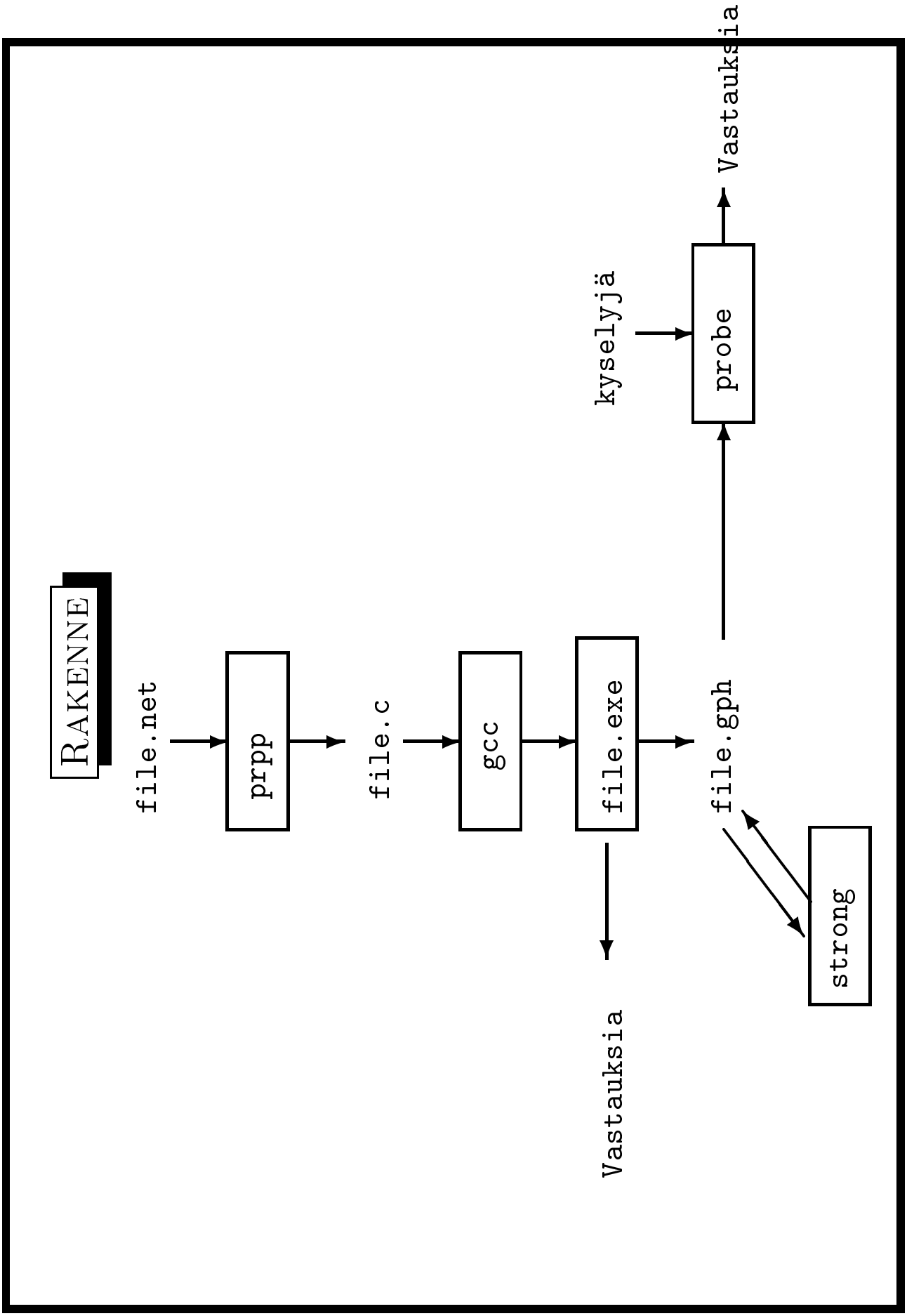
- Annetun järjestelmän analysointi käsin on vaivalloista ja virhealtista.
- Tarvitaan työkaluja joiden avulla analysoinnin eri vaiheet voidaan automatisoida.
- Kaikki olemassaolevat työkaluissa sama periaate:
  - Generoidaan saavutettavuusgraafi tai jokin vastaava rakenne.
  - Käytetään erillaisia ohjelmia ko. graafin tutkimiseen.
- Lista saatavilla olevista PV-työkaluista löytyy osoitteesta:
  `http://www.daimi.au.dk/~petrinet/tools/`

