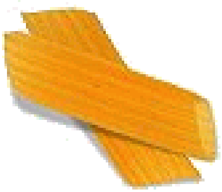
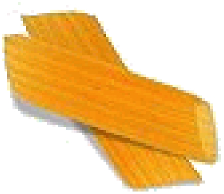


One hundred years of the PEPA tools

Stephen Gilmore
Laboratory for Foundations of Computer Science
The University of Edinburgh

12th June 2003

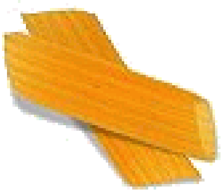


One hundred years of the PEPA tools

Stephen Gilmore
Laboratory for Foundations of Computer Science
The University of Edinburgh

12th June 2003

PEPA is ten years old!



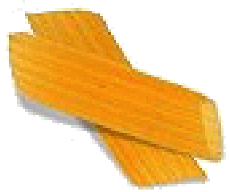
One hundred years of the PEPA tools

Stephen Gilmore
Laboratory for Foundations of Computer Science
The University of Edinburgh

12th June 2003

PEPA is ten years old!

One year programming in ML $\equiv$ ten years programming in Java



Background

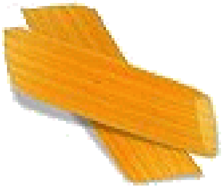
- Performance Evaluation Process Algebra (PEPA) is used as a formal description language for Markov chain modelling. PEPA is a compact language with a small number of primitive operations.

  Prefix: $(\alpha, r).P$ performs α at rate r to become P.

  Choice: $P + Q$ sets up a race between P and Q. The first to perform an action wins: the other is discarded.

  Cooperation: $P \underset{L}{\bowtie} Q$ runs P and Q in parallel, synchronising on activities in L.

  Hiding: P/L hides the activities in L, preventing cooperands from synchronising on them.



Background

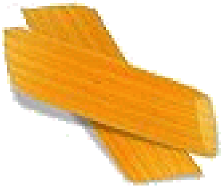
- Performance Evaluation Process Algebra (PEPA) is used as a formal description language for Markov chain modelling. PEPA is a compact language with a small number of primitive operations.

Prefix: $(\alpha, r).P$ performs α at rate r to become P.

Choice: $P + Q$ sets up a race between P and Q. The first to perform an action wins: the other is discarded.

Cooperation: $P \underset{L}{\bowtie} Q$ runs P and Q in parallel, synchronising on activities in L.

Hiding: P/L hides the activities in L, preventing cooperands from synchronising on them.



Background

- Performance Evaluation Process Algebra (PEPA) is used as a formal description language for Markov chain modelling. PEPA is a compact language with a small number of primitive operations.

Prefix: $(\alpha, r).P$ performs α at rate r to become P.

Choice: $P + Q$ sets up a race between P and Q. The first to perform an action wins: the other is discarded.

Cooperation: $P \underset{L}{\bowtie} Q$ runs P and Q in parallel, synchronising on activities in L.

Hiding: P/L hides the activities in L, preventing cooperands from synchronising on them.

Background

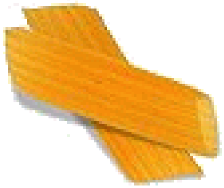
- Performance Evaluation Process Algebra (PEPA) is used as a formal description language for Markov chain modelling. PEPA is a compact language with a small number of primitive operations.

  Prefix: $(\alpha, r).P$ performs α at rate r to become P.

  Choice: $P + Q$ sets up a race between P and Q. The first to perform an action wins: the other is discarded.

  Cooperation: $P \underset{L}{\bowtie} Q$ runs P and Q in parallel, synchronising on activities in L.

  Hiding: P/L hides the activities in L, preventing cooperands from synchronising on them.



Background

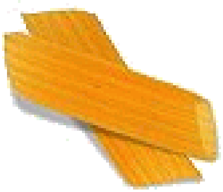
- Performance Evaluation Process Algebra (PEPA) is used as a formal description language for Markov chain modelling. PEPA is a compact language with a small number of primitive operations.

  Prefix: $(\alpha, r).P$ performs α at rate r to become P.

  Choice: $P + Q$ sets up a race between P and Q. The first to perform an action wins: the other is discarded.

  Cooperation: $P \underset{L}{\bowtie} Q$ runs P and Q in parallel, synchronising on activities in L.

  Hiding: P/L hides the activities in L, preventing cooperands from synchronising on them.



Background

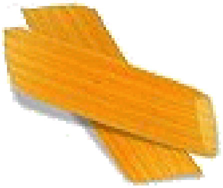
- Performance Evaluation Process Algebra (PEPA) is used as a formal description language for Markov chain modelling. PEPA is a compact language with a small number of primitive operations.

  Prefix: $(\alpha, r).P$ performs α at rate r to become P.

  Choice: $P + Q$ sets up a race between P and Q. The first to perform an action wins: the other is discarded.

  Cooperation: $P \underset{L}{\bowtie} Q$ runs P and Q in parallel, synchronising on activities in L.

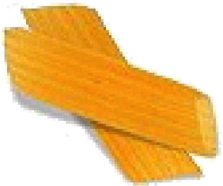
  Hiding: P/L hides the activities in L, preventing cooperands from synchronising on them.



