10 PEPA

In this lecture note we consider another class of performance modelling languages—stochastic extensions of *process algebras*, particularly the stochastic process algebra, PEPA. Like queueing networks and stochastic Petri nets, and their variants, these formal languages can be regarded as high-level model specification languages for low-level stochastic models. As we will see, the development of SPA has been very similar to that of SPN: in both cases an untimed formalism, used for studying the correct functional behaviour of systems, is extended by associating exponential delays with actions and reachability analysis is used to construct a corresponding Markov process.

Process algebras emerged as a modelling technique for the functional analysis of concurrent systems approximately thirty five years ago. Over the last twenty five years there have been several attempts to take advantage of the attractive features of this modelling paradigm within the field of performance evaluation.

10.1 Stochastic Process Algebra

Stochastic process algebras (SPA) were first proposed as a tool for performance and dependability modelling in 1990. Compared with existing performance modelling formalisms SPA offered something new—formally defined compositionality. Queueing networks, have an inherent compositionality but this is implicit and informal. As we have seen stochastic extensions of Petri nets have a semantic model but, in general, no clear compositional structure. In the process algebra the compositionality is explicit—provided by the combinators of the language—and formal—supported by the semantics and equivalence relations of the language.

It was immediately clear that having this explicit structure within models offers benefits for model construction:

- when a system consists of interacting components, the components can each be modelled separately, and then composed to form the larger system;
- models have a clear structure and are easy to understand;
- models can be constructed systematically, by either elaboration or refinement;
- the possibility of maintaining a library of model components, supporting model reusability, is introduced.

Furthermore researchers have shown that in some cases the compositional structure in the model construction can be exploited during model analysis, meaning that a number of smaller Markov processes can be solved instead of a single large one.

10.1.1 Classical Process Algebras

Process algebras are abstract languages used for the specification and design of concurrent systems. The most widely used process algebras are Milner's Calculus of Communicating Systems (CCS) and Hoare's Communicating Sequential Processes (CSP) and the SPAs

take inspiration from both these formalisms. Models in CCS and CSP have been used extensively to establish the correct behaviour of complex systems by deriving *qualitative* properties such as *freedom from deadlock* or *livelock*.

In the process algebra approach systems are modelled as collections of entities, called *agents*, which execute atomic *actions*. These actions are the building blocks of the language and they are used to describe sequential behaviours which may run concurrently, and synchronisations or communications between them.

In order to carry out performance modelling it was necessary to quantify the delay associated with activities and the probabilities associated with choices. In PEPA, as in SPN, this is achieved by assuming that the duration of each action is governed by a random variable which is exponentially distributed. This means that there is a randomly distributed delay for each action and the choice between actions is governed by a race condition meaning that there is an implicit probabilistic choice.

10.2 Modelling in PEPA

PEPA models are described as interactions of *components*. Each component can perform a set of actions: an action $a \in \mathcal{A}ct$ is described as a pair (α, r), where $\alpha \in \mathcal{A}$ is the *type* of the action and $r \in \mathbb{R}^+$ is the parameter of the negative exponential distribution governing its duration. Whenever a process P can perform an action, an instance of the given probability distribution is sampled: the resulting number specifies how long it will take to *complete* the action. A small but powerful set of combinators is used to build up complex behaviour from simpler behaviour. The combinators are: prefix, choice, parallel composition and abstraction. We explain each of the combinators informally below.

Prefix: A component may have purely sequential behaviour, repeatedly undertaking one activity after another and eventually returning to the beginning of its behaviour. At the most basic level we say that the component $(\alpha, r).P$ must first do the activity of type α at rate r and then behaves as the component P. The symbol "." denotes the prefix, the designated first action.

A simple example is a web service within a distributed system, which can serve one request at a time. Each application requiring the web service will need to gain access to the service which will then only be made available for another application when a response has been successfully transferred.

$$WS \stackrel{def}{=} (request, \top).(serve, \mu).(respond, \top).WS$$

In some cases, as here, the rate of an action is outside the control of this component. Such actions are carried out jointly with another component, with this component playing a passive role. For example, the web service is passive with respect to the *request* action, as it cannot influence the rate at which requests arrive, and this is recorded by the distinguished symbol, $\top$ (called "top").

