

Scalable Modelling in the PEPA Eclipse Plug-in

November 12, 2012

1 Introduction

PEPA has a long history of software tool support. As we saw in the lecture note, the *PEPA Eclipse Plug-in* has been recently proposed as an implementation of the language which supports the whole model development process with a rich graphical interface. Lecture note 14 provided an overview of the main plug-in features such as static analysis and numerical solution of the underlying continuous-time Markov chain (CTMC). This lecture note is concerned with a recently added module which implements the some of the scalable analysis techniques discussed in lecture note 15. These scalable analysis techniques are based on the numerical vector form of state representation.

As a running example we consider the following PEPA model, which is very similar to one presented in earlier notes:

$$\begin{aligned} \text{Process1} &\stackrel{\text{def}}{=} (\text{use}, r).\text{Process2} \\ \text{Process2} &\stackrel{\text{def}}{=} (\text{think}, s).\text{Process1} \\ \text{CPU1} &\stackrel{\text{def}}{=} (\text{use}, r).\text{CPU2} \\ \text{CPU2} &\stackrel{\text{def}}{=} (\text{reset}, t).\text{CPU1} \end{aligned}$$

$$\text{Process1}[8] \boxtimes_{\{\text{use}\}} \text{CPU1}[4]$$

The component **Process** interposes some thinking time between uses of a **CPU**, modelled as a synchronisation over the action type **use**. The **CPU** performs some reset operation, e.g., a context switch after each use. The system equation considers eight processes and four CPUs in total. In this model, a process may use any of the available CPUs. The form of the model suitable for putting into the tool is shown in Figure 1,

Recall that the numerical vector form state representation is a vector of counting variables, each representing the number of components which exhibit a specific local behaviour. In the running example, the state vector may be denoted as follows:

$$\mathbf{x} = (x_{\text{Process1}}, x_{\text{Process2}}, x_{\text{CPU1}}, x_{\text{CPU2}}),$$

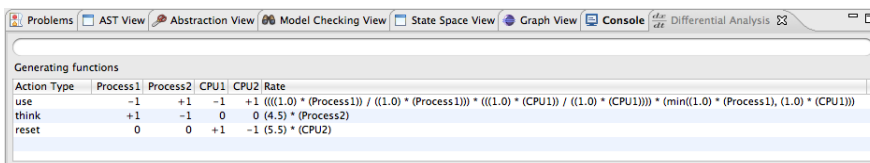
and hence the initial state is $(8, 0, 4, 0)$. *Generating functions*, denoted by $f_\alpha(\mathbf{x}, l)$, are used to give the rate at which an activity of type α is executed, and the change to a state

```

/* Rate declarations */
r = 1.0;
s = 4.5;
t = 5.5;
/* Sequential component Process */
Process1 = (use, r).Process2;
Process2 = (think, s).Process1;
/* Sequential component CPU */
CPU1 = (use, r).CPU2;
CPU2 = (reset, t).CPU1;
/* System equation */
Process1[8] <use> CPU1[4]

```

Figure 1: A sample PEPA model with two sequential processes.



Action Type	Process1	Process2	CPU1	CPU2	Rate
use	-1	+1	-1	+1	$\min(1.0 * (\text{Process1}), 1.0 * (\text{CPU1}))$
think	+1	-1	0	0	$4.5 * (\text{Process2})$
reset	0	0	+1	-1	$5.5 * (\text{CPU2})$

Figure 2: Differential Analysis View for the model in Figure 1.

due to its execution through the vector l . For instance, the shared action `use` is captured by the function

$$f_{\text{use}}(\mathbf{x}, (-1, 1, -1, 1)) = \min(r x_{\text{Process1}}, r x_{\text{CPU1}}),$$

which says that `use` decreases the population counts of `Process1` and `CPU1` and, correspondingly, increases the population counts of `Process2` and `CPU2` at a rate which is dependent upon the current state. Just as in the case of deriving the Markov process underlying the syntactic state representation of the PEPA model, the derivation of the generator functions is based on a formal structured operational semantics. For instance, the following transition in the CTMC for the initial state may be then inferred:

$$(8, 0, 4, 0) \xrightarrow{\min(1.0 \times 8.0, 1.0 \times 4.0)} (8, 0, 4, 0) + (-1, 1, -1, 1) \equiv (7, 1, 3, 1).$$

Extracting generating functions from a PEPA model presents little computational challenge because it does not require the exploration of the whole state space of the CTMC. In the plug-in, the **Differential Analysis View** updates the generating functions of the currently active model whenever its contents are saved. Figure 2 shows a screen-shot with the three generating functions of our running example.