A first PEPA tool: The PEPA Workbench

- Our first PEPA tool was the PEPA Workbench, implemented in Standard ML.
- The PEPA syntax can be represented simply as an ML datatype.

```
datatype Component =
     PREFIX of (Activity * Rate) * Component          (*  .  *)
   | CHOICE of Component * Component                  (*  +  *)
   | COOP of Component * Component * Activity list    (*  ⋈  *)
   | HIDING of Component * Activity list             (*  /  *)
   | VAR of Identifier                                (*  X  *)
   | DEF of Identifier * Component * Component        (*  ≝  *)
```



A first PEPA tool: The PEPA Workbench

- Our first PEPA tool was the PEPA Workbench, implemented in Standard ML.
- The PEPA syntax can be represented simply as an ML datatype.

```
datatype Component =
     PREFIX of (Activity * Rate) * Component          (*  .  *)
   | CHOICE of Component * Component                  (*  +  *)
   | COOP of Component * Component * Activity list    (*  ⋈  *)
   | HIDING of Component * Activity list              (*  /  *)
   | VAR of Identifier                                (*  X  *)
   | DEF of Identifier * Component * Component        (*  ≝  *)
```

A first PEPA tool: The PEPA Workbench

- Our first PEPA tool was the PEPA Workbench, implemented in Standard ML.
- The PEPA syntax can be represented simply as an ML datatype.

```
datatype Component =
     PREFIX of (Activity * Rate) * Component          (*  .  *)
   | CHOICE of Component * Component                  (*  +  *)
   | COOP of Component * Component * Activity list    (*  ⋈  *)
   | HIDING of Component * Activity list              (*  /  *)
   | VAR of Identifier                                (*  X  *)
   | DEF of Identifier * Component * Component        (*  ≝  *)
```

A first PEPA tool: The PEPA Workbench

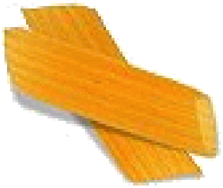
- Our first PEPA tool was the PEPA Workbench, implemented in Standard ML.
- The PEPA syntax can be represented simply as an ML datatype.

```
datatype Component =
     PREFIX of (Activity * Rate) * Component             (*  .  *)
   | CHOICE of Component * Component                      (*  +  *)
   | COOP of Component * Component * Activity list        (*  ⋈  *)
   | HIDING of Component * Activity list                  (*  /  *)
   | VAR of Identifier                                    (*  X  *)
   | DEF of Identifier * Component * Component            (*  ≝  *)
```

A first PEPA tool: The PEPA Workbench

- Our first PEPA tool was the PEPA Workbench, implemented in Standard ML.
- The PEPA syntax can be represented simply as an ML datatype.

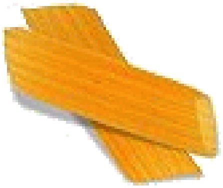
```
datatype Component =
     PREFIX of (Activity * Rate) * Component          (*  .  *)
   | CHOICE of Component * Component                  (*  +  *)
   | COOP of Component * Component * Activity list    (*  ⋈  *)
   | HIDING of Component * Activity list              (*  /  *)
   | VAR of Identifier                                (*  X  *)
   | DEF of Identifier * Component * Component        (*  ≝  *)
```



A first PEPA tool: The PEPA Workbench

- Our first PEPA tool was the PEPA Workbench, implemented in Standard ML.
- The PEPA syntax can be represented simply as an ML datatype.

```
datatype Component =
     PREFIX of (Activity * Rate) * Component         (*  .  *)
   | CHOICE of Component * Component                 (*  +  *)
   | COOP of Component * Component * Activity list   (*  ⋈  *)
   | HIDING of Component * Activity list             (*  /  *)
   | VAR of Identifier                               (*  X  *)
   | DEF of Identifier * Component * Component       (*  ≝  *)
```



A first PEPA tool: The PEPA Workbench

- Our first PEPA tool was the PEPA Workbench, implemented in Standard ML.
- The PEPA syntax can be represented simply as an ML datatype.

```
datatype Component =
      PREFIX of (Activity * Rate) * Component          (* . *)
    | CHOICE of Component * Component                  (* + *)
    | COOP of Component * Component * Activity list    (* ⋈ *)
    | HIDING of Component * Activity list              (* / *)
    | VAR of Identifier                                (* X *)
    | DEF of Identifier * Component * Component        (* ≝ *)
```

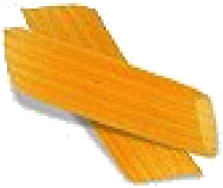
A first PEPA tool: The PEPA Workbench

- Our first PEPA tool was the PEPA Workbench, implemented in Standard ML.
- The PEPA syntax can be represented simply as an ML datatype.

```
datatype Component =
     PREFIX of (Activity * Rate) * Component          (* . *)
   | CHOICE of Component * Component                  (* + *)
   | COOP of Component * Component * Activity list    (* ⋈ *)
   | HIDING of Component * Activity list              (* / *)
   | VAR of Identifier                                (* X *)
   | DEF of Identifier * Component * Component        (* ≝ *)
```

The PEPA Workbench: derivatives

```
fun derivative E (PREFIX (a as (alpha, rate), P)) = [(a, P)]
  | derivative E (CHOICE (P, Q)) =
         (derivative E P) @ (derivative E Q)
  | derivative E (COOP (P, Q, L)) =
         let
             val (dP, dQ) = (derivative E P, derivative E Q)
             val (fP, fQ) = (filterout dP L, filterout dQ L)
         in
             (map (fn (a, P')) => (a, COOP (P', Q, L))) fP)
           @ (map (fn (a, Q')) => (a, COOP (P, Q', L))) fQ)
           @ cooperations dP dQ L
         end
  | derivative E (HIDING (P, L)) = ...
```

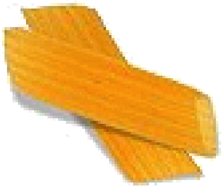


The PEPA Workbench: derivatives

```
fun derivative E (PREFIX (a as (alpha, rate), P)) = [(a, P)]
  | derivative E (CHOICE (P, Q)) =
         (derivative E P) @ (derivative E Q)
  | derivative E (COOP (P, Q, L)) =
         let
             val (dP, dQ) = (derivative E P, derivative E Q)
             val (fP, fQ) = (filterout dP L, filterout dQ L)
         in
             (map (fn (a, P')) => (a, COOP (P', Q, L))) fP)
           @ (map (fn (a, Q')) => (a, COOP (P, Q', L))) fQ)
           @ cooperations dP dQ L
         end
  | derivative E (HIDING (P, L)) = ...
```