Choice: A choice between two possible behaviours is represented as the sum of the possibilities. For example, if we consider an application in a distributed system, a computation may have two possible outcomes: access to a locally available method is required

(with probability p_1) or access to a remote web service is necessary (with probability $p_2 = 1 - p_1$). In this example the *think* action denotes processing within the application. These alternatives are represented as shown below:

$$Appl \stackrel{def}{=} (think, p_1\lambda).(local, m).Appl + \quad (think, p_2\lambda).(request, rq).(respond, rp).Appl$$

A race condition governs the behaviour of simultaneously enabled actions so only one action will complete. The continuous nature of the probability distributions ensures that the actions cannot occur simultaneously. Thus a sum will behave as either one summand or the other. When an action has more than one possible outcome, e.g. the *think* action in the application, it is represented by a choice of separate actions, one for each possible outcome. The rates of these actions are chosen to reflect their relative probabilities, following the decomposition principle of the exponential distribution. Note that this is analogous to the result of a timed transition followed by the choice between two immediate transitions in a GSPN model.

Parallel composition: In the web service example, we have already anticipated that the application and the web service will be working together within the same system. This will require them to *cooperate* when the application needs the service offered by the web service, which is not available locally. In contrast, the local activities of the application can be carried out independently of the web service. Cooperation over given actions is reflected in the parallel composition by the *cooperation set*, $L = \{request, respond\}$ in this case. Actions in this set require the simultaneous involvement of both components. The resulting action, a *shared* action, will have the same type as the two contributing actions and a rate reflecting the rate of the action in the slowest participating component. Note that this means that the rate of a passive action will become the rate of the action it cooperates with.

If, for simplicity, we assume that the distributed system consists of just two independent applications, the system is represented as the cooperation of the applications and the web service as follows:

$$Sys_1 \stackrel{def}{=} \big(Appl \parallel Appl\big) \underset{L}{\bowtie} WS \qquad L = \{request, respond\}$$

The combinator $\parallel$ is a degenerate form of the cooperation combinator, formed when two components behave completely independently, without any cooperation between them, as in the case of the two independent applications. This pure parallel combinator can be thought of as cooperation over the empty set: $(Appl \underset{\emptyset}{\bowtie} Appl)$. When we have a number of independent copies of a component it is sometimes convenient to use a vector notation to abbreviate the presentation, e.g. $Appl[5] \equiv Appl \parallel Appl \parallel Appl \parallel Appl \parallel Appl$.

Abstraction: It is often convenient to hide some actions, making them private to the component or components involved. The duration of the actions is unaffected, but their type becomes hidden, appearing instead as the unknown type τ. Components cannot synchronise on τ. For example, as we further develop the model of the distributed system we may wish to hide the access of a application to its local method. This might lead to a new representation of the application:

$$Appl' \stackrel{def}{=} Appl/\{local\}$$

and a corresponding new representation of the system:

$$Sys_2 \stackrel{def}{=} \left(Appl' \parallel Appl'\right) \underset{L}{\bowtie} WS \qquad L = \{request, respond\}$$

Use of the hiding combinator has two implications. Firstly, it ensures that no components added to the model at a later stage can share the *local* action of the application. Secondly, private actions are deemed to have no contribution to the performance measures being calculated and this might subsequently suggest simplifications to the model.

Throughout the simple example above we have used constants such as WS to associate names with behaviours. Using recursive definitions we have been able to describe components with infinite behaviours without the use of an explicit recursion operator.

Representing the components of the system as separate components means that we can easily extend our model. Now we may want to consider a distributed system consisting of more than two applications which act independently of each other but compete for the use of web service. To enhance fault tolerance the web service may be replicated. This extension may be achieved compositionally by combining more instances of the components already described. For example, in the case of three applications and two instances of the web service we have:

$$Sys_3 \stackrel{def}{=} Appl[3] \underset{L}{\bowtie} WS[2] \qquad L = \{request, respond\}$$

10.3 Model analysis

The formality of the process algebra approach allows us to assign a precise meaning to every language expression. This implies that once we have a language description of a given system its behaviour can be deduced automatically. The meaning, or semantics, of a PEPA expression is provided by structured operational semantics rules, which associate a labelled multi-transition system with every expression in the language.

A labelled transition system $(S, T, \{\stackrel{t}{\longrightarrow} \mid t \in T\})$ consists of a set of states S, a set of transition labels T and a transition relation $\stackrel{t}{\longrightarrow} \subseteq S \times S$. For PEPA the states are the syntactic terms in the language, the transition labels are the actions ($(type, rate)$ pairs), and the transition relation is given by the semantic rules. A multi-transition relation is used because the number of instances of a transition (action) is significant since it can affect the timing behaviour of a component.