©Johan Lilius 1997, Keijo Heljanko 1998

PROD

- Suunnittelu aloitettu vuonna 1990
- Ensimmäinen versio tehty kokonaan opiskelijatyönä
- Vuosina 92-95 kehitystä rahoittivat TEKES, Nokia, Kone sekä Commit;
- Käytetään teollisuusprojekteissa:
  - FSR-väylän verifoinit: Tele, VTT
  - EMMA: NTC
  - BINAP-protokolla: Tele, Nokia, TEKES
  - Uusia projekteja: UMTS MAC-protokolla, UMTS tietoturva, Nokia, TEKES
  - Uusi korkean tason analysaattori tekeillä: MARIA.
- Hyvää:
  - Paras käytettävissä oleva Pr/T-verkko analysaattori :-)
  - Antaa tehokkaat työkalut saavutettavuusgraafin redusointiin.
- Huonoa:
  - Osittain kryptinen käyttöliittymä.
  - Dokumentaatio

©Johan Lilius 1997, Keijo Heljanko 1998



©Johan Lilius 1997, Keijo Heljanko 1998

SYNTAKSI

- C-preprosessorin kieltä on laajennettu verkkokuvaus direktiiveillä:
  - Paikka:

    #place *place-name* [**lo(***lower-limit***)**]
    [**hi(***upper-limit***)**]
    [**mk(***initial-marking***)**]
  - Transitio:

    #trans *transition-name* [**fd(***list* **)**]
    [**in** {*place-name : input-expression;* ...}]
    [**gate** *gate-expression***;**]
    [**out** {*place-name : output-expression;* ...}]
    [**comp** *C-block*]

    #endtr
  - `PROD` tekee eron input- ja output- muuttujien välillä.
  - *gate-expression* voi sisältää minkä tahansa integer-arvoisen C-kielisen lausekkeen, mutta muuttujat voivat olla vain input-muuttujia.
  - *C-block* voi sisältää mielivaltaista C-koodia. Kutsumalla makroa `Accept();` ilmoitetaan, että transitio on vireessä.
  - Preprosessorimakrojen avulla verkko voidaan *parametrisoida*

©Johan Lilius 1997, Keijo Heljanko 1998

Syntaksi

```
#ifndef n
#define n 5
#endif
#define LEFT(x)  (x)
#define RIGHT(x) (1 + ((x) % n))
#place thinking  lo(<.1.>) hi(<.n.>) mk(<.1..n.>)
#place forks     mk(<.1..n.>)
#place withLeft  lo(<.1.>) hi(<.n.>)
#place eating    lo(<.1.>) hi(<.n.>)
#place withRight lo(<.1.>) hi(<.n.>)
#trans takeLeft
   in  { thinking: <.ph.>; forks: <.LEFT(ph).>; }
   out { withLeft: <.ph.>; }
#endtr
#trans takeRight
   in  { forks: <.RIGHT(ph).>; withLeft: <.ph.>; }
   out { eating: <.ph.>; }
#endtr
#trans putLeft
   in  { eating: <.ph.>; }
   out { withRight: <.ph.>; forks: <.LEFT(ph).>; }
#endtr
#trans putRight
   in  { withRight: <.ph.>; }
   out { thinking: <.ph.>; forks: <.RIGHT(ph).>; }
#endtr
```

©Johan Lilius 1997, Keijo Heljanko 1998

```
#enum wack, rts
#enum sack, rtr
#define EMPTY 2
#define  NOT(x) ((x > 0) ? 0 : 1)

#place Sender lo(<.0,rts.>) hi(<.1,wack.>) mk(<.0, rts.>)
#place Receiver lo(<.0,rtr.>) hi(<.1,sack.>) mk(<.0, rtr.>)
#place Data lo(<.0.>) hi(<.EMPTY.>) mk(<.EMPTY.>)
#place Ack lo(<.0.>) hi(<.EMPTY.>)  mk(<.EMPTY.>)

#trans tout
        in { Sender: <.b, wack.>; }
        out { Sender: <.b, rts.>; }
#endtr
#trans sdata
        in { Sender: <.b, s.>; Data: <.EMPTY.> ; }
        out { Sender: <.b, s1.>; Data: <.b.>; }
        comp { if (s == rts) { s1 = wack;  Accept(); }}
#endtr
#trans ldata
        in { Data: <.b.>;}
        out { Data: <.EMPTY.>;}
        gate (b != EMPTY);
#endtr
#trans rdata
        in { Data: <.b.>; Receiver: <.b1, rtr.>; }
        out { Data: <.EMPTY.>; Receiver: <.b, sack.>; }
        gate (b!= EMPTY);
#endtr
#trans sack
        in { Receiver: <.b, sack.>; Ack: <.EMPTY.>;}
        out { Receiver: <.b1, rtr.>; Ack: <.b.>;}
        comp { b1 = NOT(b); Accept(); }
#endtr
#trans lack
        in { Ack: <.b.>;}
        out { Ack: <.EMPTY.>;}
        gate (b != EMPTY);
#endtr
#trans rack
        in {Ack: <.b.>; Sender: <.b1, wack.>; }
        out {Ack: <.EMPTY.>; Sender: <.b2, s1.>; }
        gate (b != EMPTY);
        comp { if (b1 == b) {
                        s1 = rts; b2 = NOT(b); Accept();
                } else if (b1 == NOT(b)) {
                        s1 = wack; b2 = b1; Accept();
        }
        }
#endtr
```