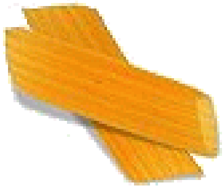
The PEPA Workbench: derivatives

```
fun derivative E (PREFIX (a as (alpha, rate), P)) = [(a, P)]
  | derivative E (CHOICE (P, Q)) =
         (derivative E P) @ (derivative E Q)
  | derivative E (COOP (P, Q, L)) =
         let
             val (dP, dQ) = (derivative E P, derivative E Q)
             val (fP, fQ) = (filterout dP L, filterout dQ L)
         in
             (map (fn (a, P') => (a, COOP (P', Q, L))) fP)
           @ (map (fn (a, Q') => (a, COOP (P, Q', L))) fQ)
           @ cooperations dP dQ L
         end
  | derivative E (HIDING (P, L)) = ...
```



The PEPA Workbench: derivatives

```
fun derivative E (PREFIX (a as (alpha, rate), P)) = [(a, P)]
  | derivative E (CHOICE (P, Q)) =
         (derivative E P) @ (derivative E Q)
  | derivative E (COOP (P, Q, L)) =
         let
             val (dP, dQ) = (derivative E P, derivative E Q)
             val (fP, fQ) = (filterout dP L, filterout dQ L)
         in
             (map (fn (a, P')) => (a, COOP (P', Q, L))) fP)
           @ (map (fn (a, Q')) => (a, COOP (P, Q', L))) fQ)
           @ cooperations dP dQ L
         end
  | derivative E (HIDING (P, L)) = ...
```



The PEPA Workbench: derivatives

```
fun derivative E (PREFIX (a as (alpha, rate), P)) = [(a, P)]
  | derivative E (CHOICE (P, Q)) =
         (derivative E P) @ (derivative E Q)
  | derivative E (COOP (P, Q, L)) =
         let
             val (dP, dQ) = (derivative E P, derivative E Q)
             val (fP, fQ) = (filterout dP L, filterout dQ L)
         in
             (map (fn (a, P’) => (a, COOP (P’, Q, L))) fP)
           @ (map (fn (a, Q’) => (a, COOP (P, Q’, L))) fQ)
           @ cooperations dP dQ L
         end
  | derivative E (HIDING (P, L)) = ...
```

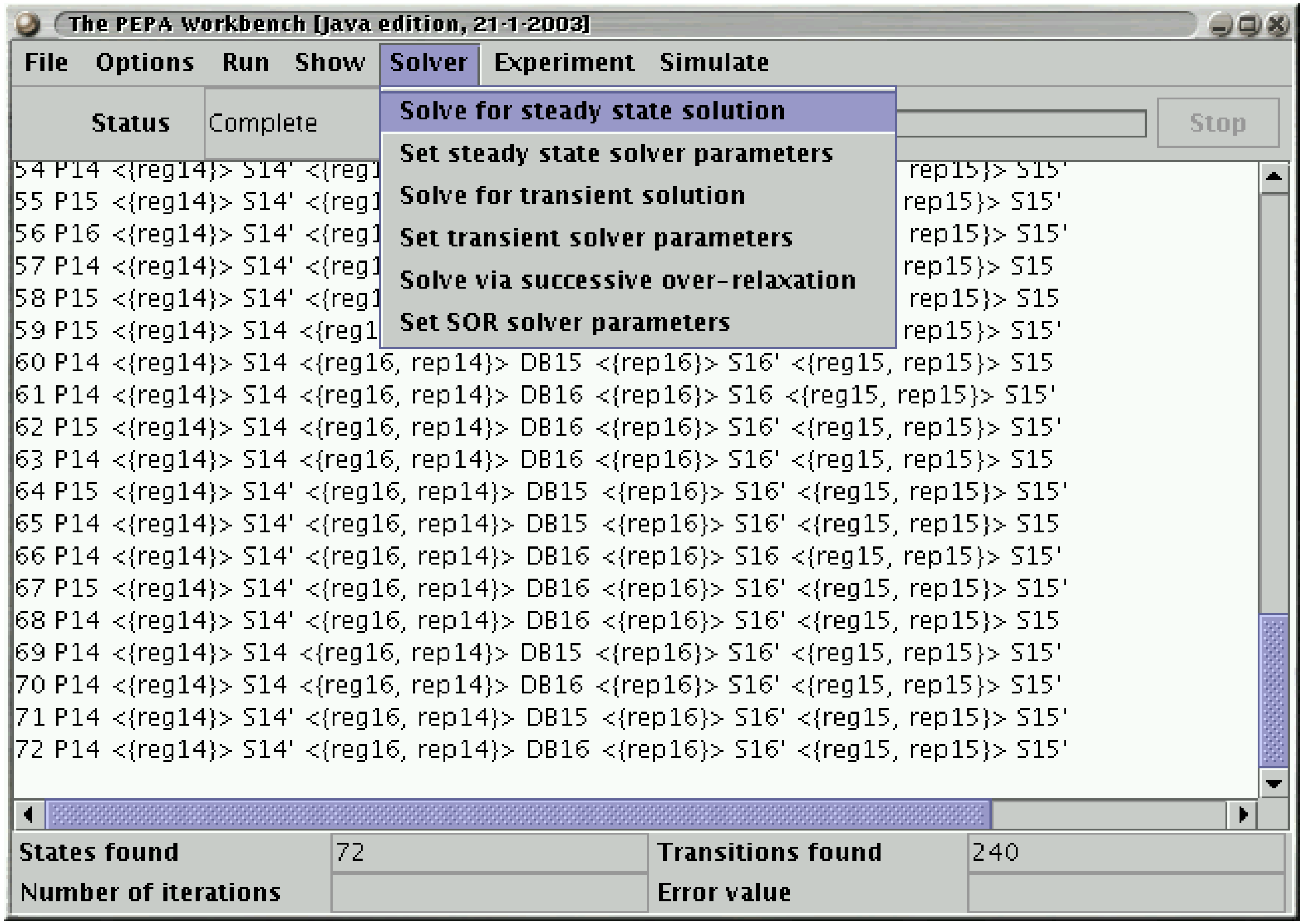
Beyond ML

- With the ML edition of the PEPA Workbench it was possible to solve small models using exterior solvers such as Maple and Matlab.