2 Scalable Model Analysis

2.1 Stochastic simulation.

The generating functions contain all the necessary information for model analysis. As discussed in Lecture note 15 they allow a straightforward application of Gillespie's stochastic

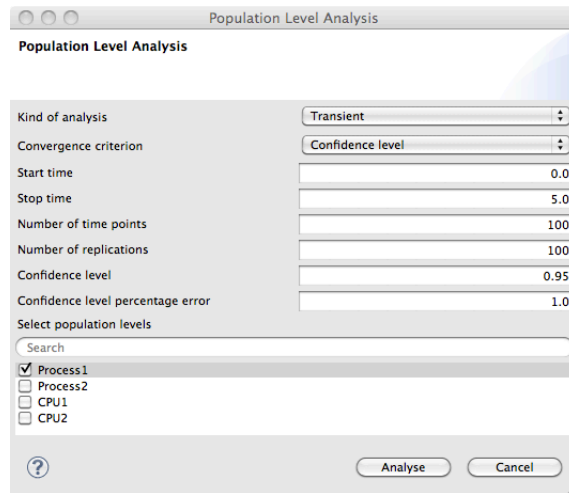


Figure 3: Stochastic simulation dialogue box.

simulation algorithm. Given a state \hat{x} , the evaluation of each of the generating functions $f_\alpha(\hat{x}, l)$ of the model gives the relative probabilities with which each action may be performed. Drawing a random number from the resulting probability density function decides which action is to be taken, and thus the corresponding target state $\hat{x} + l$. This procedure may be repeated until conditions of termination of the simulation algorithm are met.

The PEPA Eclipse Plug-in implements transient and steady-state simulation algorithms (see Figure 3). Transient simulation is based on the method of independent replications, (see Lecture note 13). The user is required to select which components of the state vector are to be kept track of, the start and stop time of the simulation, the number of equally spaced time points of interest over the simulation interval, the maximum number of replications allowed, and a desired confidence level.

Two stopping criteria may be used. The confidence-level criterion terminates the simulation when the user-specified confidence interval is within a given percentage of the statistical mean. If convergence is not reached within the maximum number of replications allowed, the results are reported together with a warning. Alternatively, the simulation may be simply stopped when the maximum number of replications is reached. In either case, the tool presents the results in a graph with errors bars giving the confidence levels obtained (see Figure 4).

Steady-state simulation is performed with the method of *batch means*. Using a similar interface to that shown in Figure 3, the user is presented with a dialogue box to set up the following parameters: length of the transient period, during which samples are discarded; confidence level desired; maximum number of batches. The duration of each batch is set to ten times the transient period.¹ At the end of a batch, the algorithm checks whether the tracked population counts have reached the desired confidence level. If the maximum number of batches is reached, the algorithm returns with a warning of potentially bad accuracy. The results are presented in a dialogue box which gives the average steady-state value and the confidence level for each tracked population count (see Figure 5a)².

¹This parameter is currently not modifiable. Future releases will expose this setting to the user.

²The lag-1 correlation is also computed as an indicator of statistical independence between adjacent batches, but details of this are beyond the scope of this course.

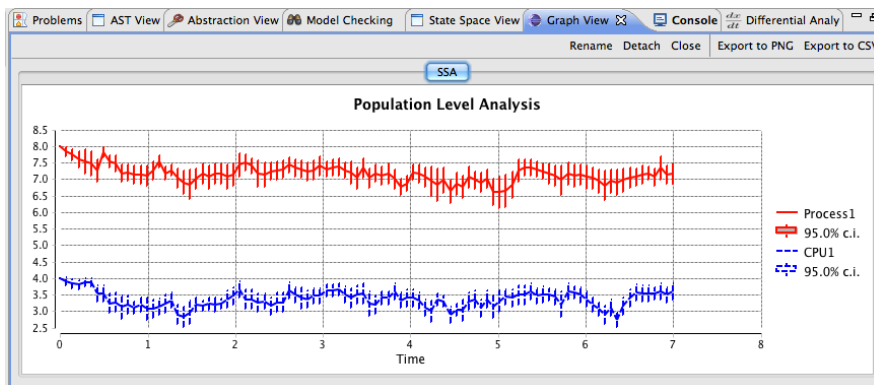


Figure 4: Results of a transient stochastic simulation.

