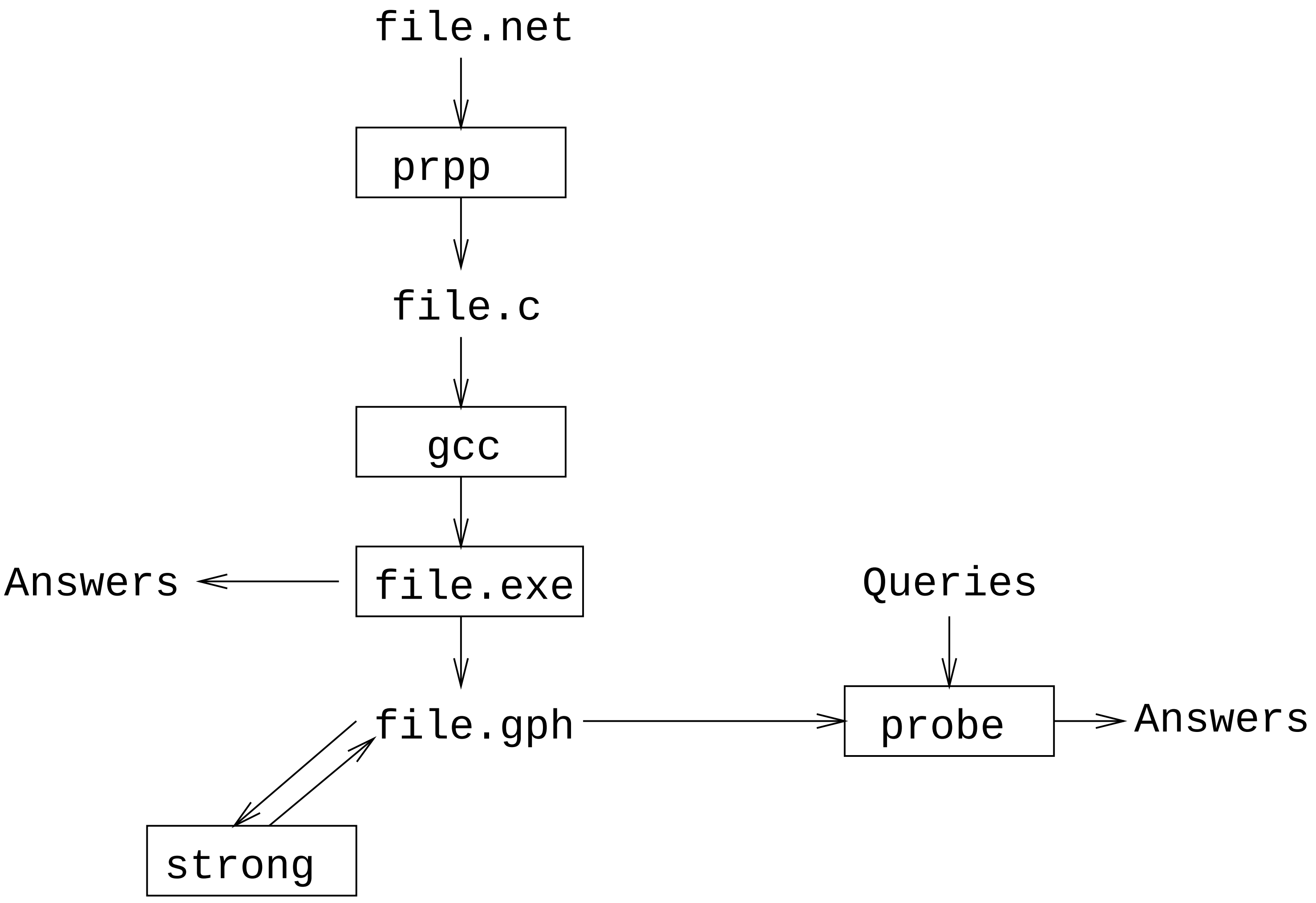
1 ANALYSIS

- Analysis by hand of a given system is tedious and prone to errors.
- *(However, there is a Petri net paradigm (Reisig et al.) based on hand-written proofs of non-trivial systems.)*
- The verification engineering approach is to have tools to automatise the different phases of analysis.
- All the existing tools are based on the same principle:
  - Reachability graph or some other similar structure is constructed.
  - Different programs are employed to analyse said graph.
- A list of available Petri net tools can be found in the following address: `http://www.daimi.au.dk/~petrinet/tools/`

PROD

- Design started in 1990. Leading figure of programming effort: Kimmo Varpaaniemi.
- First version entirely as a student exercise.
- In 1992-95, development was supported by funding from TEKES, Nokia, Kone, and Commit;
- Usage in industrial projects:
  - Verification of the FSR bus: Tele, VTT.
  - EMMA: Nokia Telecommunications.
  - BINAP protocol: Tele, Nokia, TEKES
  - New projects: UMTS MAC protocol, UMTS data security: Nokia, Tekes.
  - New high level Petri net analyser, MARIA, being built. Could be called PROD++, too.
- Benefits:
  - A fairly good Pr/T net analyser.
  - Efficient, up-to-date tools to reduce the the reachability graph.
- Liabilities:
  - Unfriendly, non-graphical user interface.
  - Incomplete documentation.