- However, users of the workbench wanted to make more detailed models (with larger state spaces).

- The ML edition of the PEPA Workbench could not solve Robert Holton's robotic workcell model efficiently enough so we interfaced it with an external solver written in C.

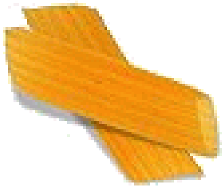
- Other users wanted to run the workbench on Solaris, Windows and Linux machines so we ported the Workbench and the solver to Java.



Status: Complete

```
54 P14 <{reg14}> S14' <{reg1                                    rep15}> S15'
55 P15 <{reg14}> S14' <{reg1                                    rep15}> S15'
56 P16 <{reg14}> S14' <{reg1                                    rep15}> S15'
57 P14 <{reg14}> S14' <{reg1                                    rep15}> S15
58 P15 <{reg14}> S14' <{reg1                                    rep15}> S15
59 P15 <{reg14}> S14 <{reg1                                     rep15}> S15'
60 P14 <{reg14}> S14 <{reg16, rep14}> DB15 <{rep16}> S16' <{reg15, rep15}> S15
61 P14 <{reg14}> S14 <{reg16, rep14}> DB16 <{rep16}> S16 <{reg15, rep15}> S15'
62 P15 <{reg14}> S14 <{reg16, rep14}> DB16 <{rep16}> S16' <{reg15, rep15}> S15'
63 P14 <{reg14}> S14 <{reg16, rep14}> DB16 <{rep16}> S16' <{reg15, rep15}> S15
64 P15 <{reg14}> S14' <{reg16, rep14}> DB15 <{rep16}> S16' <{reg15, rep15}> S15'
65 P14 <{reg14}> S14' <{reg16, rep14}> DB15 <{rep16}> S16' <{reg15, rep15}> S15
66 P14 <{reg14}> S14' <{reg16, rep14}> DB16 <{rep16}> S16 <{reg15, rep15}> S15'
67 P15 <{reg14}> S14' <{reg16, rep14}> DB16 <{rep16}> S16' <{reg15, rep15}> S15'
68 P14 <{reg14}> S14' <{reg16, rep14}> DB16 <{rep16}> S16' <{reg15, rep15}> S15
69 P14 <{reg14}> S14 <{reg16, rep14}> DB15 <{rep16}> S16' <{reg15, rep15}> S15'
70 P14 <{reg14}> S14 <{reg16, rep14}> DB16 <{rep16}> S16' <{reg15, rep15}> S15'
71 P14 <{reg14}> S14' <{reg16, rep14}> DB15 <{rep16}> S16' <{reg15, rep15}> S15'
72 P14 <{reg14}> S14' <{reg16, rep14}> DB16 <{rep16}> S16' <{reg15, rep15}> S15'
```

States found	72	Transitions found	240
Number of iterations		Error value	



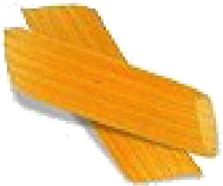
Beyond the PEPA Workbench

- Graham Clark had extended the ML edition of the workbench and developed the Java edition of the workbench from his Peparoni simulator for PEPA.
- He then implemented an editor for PEPA in the Möbius multi-paradigm modelling framework, extending PEPA to PEPA$_k$ with guards and parameters.

```
Consume[a,b] =    [a > 0] => (outa, ar).Consume[a-1,b]
 + [(b > 0) && (a == 0)] => (outb, br).Consume[a,b-1];

Breakdown = (outa, T).Breakdown
    + (fail,fr).(recover,RecoverRate).Breakdown ;

System = Consume[0,0] <outa> Breakdown;
```



Beyond the PEPA Workbench

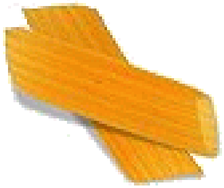
- Graham Clark had extended the ML edition of the workbench and developed the Java edition of the workbench from his Peparoni simulator for PEPA.

- He then implemented an editor for PEPA in the Möbius multi-paradigm modelling framework, extending PEPA to PEPA$_k$ with guards and parameters.

```
Consume[a,b] =    [a > 0] => (outa, ar).Consume[a-1,b]
 + [(b > 0) && (a == 0)] => (outb, br).Consume[a,b-1];

Breakdown = (outa, T).Breakdown
    + (fail,fr).(recover,RecoverRate).Breakdown ;

System = Consume[0,0] <outa> Breakdown;
```



Beyond the PEPA Workbench

- Graham Clark had extended the ML edition of the workbench and developed the Java edition of the workbench from his Peparoni simulator for PEPA.

- He then implemented an editor for PEPA in the Möbius multi-paradigm modelling framework, extending PEPA to PEPA$_k$ with guards and parameters.

```
Consume[a,b] =    [a > 0] => (outa, ar).Consume[a-1,b]
 + [(b > 0) && (a == 0)] => (outb, br).Consume[a,b-1];

Breakdown = (outa, T).Breakdown
    + (fail,fr).(recover,RecoverRate).Breakdown ;

System = Consume[0,0] <outa> Breakdown;
```

