The PEPA Workbench for PEPA Nets: User's Manual

Stephen Gilmore, LFCS, Edinburgh
stg@inf.ed.ac.uk

October 6, 2005

This document describes version 0.86.1 "Little France" of the PEPA Workbench for PEPA Nets, a tool for assisting with the analysis of systems which are described by a model expressed in either the PEPA Nets modelling language or the stochastic process algebra PEPA. The definitive reference for PEPA is the CUP book "A Compositional Approach to Performance Modelling" by Jane Hillston (published 1996). The definitive reference for PEPA Nets is the paper "PEPA Nets: A structured performance modelling formalism" by Gilmore, Hillston, Kloul and Ribaudo. PEPA papers, software, and accompanying documentation such as this, can be obtained via the World-Wide Web from the address `http://www.dcs.ed.ac.uk/pepa`.

The PEPA Workbench is intended for use with a linear algebra solution tool such as those provided within the Maple and Matlab computing environments. The objective of the modelling study is to calculate the steady-state probability distribution for the system and to derive performance measures from this.

$\star$

We describe the components of the PEPA input language for the Workbench via a simple example. Consider a PEPA model with two copies of a component, P_1, executing in a pure parallel synchronisation. P_1 is a simple sequential process which undergoes a *start* activity with rate r_1 to become P_2 which runs with rate r_2 to become P_3 which goes back to P_1 via a *stop* activity with rate r_3.

$$\begin{aligned} P_1 &\stackrel{def}{=} (start, r_1).P_2 \\ P_2 &\stackrel{def}{=} (run, r_2).P_3 \\ P_3 &\stackrel{def}{=} (stop, r_3).P_1 \end{aligned}$$

The system which we study is $P_1 \underset{\emptyset}{\bowtie} P_1$ (equivalent notation for this is $P_1 \parallel P_1$).

We have made use of aspects of the mathematical syntax for PEPA in this definition. Before solving this model we first need to encode these definitions and the pure parallel synchronisation term in the ASCII syntax accepted by the Workbench. We place these definitions in the file `tiny.pepa`.

```
P1 = (start, r1).P2;
P2 = (run, r2).P3;
P3 = (stop, r3).P1;
(P1 <> P1)
```

[**Note:** The parentheses around the system definition are not optional. The Workbench will reject with a syntax error any models which do not have parentheses around the system definition. This is a change from the syntax accepted by earlier versions of the Workbench.]

This tiny example shows the form of all PEPA definitions which are accepted by the PEPA Workbench; a [non-empty] sequence of definitions of sequential components followed by a parallel composition of these to form the description of the system to be studied.

The purpose of the PEPA Workbench is to process models such as these in order to compute the state-space and the transitions of the model. Assuming that you access the PEPA Workbench by typing `pwb` then the model is processed as shown in Figure 1 where the user only typed the command `pwb -nohashing tiny.pepa` and the rest was produced by the Workbench.

```
[unix]:  pwb  -nohashing tiny.pepa
PEPA Workbench for PEPA Nets Version 0.86.1 ‘‘Little France’’
[ Identifiers will not be hashed ]
Processing input from tiny.pepa
Compiling the model
Generating the derivation graph
The model has 9 states
The model has 18 transitions
The model has 0 firings
Exiting PEPA workbench.
```

Figure 1: A sample PEPA Workbench session

The PEPA Workbench has a number of command-line options which can be described using `pwb -help` as seen in Figure 2.

```
Usage:
    pwb [options] filename.pepa
where options include
  -help               Print this message and exit
  -version            Print version number information and exit
  -silent             Run silently, produce no console messages
  -nohashing          Do not write a hash table file
  -statespaceonly     Do not write a transition matrix file
  -aggregate          Try to aggregate the model to reduce state space
  -aggregating        Provided as a synonym for -aggregate
  -viewintermediate   View intermediate transformations when aggregating
  -branching          Print branching information for each state
  -xmloutput          Write output in XML format
  -maple              Write output in Maple(TM) format
  -matlab             Write output in Matlab(TM) format
  -mathematica        Write output in Mathematica(TM) format
  -priorities         Allow use of priorities on firings
  -compile            Compile a PEPA net to an equivalent PEPA model
  -bridge             Bridge between the PEPA Workbench and Dizzy simulator
  -listing            Generate a compiler listing when compiling to PEPA
```