Structure


file.net
prpp
file.c
gcc
file.exe
Answers
Queries
file.gph
probe
Answers
strong

Syntax

- The language of the C preprocessor extended with net description directives;
  - Place:

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

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

    #**endtr**
  - `PROD` distinguishes input variables and output variables.
  - *gate-expression* may contain any C expression returning an integer where the variables can only be the input variables.
  - *C-block* can contain arbitrary C code. Invocation of macro `Accept();` that the transition is in concession.
  - The preprocessor macros enable the *parametrization* of the net.

Syntax

```
#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
```

```
#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
```

Usage

- It is assumed that the philosopher net is stored in a file called `ph.net`
- Run the `PROD` program:

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

- Run the generated reachability graph generator:

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

- Compute the strongly connected components:

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

- Start the graph exploration program:

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

- Query general information:

```
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
```

Usage

- Check certain precomputed sets of nodes:

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

- Check deadlocks:

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

Strongly connected components

- A strongly connected component of a directed graph is a subgraph whose all nodes have a path to all the other nodes of the subgraph.
- A single node is the trivial case of a strongly connected component.
- `strong` computes the *maximally* strongly connected components in the reachability graph.
- The maximally strongly connected components compose a graph.
- Typically, a reactive system should have only one non-trivial terminal strongly connected component. This is to avoid non-executable functions in the system.

Usage

- Let us consider the strongly connected components:

```
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,
```

```
 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#
```

Usage

- Let us find a transition sequence which ends up in a deadlock:

```
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
------------------------------------------------
```

Usage

- Markings (global States) and fired transitions (state transitions) can be explored with the following commands:
  - `look` n: display the contents of the marking n.
  - `succ` n: display the follower markings of the marking n with the transitions and valuations.
  - `pred` n: display the predecessor markings of the marking n with the transitions and their respective valuations.
  - `next` n: move to the next marking along the arc n.

  If the number of the marking is omitted, the active marking is being explored. The active marking can be changed with the command `goto` n.
- The amount of information displayed with the `probe` commands can be set by giving the command `query` with one of the options `mute`, `verbose`, or `superverbose`. The default is between `mute` and `verbose`.

Testers

- Often it is impossible or impractical to generate all of the reachability graph. E.g., the size of the reachability graph for the n dining philosophers problem is

$3^n - 1$	nodes, and
$n \cdot (2 \cdot 3^{n-1} - 1)$	arcs.

```
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
```

Testers

- If the size of the reachability graph renders its complete generation impossible, the following method enables the detection of several errors:
  - Instead of searching for the error in the generated graph, each marking is *tested* during the graph generation.
  - This approach is called the *on-the-fly* verification.
- `PROD` supports the on-the-fly detection of errors with *testers*:
  1. Markings with no transitions in concession (`deadlock`).
  2. Errors in general (`reject`).
  3. Circular paths (loops) from which is impossible to leave (`livelock`).
  4. Infinite paths (`infinite`).

  It is also possible to define a tester by means of temporal logics.

TESTERS

- The basic idea about on-the-fly verification is that the a special *tester* place is inserted to the net so that it will receive a token when the error occurs.
- `livelock` and `infinite` testers are searching for a loop traversing the tester place in the reachability graph. Within the scope of this course the `infinite` tester is not needed.

Tester

- It is simple to look for a deadlock or divergence with a tester.

  ```
  #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` can insert the tester place to the net.
- If the test purpose is to observe a particular error it can be done by switching the tester "on" from the net.
- Suppose we want to check that the philosopher n does not starve. Starvation of a philosopher means that there is a loop in the graph where all the marking have the philosopher n in the marking `withLeft`.
- Transition `takeLeft` is connected with the tester place so that the tester place marking becomes `<.1.>` when the philosopher n grabs the left fork.
- Correspondingly, the transition `takeRight` is connected with the tester place so that the the tester place marking is restored as `<.0.>` when the philosopher n grabs the right fork.

Testers

- Now the net will look like this:

```
#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
```

Testers

- A divergence can be found as follows:
  - The reachability graph generator informs about the presence of a loop:

```
(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
```

  - The path to the divergence is below:

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