Beyond the PEPA Workbench

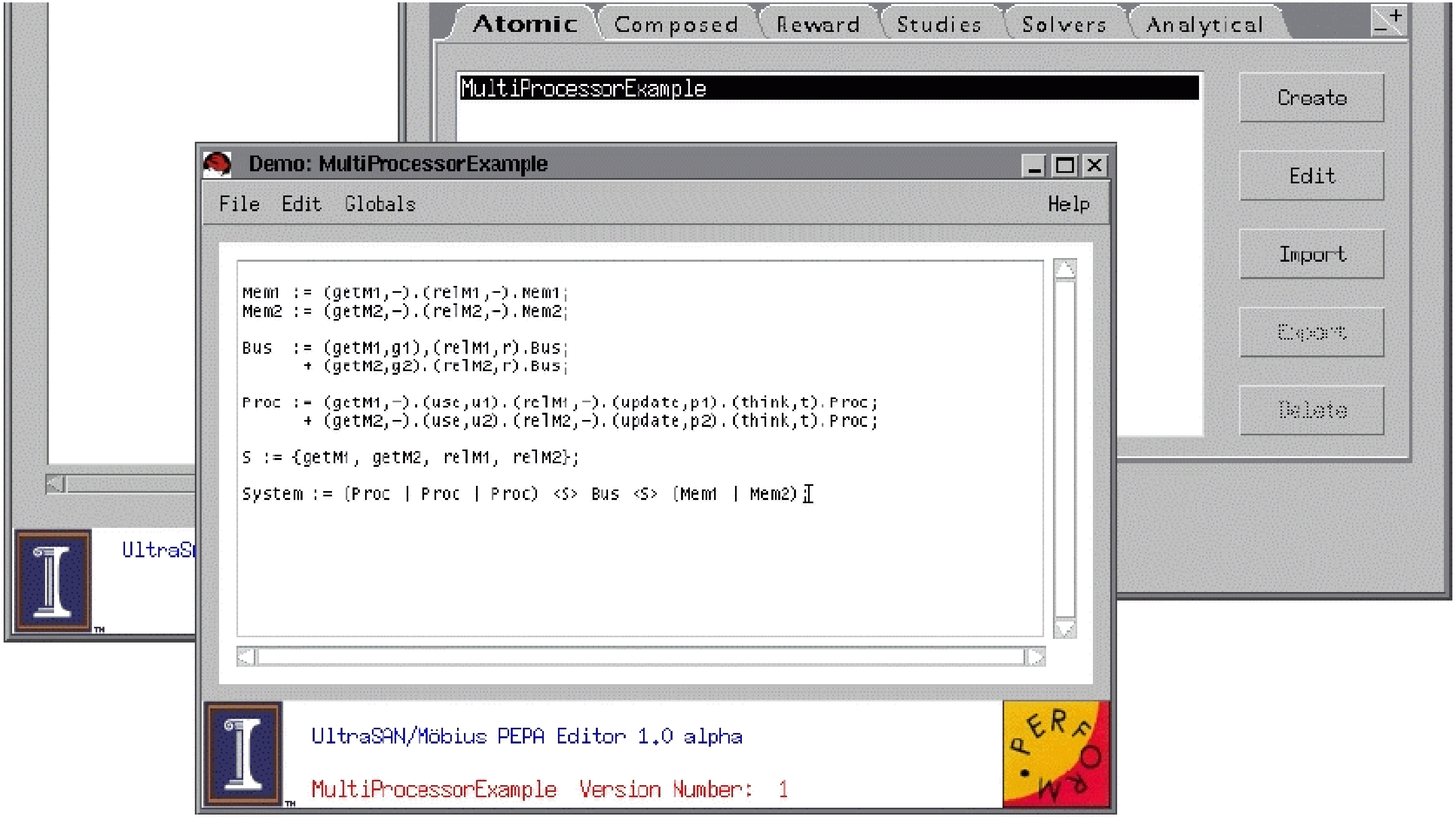
- Graham Clark had extended the ML edition of the workbench and developed the Java edition of the workbench from his Peparoni simulator for PEPA.

- He then implemented an editor for PEPA in the Möbius multi-paradigm modelling framework, extending PEPA to PEPA$_k$ with guards and parameters.

```
Consume[a,b] =    [a > 0] => (outa, ar).Consume[a-1,b]
 + [(b > 0) && (a == 0)] => (outb, br).Consume[a,b-1];

Breakdown = (outa, T).Breakdown
    + (fail,fr).(recover,RecoverRate).Breakdown ;

System = Consume[0,0] <outa> Breakdown;
```



```
Mem1 := (getM1,-).(relM1,-).Mem1;
Mem2 := (getM2,-).(relM2,-).Mem2;

Bus  := (getM1,g1),(relM1,r).Bus;
      + (getM2,g2).(relM2,r).Bus;

Proc := (getM1,-).(use,u1).(relM1,-).(update,p1).(think,t).Proc;
      + (getM2,-).(use,u2).(relM2,-).(update,p2).(think,t).Proc;

S := {getM1, getM2, relM1, relM2};

System := (Proc | Proc | Proc) <S> Bus <S> (Mem1 | Mem2);
```