Figure 2: Command-line options for the PEPA Workbench

You now have three files which you did not have before: `tiny.table`, `tiny.log` and `tiny.maple`[1]. These are text files which can be read on the screen or searched using a UNIX utility such as `grep` or printed on a printer. The real content of the analysis is in `tiny.maple` so we shall consider it first. It contains the entries for the transition matrix **Q**. Depicted mathematically, **Q** would be the matrix shown below where blank entries indicate zeroes. It does not matter for a tiny example such as this one but as a general point the transition matricies of any Markov processes are best stored within a computer as *sparse* matricies.

$$\begin{pmatrix}
-2r_1 & r_1 & r_1 & & & & & & & & \\
 & -r_1-r_2 & & r_2 & r_1 & & & & & & \\
 & & -r_1-r_2 & & r_1 & r_2 & & & & & \\
r_3 & & & -r_1-r_3 & & & & & r_1 & & \\
 & & & & -2r_2 & & r_2 & & r_2 & & \\
r_3 & & & & & -r_1-r_3 & r_1 & & & & \\
 & r_3 & & & & & -r_2-r_3 & & & r_2 & \\
 & & r_3 & & & & & -r_2-r_3 & r_2 & & \\
 & & & r_3 & & r_3 & & & & -2r_3 &
\end{pmatrix}$$

The `tiny.maple` file contains this matrix information in a form suitable for consumption by a solution tool such as Matlab or Maple. When using the Matlab version of the Workbench the transitions file would contain this.

```
Q(1,3) = Q(1,3) + r1;
Q(1,1) = Q(1,1) - r1; % t1 --start,r1-> t3
Q(1,2) = Q(1,2) + r1;
Q(1,1) = Q(1,1) - r1; % t1 --start,r1-> t2
...
```

When using the Maple version of the Workbench the transitions file would contain this.

```
Q[1,3] := Q[1,3] + r1:
Q[1,1] := Q[1,1] - r1:  # t1 --start,r1-> t3
Q[1,2] := Q[1,2] + r1:
Q[1,1] := Q[1,1] - r1:  # t1 --start,r1-> t2
...
```

[**Note:** The letter "t" is taken from the first character of the filename ("tiny.pepa" in this case). If you wish to make a detailed comparison of the state spaces generated by two separate PEPA models then it is helpful to store them in files with different first characters in the name (so that, for example, one state space is named s1, s2, and s3 and the other is named t1, t2, and t3).]

The essence of the two presentations is the same. A square sparse matrix [the sparse element is zero] is modified by a sequence of assignments which record the rate at which a transition can move the system from one state to another. Explanatory comments, ignored by the solution package, give the identifier of the transition and make clear the direction of the state change. The comments are stating this information: $3 \xleftarrow{a,r_1} 1 \xrightarrow{a,r_1} 2$. This information is useful to the modeller when examining the state space to detect symmetries within the PEPA model or when 'debugging' a faulty model.

$\star$

[1]The name of this file is instead `tiny.m` for the Matlab version of the PEPA Workbench.

The states of the model have been associated with numbers for use as the indices for the transition matrix. Now we need to understand which state has which number. For this information we look at both the files for the state table `tiny.table` and (unless you specify `-nohashing`) for the associated hash table `tiny.hash`. These contain the information shown in Figure 3. The state table shows the assignment of state numbers to states, identified in terms of their hash table identifier equivalents.

The state table			**The hash table**		
1	$\mapsto$	$a \parallel a$	P_1	$\mapsto$	a
2	$\mapsto$	$d \parallel a$	P_2	$\mapsto$	d
3	$\mapsto$	$a \parallel d$	P_3	$\mapsto$	g
4	$\mapsto$	$g \parallel a$	r_1	$\mapsto$	c
5	$\mapsto$	$d \parallel d$	r_2	$\mapsto$	f
6	$\mapsto$	$a \parallel g$	r_3	$\mapsto$	i
7	$\mapsto$	$d \parallel g$	*run*	$\mapsto$	e
8	$\mapsto$	$g \parallel d$	*start*	$\mapsto$	b
9	$\mapsto$	$g \parallel g$	*stop*	$\mapsto$	h

Figure 3: State table and hash table for the tiny model.

Why have a state table? The modeller using the Workbench needs to know which states are which because certain states will be distinguished success states or completion states or utilisation states and the experimentation upon the model proceeds by considering the effect on the model of changes of relative rate.

Why have a hash table? All of the limiting problems with the use of the PEPA Workbench when processing large PEPA models are to do with space requirements either in terms of memory usage or disk usage. In generating the state space of a model the Workbench may exhaust the main memory capacity and fail. In comparison the time taken to process a model is of lesser importance because a modeller could simply run the model for longer. Thus when we attempt to improve the PEPA Workbench it is always by seeking to reduce the memory usage [perhaps at the cost of some run-time penalty]. Problems related to state-space size are a concern for all Markov modelling tools.