©Johan Lilius 1997, Keijo Heljanko 1998

KÄYTTÖ

- Oletetaan, että filosofiverkko on talletettu tiedostoon `ph.net`
- Ajetaan prod:

```
(topos) prod/sample %:  prod ph.init
```

- Ajetaan generoitu saavutettavuusgraafigeneraattori:

```
(topos) prod/sample %:  ph -.
.........................
```

- Lasketaan vahvasti kytketyt komponentit:

```
(topos) prod/sample %:  strong ph
```

- Käynnistetään graafintutkintaohjelma:

```
(topos) prod/sample %:  probe ph
```

- Kysellään yleisiä tietoja:

```
0#statistics
Number of nodes:  242
Number of arrows:  805
Number of terminal nodes:  1
Number of nodes that have been completely
processed:  242
Number of strongly connected components:  2
Number of nontrivial terminal strongly
connected components:  0
```

©Johan Lilius 1997, Keijo Heljanko 1998

Käyttö

- Tarkastellaan erityisiä valmiiksi laskettuja solmujoukkoja:

```
0#sets
Strongly connected components:  %%0..%%1
Special sets:
%0:  ** terminal nodes **
```

- Tarkastellaan lukkiumaa:

```
0#showends %0
Node 91, belongs to strongly connected
component %%0
withLeft:  <.1.> + <.2.> + <.3.> + <.4.> +
<.5.>
------------------------
1 path end nodes
------------------------
```

©Johan Lilius 1997, Keijo Heljanko 1998

Vahvasti kytketyt komponentit

- Suunnatun graafin on vahvasti kytketty komponentti on aligraafi jonka jokaisesta solmusta on polku jokaiseen toiseen aligraafin solmuun.
- Yksittäinen solmu muodostaa triviaalisti vahvasti kytketyn komponentin.
- `strong` laskee saavutettavuusgraafin *maksimaalisesti* vahvasti kytketyt komponentit.
- Saavutettavuusgraafin *maksimaalisesti* vahvasti kytketyt komponentit muodostavat graafin.
- Tyypillisesti reaktiivisessa järjestelmässä tulisi olla vain yksi ei-triviaali terminaalinen vahvasti kytketty komponentti. Tämä siksi, että järjestelmässä ei saisi olla toimintoja, joita ei päästä suorittamaan.

©Johan Lilius 1997, Keijo Heljanko 1998

KÄYTTÖ

- Tarkastellaan vahvasti kytkettyjä komponentteja:

```
0#allcomps
Component %%0
-----------------------------------------------
  91
%%0 has 1 nodes
-----------------------------------------------
Component %%1
-----------------------------------------------
   0,        1,        2,        3,        4,        5,        6,        7,
   8,        9,       10,       11,       12,       13,       14,       15,
  16,       17,       18,       19,       20,       21,       22,       23,
  24,       25,       26,       27,       28,       29,       30,       31,
  32,       33,       34,       35,       36,       37,       38,       39,
  40,       41,       42,       43,       44,       45,       46,       47,
  48,       49,       50,       51,       52,       53,       54,       55,
  56,       57,       58,       59,       60,       61,       62,       63,
  64,       65,       66,       67,       68,       69,       70,       71,
  72,       73,       74,       75,       76,       77,       78,       79,
  80,       81,       82,       83,       84,       85,       86,       87,
  88,       89,       90,       92,       93,       94,       95,       96,
  97,       98,       99,      100,      101,      102,      103,      104,
 105,      106,      107,      108,      109,      110,      111,      112,
 113,      114,      115,      116,      117,      118,      119,      120,
 121,      122,      123,      124,      125,      126,      127,      128,
 129,      130,      131,      132,      133,      134,      135,      136,
 137,      138,      139,      140,      141,      142,      143,      144,
 145,      146,      147,      148,      149,      150,      151,      152,
 153,      154,      155,      156,      157,      158,      159,      160,
 161,      162,      163,      164,      165,      166,      167,      168,
 169,      170,      171,      172,      173,      174,      175,      176,
 177,      178,      179,      180,      181,      182,      183,      184,
```