UltraSAN/Möbius PEPA Editor 1.0 alpha

MultiProcessorExample Version Number: 1

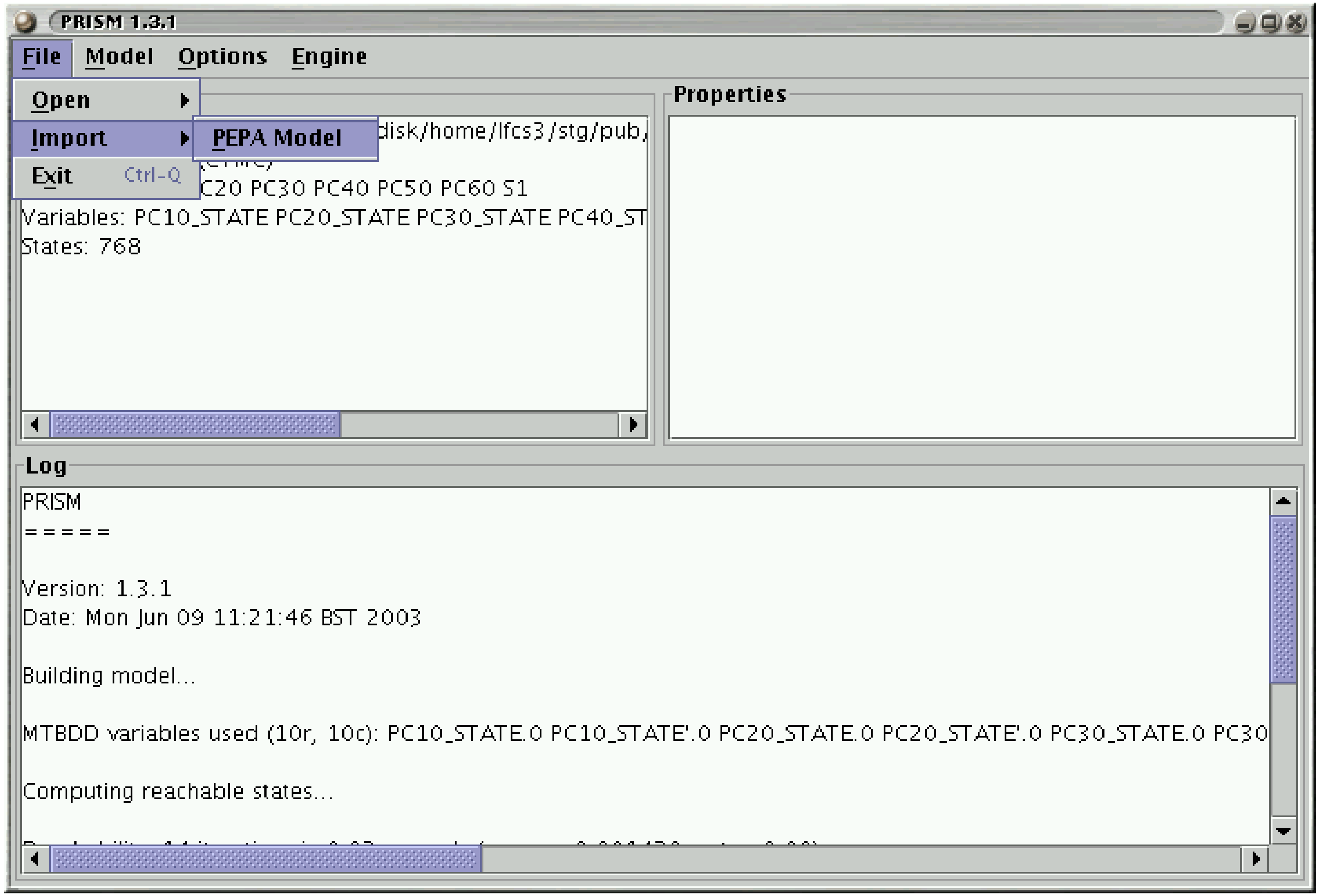
PEPA and PRISM

- PRISM is a probabilistic model checker which supports modelling in DTMCs, CTMCs and MDPs with PTCL and CSL model checking.

- The matrix storing the state space of the system is expressed as an MTBDD built using the CUDD package.

- Support for the PEPA language in PRISM was provided in two steps:

  1. extending the PRISM input language with a new **system** construct providing the PEPA composition operators for synchronisation over activity sets and hiding; and
  2. compiling the PEPA language into the extended PRISM language.



```
disk/home/lfcs3/stg/pub/
C20 PC30 PC40 PC50 PC60 S1
Variables: PC10_STATE PC20_STATE PC30_STATE PC40_ST
States: 768
```

```
PRISM
=====

Version: 1.3.1
Date: Mon Jun 09 11:21:46 BST 2003

Building model...

MTBDD variables used (10r, 10c): PC10_STATE.0 PC10_STATE'.0 PC20_STATE.0 PC20_STATE'.0 PC30_STATE.0 PC30

Computing reachable states...
```

PEPA modelling with PRISM

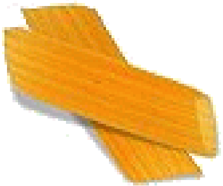
- PEPA modelling with PRISM has proved to be very effective in practice. The largest PEPA model so far solved has been solved with PRISM.

- However, there are a number of places where the user needs to understand the tool chain thoroughly:
  - The PEPA-to-PRISM compiler rejects (valid) PEPA models which use active/active synchronisation or anonymous components;
  - The compiler can fail during compilation with Java stack overflow;
  - PRISM can reject models which the PEPA-to-PRISM compiler outputs;
  - The CUDD package can fail with out-of-memory errors and need to be reconfigured.

PEPA and IPC/Dnamaca

The Imperial PEPA compiler (IPC) compiles PEPA models into Petri nets which are solved with the Dnamaca solver. Dnamaca provides a number of numerical solvers and outperforms PRISM on small PEPA models.

```
                          \transition{P1_start} {
                              %% PEPA action type { start }
                              \condition{ P1 > 0 }
                              \action {
                                  next -> P1 = P1 - 1;
P1 = (start, r1).P2;  =           next -> P2 = P2 + 1;
                              }
                              \priority{1}
                              \rate{ PEPA_r1 }
                          }
```

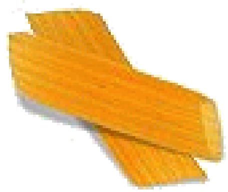


Synchronisation in Dnamaca

Suppose that two copies of P synchronise on the run activity.

```
P2 = (run, r2).P3; =
```

```
\transition{P2_run__P2_1_run} {
    %% PEPA action type { run }
    \condition{ P2 > 0 && P2_1 > 0 }
    \action {
        next -> P2_1 = P2_1 - 1;
        next -> P3_1 = P3_1 + 1;
        next -> P2 = P2 - 1;
        next -> P3 = P3 + 1;
    }
    \priority{1}
    \rate{ PEPA_r2 }
}
```



PEPA modelling with IPC and Dnamaca

- More of the PEPA language is supported by IPC/Dnamaca than by PRISM. Active/active synchronisation and anonymous components are supported.