Notice that in the hash table no letters which were used as identifiers in the PEPA input are re-used as internal identifiers within the Workbench. This will not cause a difficulty if it happens but we take this opportunity to point out that—as with all model development—the user should choose meaningful identifiers for the activities, rates and components of a PEPA model.

$\star$

At this point the PEPA Workbench has done its work and it is now time to use the available solution tool to solve the model to find the steady state probability distribution. Performance measures can be calculated from this distribution. For the particular 9×9 matrix generated by the tiny example shown here it is as easy to solve symbolically using Maple (see Appendix B) as it is to solve numerically using Matlab (see Appendix A). For examples of even moderate size seeking symbolic solutions becomes impractical.

PEPA grammar

We now present the input grammar of the ASCII syntax for the PEPA language in EBNF notation.

program	::=	*declaration*$^+$ `(` *composition* `)`	
declaration	::=	[`#`] *id* `=` *seq_component* `;`	
seq_component	::=	*seq_component* `/ {` [*idseq*] `}`	hiding
	|	*seq_component* `+` *seq_component*	choice
	|	`(` *id*`,` *rate* `)` `.` *seq_component*	prefixing
	|	*id*	variable
	|	`(` *seq_component* `)`	grouping
composition	::=	*seq_component* `<` [*idseq*] `>` *seq_component*	
	|	*seq_component* `<` [*idseq*] `>` *composition*	
	|	`(` *composition* `)`	
idseq	::=	*id*	
	|	*id* `,` *idseq*	
rate	::=	*id*	
	|	*int*	
	|	`infty`	
id	::=	alphanumeric sequence	
int	::=	unsigned numeric sequence	

In addition LaTeX-style comments are supported, that is, the PEPA Workbench ignores any characters which follow a percent sign on a line. Also, Java-style single-line comments are supported, beginning with the sequence `//`.

The form of the component which follows the declaration sequence should be a parallel composition of (a subset of) the components which have been declared in the declaration sequence. Note that parentheses are mandatory around this component.

As noted above, a reserved word is used to represent the infinity symbol which denotes that a component is passive with respect to this action. The reserved word is `infty` [as in TeX and LaTeX] where the symbol $\top$ is used in the mathematical syntax for PEPA.

By convention, PEPA components begin with an uppercase letter whereas activity names begin with a lowercase letter.

Example sequential components: If `P` and `Q` are the only components declared in the declaration sequence in the PEPA program then the following are legal component expressions which could appear in later declarations.

`P / {a}`	Hiding any `a` action of `P`
`P + Q`	Choice between `P` or `Q`
`(a, r1).P`	Prefixing `P` by `a` with rate `r1`
`(a, 12).P`	Prefixing `P` by `a` with rate 12
`(a, infty).P`	Prefixing `P` by `a` performed passively

Sequential components non-examples: If `P` and `Q` are the only components declared in the declaration sequence in the PEPA program then the following are illegal component expressions for the reason given.

`R`	Variable not declared
`P \ {a}`	The hiding symbol is the division sign
`P / a`	Set brackets are not optional
`P <> Q`	No use of parallel in sequential components
`P.Q`	No sequential composition in PEPA
`(a, r1) P`	Omitted full stop
`(a, 12.0).P`	Real number constants not allowed
`(a, 5 / 2).P`	No arithmetic operations; only constants
`P[b/a]`	No renaming in PEPA

Example parallel compositions: If `P` and `Q` are the only components declared in the declaration sequence in the PEPA program then the following are legal parallel compositions which could appear after the declarations.

`(P <> Q)`	`P` and `Q` proceed independently in parallel
`(P <a, b> Q)`	`P` and `Q` synchronise on `a` and `b`
`((P <> P) <a, b> (Q <> Q))`	Copies of `P` and `Q` synchronise on `a` and `b`

Parallel compositions non-examples: If `P` and `Q` are the only components declared in the declaration sequence in the PEPA program then the following are illegal parallel compositions for the reason given.