©Johan Lilius 1997, Keijo Heljanko 1998

```
 185,      186,      187,      188,      189,      190,      191,      192,
 193,      194,      195,      196,      197,      198,      199,      200,
 201,      202,      203,      204,      205,      206,      207,      208,
 209,      210,      211,      212,      213,      214,      215,      216,
 217,      218,      219,      220,      221,      222,      223,      224,
 225,      226,      227,      228,      229,      230,      231,      232,
 233,      234,      235,      236,      237,      238,      239,      240,
 241
%%1 has 241 nodes
-----------------------------------------------
2 strongly connected components
-----------------------------------------------
0#
```

©Johan Lilius 1997, Keijo Heljanko 1998

KÄYTTÖ

- Etsitään vielä eräs lukkiumaan johtanut transitiosekvenssi:

```
0#query volatile verbose bspan(true) %0
PATH
Node 0, belongs to strongly connected component %%1
  thinking: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 1
Node 1, belongs to strongly connected component %%1
  thinking: <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.2.> + <.3.> + <.4.> + <.5.>
  withLeft: <.1.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 2
Node 6, belongs to strongly connected component %%1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.3.> + <.4.> + <.5.>
  withLeft: <.1.> + <.2.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 3
Node 21, belongs to strongly connected component %%1
  thinking: <.4.> + <.5.>
  forks: <.4.> + <.5.>
  withLeft: <.1.> + <.2.> + <.3.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 4
Node 51, belongs to strongly connected component %%1
  thinking: <.5.>
  forks: <.5.>
  withLeft: <.1.> + <.2.> + <.3.> + <.4.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 5
Node 91, belongs to strongly connected component %%0
  withLeft: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
-----------------------------------------------
1 paths
-----------------------------------------------
```

©Johan Lilius 1997, Keijo Heljanko 1998

KÄYTTÖ

- Tiloja ja tilasiirtymiä voi tutkia seuraavien komentojen avulla:
  - `look` n: tulostaa merkinnän n sisällön
  - `succ` n: tulostaa merkinnän n seuraajamerkinnät sekä mitä kautta niihin päästään.
  - `pred` n: tulostaa merkinnän n edeltäjämerkinnät sekä mitä kautta niihin päästään.
  - `next` n: siirrytään kaarta n pitkin seuraavaan merkintään.

  Jos komennosta jätetään merkinnän numero pois, tarkastellaan aktiivista merkintää. Aktiivista merkintää voi vaihtaa `goto` n komennon avulla.
- `probe`:n palauttaman informaation määrä voi säätää antamalla `query` komennon yhteydessä optiot `mute`, `verbose` tai `superverbose`. Oletusarvo on `mute`:n ja `verbose`:n välissä.

©Johan Lilius 1997, Keijo Heljanko 1998

Testerit

- Usein koko saavutettavuusgraafin generointi ei ole mahdollista. Esim. n:n filosofin saavutettavusgraafissa on

  $3^n - 1$ solmua, ja

  $n \cdot (2 \cdot 3^{n-1} - 1)$ kaarta.

```
n =  0, s =          0, k =          0
n =  1, s =          1, k =          1
n =  2, s =          3, k =         10
n =  3, s =          9, k =         51
n =  4, s =         27, k =        212
n =  5, s =         81, k =        805
n =  6, s =        243, k =       2910
n =  7, s =        729, k =      10199
n =  8, s =       2187, k =      34984
n =  9, s =       6561, k =     118089
n = 10, s =      19683, k =     393650
n = 11, s =      59049, k =    1299067
n = 12, s =     177147, k =    4251516
n = 13, s =     531441, k =   13817453
n = 14, s =    1594323, k =   44641030
n = 15, s =    4782969, k =  143489055
n = 16, s =   14348907, k =  459165008
```

©Johan Lilius 1997, Keijo Heljanko 1998

Testerit

- Jos saavutettavuusgraafin koko tekee sen täydellisen generoimisen mahdottomaksi, voidaan kuitenkin usein virheitä löytää seuraavan menetelmän avulla:
  - Sen sijaan että virhettä etsitään generoidusta graafista *testataan* jokaista merkintää graafia generoitaessa.
  - Tällaista menetelmää kutsutaan *on-the-fly* (lennossa) verifioinniksi.
- `PROD` tukee seuraavien virheiden etsintää lennosta *testerien* avulla:
  1. lukkiumat (`deadlock`)
  2. yleinen virheellinen tila (`reject`)
  3. pillastumat (`livelock`)
  4. äärettömät polut (`infinite`)

  Lisäksi on mahdollista määritellä testeri temporaalilogiikan avulla.