- However, there are still a number of places where the user needs to understand the tool chain thoroughly:
  - The IPC compiler can fail during compilation with Haskell memory exhaustion;
  - Dnamaca can reject models which IPC outputs; and
  - Dnamaca's numerical procedures can fail to converge.

Dnamaca features and PEPA extensions

- Because Dnamaca supports non-Markovian modelling, beyond the models which are expressible in PEPA, it would be possible to support PEPA extensions with Dnamaca:
  - PEPA$_k$ guards and parameters; *[Clark, Sanders, '01]*
  - Weighted (WSCCS-style) PEPA; *[Bradley, '02]*
  - PEPA nets with priorities; *[Gilmore, Hillston, Ribaudo, Kloul, '03]*
  - Semi-Markov PEPA; *[Bradley, '03]*
  - . . .

PEPA nets

- PEPA nets are Petri nets with PEPA tokens. An example token is

$$\begin{aligned}
\mathsf{Agent} &\stackrel{\mathit{def}}{=} (\mathbf{go}, \lambda).\mathsf{Agent}' \\
\mathsf{Agent}' &\stackrel{\mathit{def}}{=} (\mathsf{interrogate}, \mathsf{r_i}).\mathsf{Agent}'' \\
\mathsf{Agent}'' &\stackrel{\mathit{def}}{=} (\mathbf{return}, \mu).\mathsf{Agent}''' \\
\mathsf{Agent}''' &\stackrel{\mathit{def}}{=} (\mathsf{dump}, \mathsf{r_d}).\mathsf{Agent}
\end{aligned}$$

  go and **return** are *firings* of the PEPA net. interrogate and dump are local transitions.

- A PEPA net can be processed with the PEPA Workbench for PEPA nets or compiled to PEPA using the PEPA net compiler.

The PEPA nets compiler

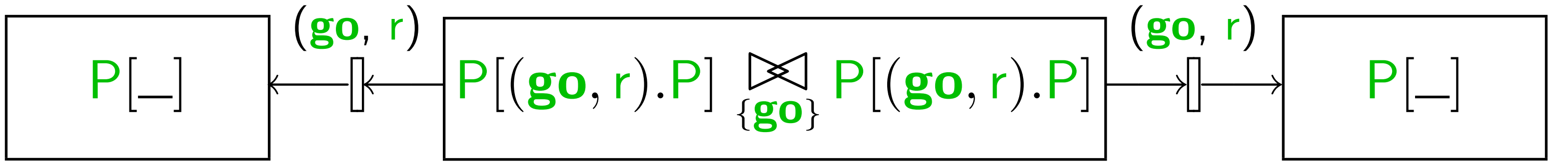
- The PEPA nets compiler compiles out token movement.

$$
\begin{aligned}
\mathsf{Agent\_\_at\_\_P2} &\stackrel{\mathit{def}}{=} (\mathsf{go\_\_to\_\_P1}, \lambda).\mathsf{Agent'\_\_at\_\_P1} \\
&\phantom{\stackrel{\mathit{def}}{=}}\!\!\!\!+ (\mathsf{go\_\_to\_\_P3}, \lambda).\mathsf{Agent'\_\_at\_\_P3} \\
\mathsf{Agent'\_\_at\_\_P1} &\stackrel{\mathit{def}}{=} (\mathsf{interrogate\_\_at\_\_P1}, \mathsf{r_i}).\mathsf{Agent''\_\_at\_\_P1} \\
\mathsf{Agent'\_\_at\_\_P3} &\stackrel{\mathit{def}}{=} (\mathsf{interrogate\_\_at\_\_P3}, \mathsf{r_i}).\mathsf{Agent''\_\_at\_\_P3} \\
\mathsf{Agent''\_\_at\_\_P1} &\stackrel{\mathit{def}}{=} (\mathsf{return\_\_to\_\_P2}, \mu).\mathsf{Agent'''\_\_at\_\_P2} \\
\mathsf{Agent''\_\_at\_\_P3} &\stackrel{\mathit{def}}{=} (\mathsf{return\_\_to\_\_P2}, \mu).\mathsf{Agent'''\_\_at\_\_P2} \\
\mathsf{Agent'''\_\_at\_\_P2} &\stackrel{\mathit{def}}{=} (\mathsf{dump\_\_at\_\_P2}, \mathsf{r_d}).\mathsf{Agent\_\_at\_\_P2}
\end{aligned}
$$

- (Strictly speaking, tokens must specify their cells within places. Different cells at the same place fall under different synchronisation sets.)

Loss of expressivity

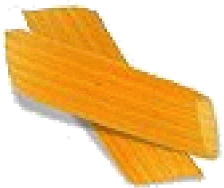
- In PEPA nets it is possible for tokens to synchronise on their exit actions from a place:



$P[\_]$ $\quad (\mathbf{go}, r) \quad$ $P[(\mathbf{go}, r).P] \underset{\{\mathbf{go}\}}{\bowtie} P[(\mathbf{go}, r).P]$ $\quad (\mathbf{go}, r) \quad$ $P[\_]$

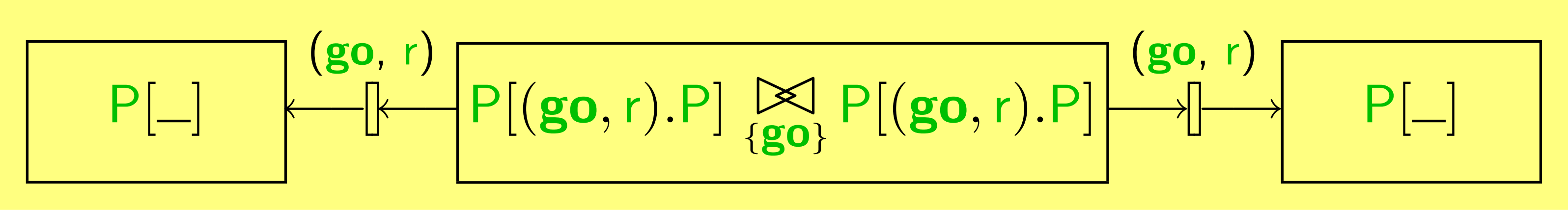
- The PEPA nets compiler cannot compile this idiom because the two tokens must go to different cells—cells can only contain a single token—and their exit activities must specify the destination cell. Therefore these activity renamings are distinct and so synchronisation is not possible.