`P`	Composition not enclosed in parentheses
`(R)`	Variable not declared
`(P | Q)`	Pure parallel composition symbol is `<>`
`(P < a b > Q)`	Omitted comma
`((a, r1).P)`	No sequential operations in the composition

Modelling with PEPA Nets

The following input file describes a PEPA Net. The model description has several parts. First, components as described just as they would be in PEPA. Then places in the net are specified, including cells which can contain a PEPA component (circulating around the net). The initial marking is specified at this stage. Some places are empty, but every place suggests the type of component which it should contain. (For example "`Agent[_]`" denotes a currently empty place, which could in the future contain an agent.) Some cells already have a PEPA component in one of its local states ("`Agent[Agent]`" is an agent-type cell which contains an agent in its initial state). Arcs from place to place describe the structure of the net. Arcs are labelled with an activity name and a rate. For example, "`P2 -(go, lambda)-> P3`" denotes that there is an arc from place `P2` to place `P3` which can be crossed by a component which performs the `go` activity. When the component performs that activity it will move from place `P2` to place `P3` and experience any change of state which it would experience through performing that activity. (For example, transitioning from the `Agent` state to the `Agent1` state.)

```
% This is a sample input file for the PEPA Workbench for PEPA Nets

% Components
Agent = (go, lambda).Agent1;
Agent1 = (interrogate, r_i).Agent2;
Agent2 = (return, mu).Agent3;
Agent3 = (dump, r_d).Agent;

Master = (dump, infty).Master1;
Master1 = (analyse, r_a).Master;

Probe = (monitor, r_m).Probe
      + (interrogate, infty).Probe;

% Places in the net
P1 = Agent[_] <interrogate> Probe;
P2 = Agent[Agent] <dump> Master;
P3 = Agent[_] <interrogate> Probe;

% Arcs
P2 -(go, lambda)-> P3;
P3 -(return, mu)-> P2;

P2 -(go, lambda)-> P1;
P1 -(return, mu)-> P2;

% The initial marking
(P1, P2, P3)
```

Experimenting with a model

Finding the equilibrium probability distribution for a model is an important first step in investigating its behaviour. Once this has been achieved the modeller can conduct a series of experiments to investigate the model thoroughly. The series of experiments which are undertaken will vary from model to model but they commonly involve selecting an interesting subset of the state space and finding the probability of being in those states. An activity which must be performed in doing this part of the experimentation is finding the numbers which are associated with the states which are of interest. We provide another tool to assist with this: the tool is called the PEPA State Finder.

The modeller first describes some states of interest through the use of a simple pattern language with stars for wild cards and vertical bars for separators between model components. Returning to our tiny example, a modeller interested in testing how often either component in the tiny example was component `P1` would prepare a file called `tiny.psf` with the following contents.

```
% We are checking for P1 in either case
test: P1|*
test: *|P1
```

The identifier `test` gives the name of the function and after the colon comes the pattern of interest. Assuming that you access the PEPA State Finder by typing `psf` then the function is processed as shown in Figure 3 where the user only typed the `psf` command, the model name and the function name and the rest was produced by the tool.

```
[unix]:  psf
PEPA State Finder [Version 0.06, solver, 2-12-1997]
PEPA model name:  tiny
PSF file name:  tiny
Function ``test'' written to file ``test.fun''.
Exiting PEPA State Finder.
```

Figure 4: A sample PEPA State Finder session

The function which was generated by this is shown in Appendix C. For this tiny example the function could easily have been created by hand since the model has a very regular structure and only nine states but with larger models the usefulness of the PEPA State Finder becomes clearer. Notice that here the tool correctly avoids counting the state $P_1 \parallel P_1$ twice. This would be a mistake which could easily be made otherwise.

The PEPA State Finder must be run after running the PEPA Workbench on a model because it searches through the table file which the Workbench produces.

Limitations

The PEPA Workbench has a number of limitations.

- The Workbench implements Cyclic PEPA only (see [Hillston, 1996] for the definition of this term). These are the only PEPA models for which steady-state probability distributions can be calculated. This is a consequence of the language definition and will not be changed.
- The Workbench does not implement the apparent rates of PEPA. All synchronisations must be between one active component and one passive component. This is an error which should be corrected in later versions of the PEPA Workbench.