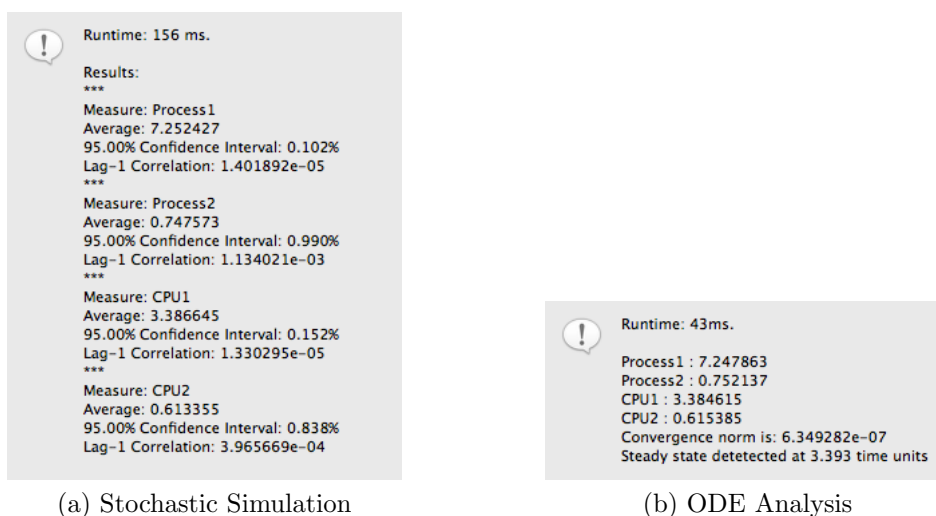


Figure 5: Steady state results displayed in dialogue boxes.

2.2 Differential Analysis.

The vector $x(t)$ of functions which is solution of the ordinary differential equation (ODE) $\frac{dx(t)}{dt} = \sum_l \sum_\alpha l f_\alpha(x(t), l)$ provides an approximating continuous trajectory of the population counts of the PEPA components as a function of time. The plug-in supports the numerical integration of the initial value problem corresponding to the model equation of the PEPA description. In the running example, the initial state of the vector $x(t) = (x_{Process_1}(t), x_{Process_2}(t), x_{CPU_1}(t), x_{CPU_2}(t))$ is $x(0) = (8, 0, 4, 0)$. The plug-in supports both transient and steady-state analysis. In either case, the user is required to specify the integration interval and the absolute and relative tolerance errors for the numerical solver, based on the `odetojava` library. For transient analysis, a mesh of time points of interest must be specified using the triple `start time, time step, stop time`. At those points the solution of the ODE will be computed and the results will be returned in the form of a graph. For steady-state analysis, the user is also required to specify a tolerance error for equilibrium detection. At each time point τ analysed during the numerical integration, if the Euclidean norm of $\frac{dx(t)}{dt}|_{t=\tau}$ is below that threshold, the model is said to have reached equilibrium, and the corresponding solution vector $x(\tau)$ is returned.

The user interface for either form of differential analysis is structurally similar to that for stochastic simulation—the top area of a dialogue box permits the set-up of the numerical integration, whereas in the bottom area the user chooses the components to visualise. As with steady-state stochastic simulation, the results are presented in a dialogue box (see Figure 5b).

Throughput Calculation. The generating functions can be interpreted as giving the throughput of an action type in each state xi of the Markov chain. For instance, if the simulation model collects samples of $\min(r x_{Process_1}, r x_{CPU_1})$ then the statistics of this random variable will give the throughput of `use`. The graphical interface of Figure 3 is partially re-used to specify the simulation settings. However, in the bottom area are listed all the action types declared in the PEPA model. A fluid approximation of throughput, which is also implemented, is given by the evaluation of the function $f_\alpha(x(t), l)$. Finally, throughput and population-count measures can be combined in order to compute average response times, using Little’s Law using the same approach as we have seen in earlier lecture notes.