Based on the transition relation, a *derivation graph* (DG), can be associated with each language expression. This graph describes all the possible evolutions of any component and provides a useful way to reason about the behaviour of a model[1]. A certain amount of care is needed in defining the derivation graph. Consider a simple component, P, which will repeatedly carry out the action $a = (\alpha, r)$, i.e. $P \stackrel{def}{=} (\alpha, r).P$. For a classical process algebra we need only consider which actions it is possible for an agent to perform. Thus, the agent $P + P$ has the same behaviour as the agent P—both are capable of an α named action and subsequently behave as P—so these agents are considered to be equivalent. In a SPA multiple instances of an action become apparent because the duration of an action of that type will be the minimum of the corresponding random variables, i.e. the

[1]The DG is analogous to the reachability graph of a SPN.

apparent rate of the action will be the sum of the rates. Thus $P + P$ appears to carry out the first α named action at twice the rate of the agent P. Consequently the two cannot be regarded as equivalent and in the DG we use a multi-transition relation rather than a simple transition relation.

An example derivation graph is shown in Figure 25 where the DG of the PEPA model Sys_4, consisting of a single application accessing the web service, is shown. To make the correspondence between the language expressions and the nodes in the DG clearer, in the left hand part of the figure we have expanded the derivatives of the components *Appl* and WS.

$$
\begin{array}{rcl}
Appl & \stackrel{def}{=} & (think, p_1\lambda).Appl_1 \\
 & + & (think, p_2\lambda).Appl_2 \\
Appl_1 & \stackrel{def}{=} & (local, m).Appl \\
Appl_2 & \stackrel{def}{=} & (request, rq).Appl_3 \\
Appl_3 & \stackrel{def}{=} & (respond, rp).Appl \\
WS & \stackrel{def}{=} & (request, \top).WS_1 \\
WS_1 & \stackrel{def}{=} & (serve, \mu).WS_2 \\
WS_2 & \stackrel{def}{=} & (respond, \top).WS \\
Sys_4 & \stackrel{def}{=} & Appl \underset{L}{\bowtie} WS \\
L & = & \{request, respond\}
\end{array}
$$

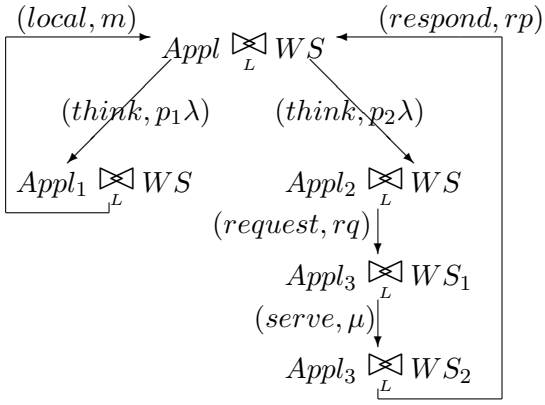

Figure 25: Derivation graph underlying Sys_4.

Looking at the DG we can derive qualitative properties of the model. In this case, for instance, we can see that the PEPA model is free from deadlock and live. Moreover, the Markov process underlying any finite PEPA component can be obtained directly from the DG: a state of the Markov process is associated with each node of the graph and the transitions between states are defined by considering the rates labelling the arcs. Since all activity durations are exponentially distributed, the total transition rate between two states will be the sum of the activity rates labelling arcs connecting the corresponding nodes in the DG. Starting from the DG of Figure 25, the derivation of the corresponding Markov process is straightforward and results in the generator matrix shown below.

$$
\boldsymbol{Q} = \begin{pmatrix}
-\lambda & p_1\lambda & p_2\lambda & 0 & 0 \\
m & -m & 0 & 0 & 0 \\
0 & 0 & -rq & rq & 0 \\
0 & 0 & 0 & -\mu & \mu \\
rp & 0 & 0 & 0 & -rp
\end{pmatrix}
$$

As with any Markov process-based technique, numerical solution of the global balance equation will lead to the steady state probability distribution.

In order to ensure that the Markov process underlying a PEPA model is ergodic, the DG of a PEPA model must be strongly connected.

Just as for SPN, performance measures such as throughput and utilisation are often derived via a *reward structure* which is defined over the Markov process. This can either be done explicitly by the modeller, or as we will see, automatically by the tool for commonly required measures. A reward structure associates a reward with each state of the model. For steady state measures, the expected value of the reward (i.e. the sum over the entire state space of (probability of a state $\times$ reward in that state)) is calculated. In a process algebra it can be easier to associate rewards with actions. In this case the reward associated with a state will be the total reward attached to the actions that the state enables. Recall that in PEPA no reward can be attached to internal, τ, actions.