©Johan Lilius 1997, Keijo Heljanko 1998

Testerit

- Lennossa verifioinnin perusidea on, että verkkoon kytketään erityinen *testeri*-paikka siten, että siihen tulee merkintä ainostaan kun jokin haluttu virhe on verkossa tapahtunut.
- `livelock`- ja `infinite`-testerit etsivät saavutettavuusgraafista silmukkaa joka kulkee testeri-paikan kautta. Tämän kurssin puitteissa ei `infinite`-testeriä tarvita.

©Johan Lilius 1997, Keijo Heljanko 1998

Testeri

- Lukkiuman tai pillastuman etsiminen testerin avulla on yksinkertaista:

  ```
  #place tester lo(<.0.>) hi(<.0.>) mk(<.0.>)
  #tester tester deadlock(<.0.>)
  ```

  tai

  ```
  #place tester lo(<.0.>) hi(<.0.>) mk(<.0.>)
  #tester tester livelock(<.0.>)
  ```

  `PROD` osaa itse kytkeä paikan verkkoon.
- Jos halutaan tarkastella jotain tiettyä virhettä voidaan se tehdä kytkemällä testeri päälle verkosta käsin.
- Oletetaan, että halutaan tarkistaa, että filosofi n ei kuole nälkään. Filosofin n nääntyminen tarkoittaa, että graafissa on silmukka jonka jokaisessa merkinnässä paikan `withLeft` merkintä sisältää filosofin n.
- Transito `takeLeft` kytketään testeripaikkan siten, että sen merkinnäksi tulee `<.1.>` kun filosofi n saa vasemman haarukan.
- Vastaavasti transitio `takeRight` kytketään testeripaikkan siten, että sen merkinnäksi palautuu `<.0.>` kun filosofi n saa oikean haarukan.

©Johan Lilius 1997, Keijo Heljanko 1998

Testerit

- Verkosta tulee nyt seuraavanlainen:

```
#define n 5
#define LEFT(x)  (x)
#define RIGHT(x) (1 + ((x) % n))
#place thinking  lo(<.1.>) hi(<.n.>) mk(<.1..n.>)
#place forks     mk(<.1..n.>)
#place withLeft  lo(<.1.>) hi(<.n.>)
#place eating    lo(<.1.>) hi(<.n.>)
#place withRight lo(<.1.>) hi(<.n.>)
#place tester    lo(<.0.>) hi(<.1.>) mk(<.0.>)
#tester tester   livelock(<.1.>)
#trans takeRight
   in  { forks: <.RIGHT(x).>; withLeft: <.x.>;
         tester: (x == n)<.1.>; }
   out { eating: <.x.>; tester: (x == n)<.0.>; }
#endtr
#trans takeLeft
   in  { thinking: <.x.>; forks: <.LEFT(x).>;
         tester: (x == n)<.0.>; }
   out { withLeft: <.x.>; tester: (x == n)<.1.>; }
#endtr
#trans putLeft
   in  { eating: <.x.>; }
   out { withRight: <.x.>; forks: <.LEFT(x).>; }
#endtr
#trans putRight
   in  { withRight: <.x.>; }
   out { thinking: <.x.>; forks: <.RIGHT(x).>; }
#endtr
```

©Johan Lilius 1997, Keijo Heljanko 1998

Testeri

- Pillastumaa päästään tarkastelemaan seuraavasti:
  - Saavutettavuusgraafigenerattori ilmoittaa löydetyn silmukan:

```
(topos) prod/philosphers %: prod dining.init
(topos) prod/philosphers %: dining -.

Livelock reached
Loop 5 [1> 7 [0> 11 [0> 15 [0> 18 [0> 21 [1> 24 [1> 27 [3> 5
For more information, start "probe dining"
and look at the set %1
```

  - Silmukkaan johtava polku on seuraava:

```
(topos) prod/philosphers %: probe ph
0#query bspan(true) %1
   0  [4>    5
1 paths
Built set %2
-----------------------------------------------
```

©Johan Lilius 1997, Keijo Heljanko 1998