- We consider this to be a small loss of expressivity.



Loss of expressivity

- In PEPA nets it is possible for tokens to synchronise on their exit actions from a place:

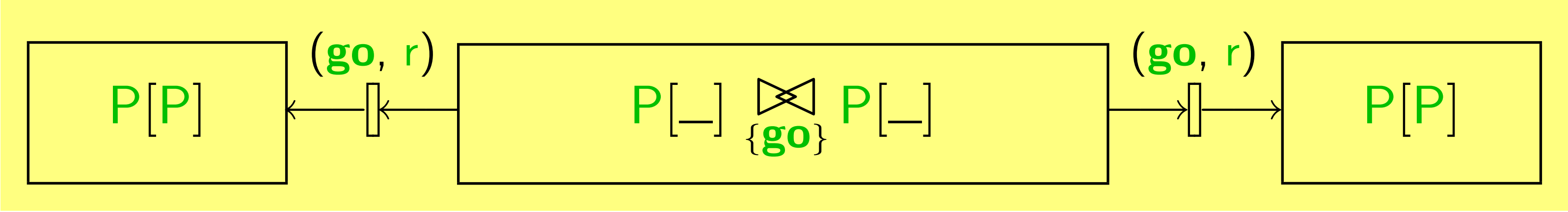


- The PEPA nets compiler cannot compile this idiom because the two tokens must go to different cells—cells can only contain a single token—and their exit activities must specify the destination cell. Therefore these activity renamings are distinct and so synchronisation is not possible.

- We consider this to be a small loss of expressivity.

Loss of expressivity

- In PEPA nets it is possible for tokens to synchronise on their exit actions from a place:

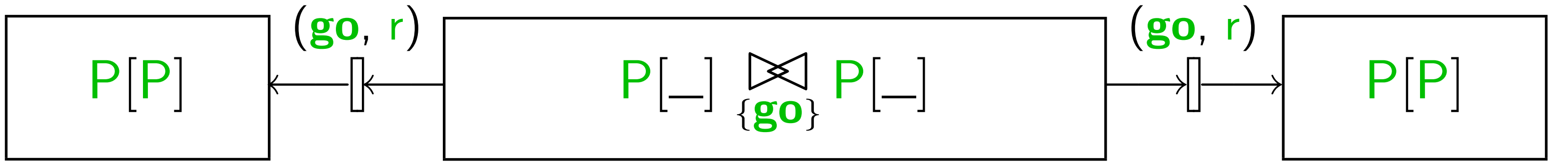


- The PEPA nets compiler cannot compile this idiom because the two tokens must go to different cells—cells can only contain a single token—and their exit activities must specify the destination cell. Therefore these activity renamings are distinct and so synchronisation is not possible.

- We consider this to be a small loss of expressivity.

Loss of expressivity

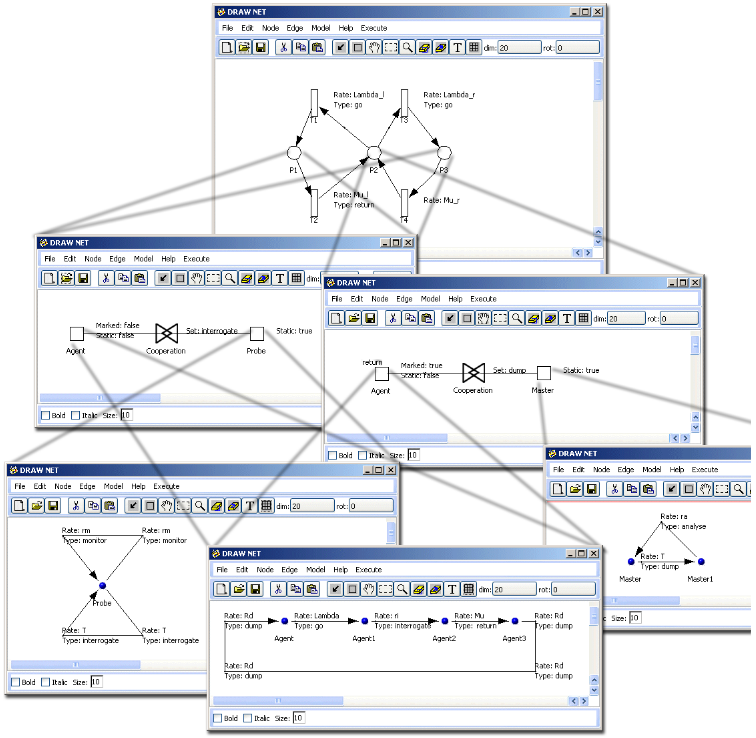
- In PEPA nets it is possible for tokens to synchronise on their exit actions from a place:

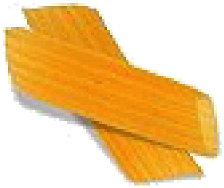


- The PEPA nets compiler cannot compile this idiom because the two tokens must go to different cells—cells can only contain a single token—and their exit activities must specify the destination cell. Therefore these activity renamings are distinct and so synchronisation is not possible.

- We consider this to be a small loss of expressivity.

PEPA nets in DrawNET





Conclusions

- Compiling PEPA and PEPA net models to other formalisms seems to be a very profitable activity.
- However, there are typically many small details in the translation which need to be taken care of.
- It is tempting to lift features of the host tool back to the PEPA level but sometimes desirable properties of the PEPA language are lost.
- It is important to strike a balance between exploiting opportunities and losing theoretical properties.