10.4 A PEPA model of the PC LAN Example with 4 nodes

The model below is a PEPA representation of the PC LAN with four nodes considered in lecture note 4 (and again in lecture note 8). Each PC is represented by a separate component which, when empty (e.g. PC_{n0}) can either have an arrival or will allow the token to "walk" past ($walkon_n$). If the PC has a data packet waiting (PC_{n1}) it remains in that state until served by the token. The token moves round the LAN deciding what to do at each node on the basis of the actions offered by the corresponding PC. Thus although it looks as if it has a choice at each node, in fact only one of the actions $walkon_n$ and $serve_n$ will be enabled at any given time (these actions must be carried out in cooperation with the appropriate PC).

$$
\begin{aligned}
PC_{10} &\stackrel{def}{=} (arrive, \lambda).PC_{11} + (walkon_2, \omega).PC_{10} \\
PC_{11} &\stackrel{def}{=} (serve_1, \mu).PC_{10}
\end{aligned}
$$

$$
\begin{aligned}
PC_{20} &\stackrel{def}{=} (arrive, \lambda).PC_{21} + (walkon_3, \omega).PC_{20} \\
PC_{21} &\stackrel{def}{=} (serve_2, \mu).PC_{20}
\end{aligned}
$$

$$
\begin{aligned}
PC_{30} &\stackrel{def}{=} (arrive, \lambda).PC_{31} + (walkon_4, \omega).PC_{30} \\
PC_{31} &\stackrel{def}{=} (serve_3, \mu).PC_{30}
\end{aligned}
$$

$$
\begin{aligned}
PC_{40} &\stackrel{def}{=} (arrive, \lambda).PC_{41} + (walkon_1, \omega).PC_{40} \\
PC_{41} &\stackrel{def}{=} (serve_4, \mu).PC_{40}
\end{aligned}
$$

$$
\begin{aligned}
Token_1 &\stackrel{def}{=} (walkon_2, \omega).Token_2 + (serve_1, \mu).(walk_2, \omega).Token_2 \\
Token_2 &\stackrel{def}{=} (walkon_3, \omega).Token_3 + (serve_2, \mu).(walk_3, \omega).Token_3 \\
Token_3 &\stackrel{def}{=} (walkon_4, \omega).Token_4 + (serve_3, \mu).(walk_4, \omega).Token_4 \\
Token_4 &\stackrel{def}{=} (walkon_1, \omega).Token_1 + (serve_4, \mu).(walk_1, \omega).Token_1
\end{aligned}
$$

$$
\begin{aligned}
LAN &\stackrel{def}{=} (PC_{10} \parallel PC_{20} \parallel PC_{30} \parallel PC_{40}) \underset{L}{\bowtie} Token_1 \\
&\quad \text{where } L = \{walkon_1, walkon_2, walkon_3, walkon_4, serve_1, serve_2, serve_3, serve_4\}.
\end{aligned}
$$

Here we have arbitrarily chosen a starting state in which all the PCs are empty and the Token is at PC1.

10.5 Tool support for PEPA

A later note will give more details of the tool support for PEPA, which is the PEPA Plug-in for Eclipse. Here we just give a first example of using the tool to derive some information about the simple web service example presented in Section 2.2.

The form of PEPA models presented in notes and papers is based on the typesetting program LaTeX. For the tool there is a concrete syntax using standard ASCII characters. Thus the model shown on the left hand side of Figure 25 is represented in the `WS.pepa` file shown below.

```
p1 = 0.3;
p2 = 0.7;
lambda = 1.0;
m = 100;
rq = 500;
rp = 200;
mu = 20;
Appl = (think, p1*lambda).Appl1 + (think,p2*lambda).Appl2;
Appl1 = (local,m).Appl;
Appl2 = (request, rq).Appl3;
Appl3 = (respond, rp).Appl;
WS = (request,infty).WS1;
WS1 = (serve, mu). WS2;
WS2 = (respond, infty).WS;
Appl[1] <request,respond> WS[1]
```

In the tool there is an editing pane in the centre of the screen. This is where you can type in new PEPA files. When a file is saved it is parsed and subjected to static analysis. Any errors will be reported in the Problems view, below the Editor view.