Appendix A: A sample MATLAB session

In this appendix the nine state model is solved for the particular case when $r_1 = r_2 = r_3 = 2.0$. Since all of the transitions proceed at the same rate, and the model is symmetric, all of the states are equally likely, with calculated probability $\frac{1}{9} \approx 0.1111$.

```
[unix]: matlab -nodesktop -nosplash -nojvm

                    < M A T L A B >
            Copyright 1984-2005 The MathWorks, Inc.
            Version 7.0.4.352 (R14) Service Pack 2
                    January 29, 2005


  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>> Q=sparse(9,9);
>> r1=2.0;
>> r2=2.0;
>> r3=2.0;
>> tiny;
>> for i=1:9
     Q(i,9) = 1.0;
   end
>> b=zeros(9,1);
>> b(9) = 1.0;
>> QT = Q';
>> P = QT\b

P =

    0.1111
    0.1111
    0.1111
    0.1111
    0.1111
    0.1111
    0.1111
    0.1111
    0.1111

>> quit
```

Appendix B: A sample Maple session

In this appendix the nine state model is solved symbolically for all values of r_1, r_2 and r_3.

```
[unix]: maple
   |\^/|     Maple 9 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2003
 \  MAPLE  / All rights reserved. Maple is a trademark of
 <____ ____> Waterloo Maple Inc.
      |      Type ? for help.
> with(linalg):
Warning, the protected names norm and trace have been redefined and unprotected
> Q := array(sparse,1..9,1..9):
> read `tiny.maple`:
> b := array(sparse,1..9):
> for i to 9 do
      Q[i,9] := 1.0
  od:
> b[9] := 1:
> QT := transpose(Q):
> P := linsolve(QT,b);

              2   2    2          2             2       2   2
             r3  r2   r3  r2 r1  r3  r2 r1  r3 r2  r1  r3  r1
     P := [ -------, ---------, ---------, ---------, -------,
              %1         %1         %1         %1         %1

              2             2          2    2   2
         r3 r2  r1  r3 r2 r1   r3 r2 r1   r1  r2
         ---------, ---------, ---------, ------- ]
             %1         %1         %1         %1

              2        2   2     2   2           2
%1 := 2. r3 r2  r1 + r3  r2  + r1  r2  + 2. r2 r1  r3 +
                  2     2   2
      2. r2 r1 r3  + r3  r1

> quit
bytes used=1962176, alloc=1638100, time=0.18
```

Appendix C: A function generated by the PEPA state finder

This is the function which was generated when the `tiny.psf` file was processed by the PEPA State Finder. The body of the `test` function adds together the probabilities for the relevant states. After a comment symbol on each line comes the description of the state. As we expected, all of the states contain a P_1 component.

```
#  Function:  test
#  Table:     tiny.table

test := proc (P)
(       0
        +  P[1] # P1|P1
        +  P[3] # P1|P2
        +  P[6] # P1|P3
        +  P[2] # P2|P1
        +  P[4] # P3|P1
)
end:

print (‘PEPA: Defined function ‘‘test’.    Usage:  test(P);‘);
```

Appendix D: Installing and running the PEPA Workbench for PEPA Nets

The PEPA Workbench for PEPA Nets is a Standard ML application. It is available both in source code form and as pre-compiled binaries for Linux and Windows. We have tested the PEPA Workbench for PEPA Nets on Linux Fedora Core 3 and Windows XP, as well as earlier versions of these operating systems. The 0.86.1 "Little France" release of the PEPA Workbench for PEPA Nets has been compiled using the MLton compiler for Standard ML, but can also be compiled using other compilers for the language. The PEPA Workbench for PEPA Nets does not need root or Administrator privileges to build and install, and the pre-compiled binaries will run from the current working directory. The Workbench can reside in any directory on the filesystem and always writes files to the directory in which the PEPA (or PEPA Net) model resides. The compiled image of the Workbench is not large (about 1 Megabyte).

D.1 Running under Windows with Cygwin

The compiled version of the PEPA Workbench for PEPA Nets is stored in `pwb.exe`. It depends on the Cygwin implementation of UNIX for Windows. Before installing the PEPA Workbench for PEPA Nets you should first install Cygwin (from `www.cygwin.com`). We have tested the PEPA Workbench for PEPA Nets with Cygwin version 1.5.18-1. The PEPA Workbench for PEPA Nets depends on the dynamically-linked library `cygwin1.dll`, and will not operate without it.

PEPA and PEPA Net source files may be stored in any folder, even if the folder contains spaces in the name as in the following example.

```
./pwb -nohashing C:/Documents\ and\ Settings/user/My\ Documents/tiny.pepa
```

D.2 Running under Linux

The compiled version of the PEPA Workbench for PEPA Nets is stored in `pwb`. It expects to access the C library `glibc` version 2.2.5 or higher. We have tested the compiled version with `glibc` version 2.3.5.

D.3 Building the PEPA Workbench for PEPA Nets from source

For other platforms for which a binary version of the PEPA Workbench for PEPA Nets is not provided first obtain an implementation of the Standard ML programming language and modify the Makefile in the source directory to compile the Workbench using your chosen Standard ML compiler. Re-compile the Workbench with `make mlton`.