It is important to note that there are two stages to Markovian analysis of a model, *mymodel*`.pepa`:

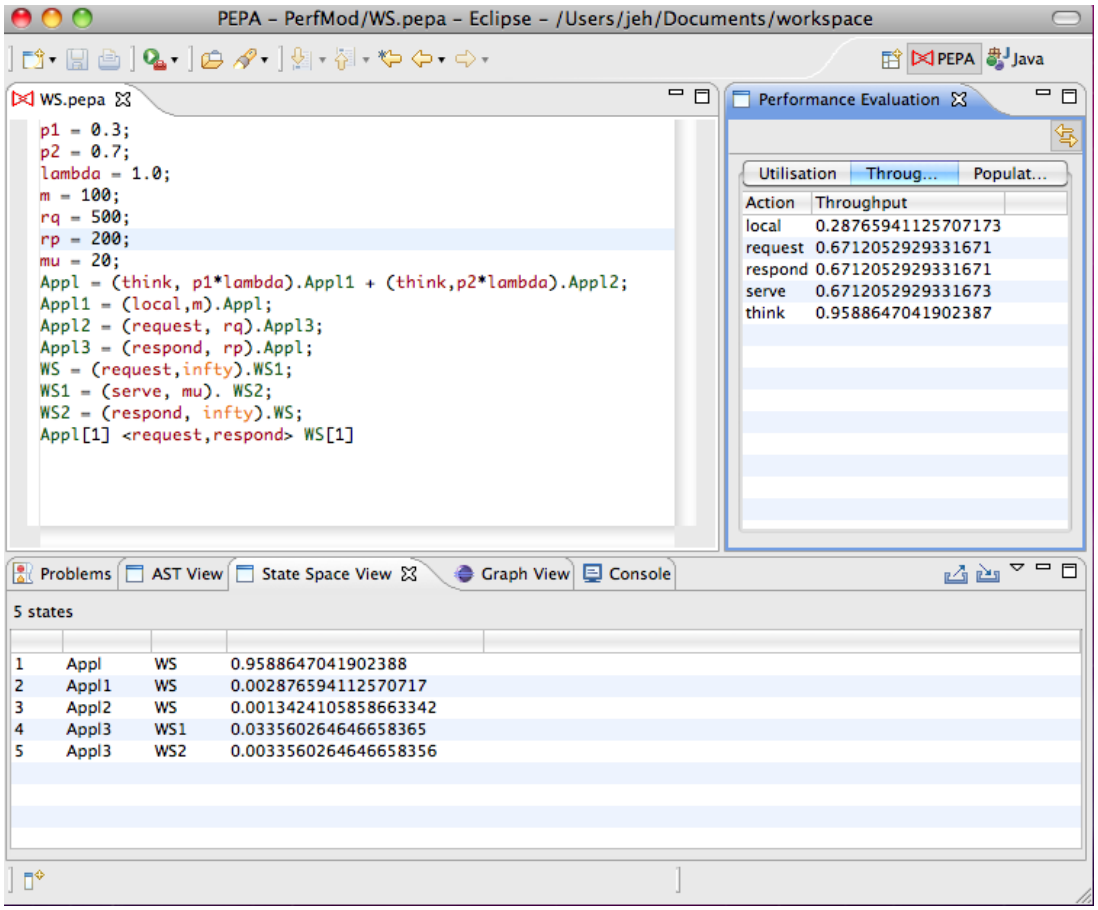

Figure 26: Screenshot of the PEPA Plugin for Eclipse

1. In the first stage the state space of the PEPA model is generated using the PEPA - CTMC - Derive command. A tabular representation of the derived state space is shown in the State Space View.

2. In the second stage this state space is interpreted as a Markov process and solved for steady state, using the appropriate solver. This achieved by selecting the command PEPA - CTMC - Steady State Analysis. Once the model is solved the State Space View will be updated with a column showing the steady state probability of each row.

Additionally some performance measures are derived automatically based on the structure of the model. Thus in the Performance Evaluation View at the right hand side of the screen you can find the throughput of each action, the relative probability of each state within each component, and the average number of instances of each state in each component.

A screenshot of the tool, with the model in the Editor pane and the results of steady state analysis, is shown in Figure 26.

Documentation for the tool can be found at `http://www.dcs.ed.ac.uk/pepa/documentation/` whilst the tool itself can be downloaded from `http://www.dcs.ed.ac.uk/pepa/tools/`

`plugin/download.html`. The second practical will involve using the PEPA Plugin for Eclipse so you are encouraged to install the tool and familiarise yourself with it.

10.6 Assumptions

The assumptions that we need to make about our models are those that we need to make for Markov modelling in general. Just as when we were considering GSPNs we were able to express some of these assumptions in terms of the Petri net, we can express some of the assumptions in terms of PEPA.

In order to ensure that the underlying Markov process is ergodic, i.e. does have a steady state probability distribution, the derivation graph of a PEPA model must be strongly connected. Necessary conditions for ergodicity, have been defined in terms of syntactic rules for PEPA models; e.g. if cooperation occurs it must be the highest level combinator—you cannot have a choice between two components which are themselves cooperation expressions.