8 Stochastic Process Algebra

In this lecture note we consider another class of performance modelling paradigms—stochastic extensions of *process algebras*. 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. The advantages of SPAs are that they incorporate the attractive features of process algebras and thus bring to the area of analytic performance modelling several attributes which are not offered by the existing formalisms. Perhaps the most important such feature is the compositionality which is inherent in the models and can be exploited during their analysis. In this note we present some simple examples which illustrate this and several other aspects of the SPA approach to performance modelling. Several stochastic process algebras have appeared in the literature but they are all broadly similar. Here we will concentrate on PEPA (Performance Evaluation Process Algebra).

8.1 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 CCS two agents communicate when one performs an action, a say, while the other performs the complementary action $\overline{a}$. The resulting communication action is regarded as an *internal* action that is invisible to the environment. Agents may proceed with their internal actions simultaneously but it is important to note that the semantics given to the language imposes an interleaving on such concurrent behaviour, i.e. it is not possible for two actions to occur simultaneously. The grammar of the language makes it possible to construct an agent which has a designated first action (prefix); has a choice over alternatives (choice); or has concurrent possibilities (composition).

The communication mechanism is different in CSP as there is no notion of complementary actions: this is a major distinction between CCS and CSP. In CSP two agents communicate by simultaneously executing actions with the same label. Since during the communication the joint action remains visible to the environment, it can be reused by other concurrent processes so that more than two processes can be involved in the communication. This is the communication mechanism adopted in the SPA languages.

8.2 PEPA

Process algebras offer several attractive features which are not necessarily available in existing performance modelling paradigms. The most important of these are *compositionality*, the ability to model a system as the interaction of its subsystems, *formality*, giving a precise meaning to all terms in the language, and *abstraction*, the ability to build up complex models from detailed components but disregarding internal behaviour when it is appropriate to do so. Queueing networks offer compositionality but not formality; SPN and GSPN offer formality but not compositionality; neither queueing networks nor Petri nets offer abstraction mechanisms.

PEPA extends classical process algebra by associating a random variable, representing duration, with every action. These random variables are assumed to be exponentially distributed and this leads to a clear relationship between the process algebra model and a Markov process.

PEPA models are described as interactions of *components*. Each component can perform a set of actions: an action a is described as a pair (α, r), where α is the *type* of the action and $r \in \mathbb{R}^+$ is the parameter of the exponential distribution governing its duration. Whenever a process P can perform an action, an instance of a given probability distribution is sampled: the resulting number specifies how long it will take to *complete* the action. A small set of combinators is used to build up complex behaviour from simpler behaviour. The combinators are familiar: sequential composition (*prefix*), *choice*, synchronisation (*cooperation*) and abstraction(*hiding*). We explain each of them below, in terms of a extremely simple model of a web-based information system.

Prefix (.): A component may have purely sequential behaviour, repeatedly undertaking one activity after another and eventually returning to the beginning of its behaviour. A simple example is a web server, which allows one data transfer at a time. Each browser requiring web pages etc. will need to acquire access to the server and only when the transfer is complete will the server be released and available again for acquisition.

$$Server \stackrel{def}{=} (get, \top).(download, \mu).(rel, \top).Server$$

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 server is passive with respect to the *get* action 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 a browser in the information system, displaying the current data may have two possible outcomes: demand for access to data available in the local cache (with probability p_1) or demand for access to data stored at the remote server (with probability $p_2 = 1 - p_1$). These alternatives are represented as shown below:

$$\begin{aligned} Browser \stackrel{def}{=}\ & (display, p_1\lambda).(cache, m).Browser\ + \\ & (display, p_2\lambda).(get, g).(download, \top).(rel, r).Browser \end{aligned}$$

A race condition is assumed to govern the behaviour of simultaneously enabled actions and 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 *display* action in the browser, 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 (decomposition principle). Note that in a GSPN we would represent this situation by a single timed transition to represent the *display* action, which when it fired enabled two immediate transitions with weights p_1 and p_2 to reflect the different possible outcomes.

Cooperation ($\underset{L}{\bowtie}$): We have already anticipated that the browser and the server in the example will be working together within the same system. This will require them to *cooperate* when the browser needs to download data which is not available locally. In contrast, the local activities of the browser can be carried out independently of the server. Cooperation over given actions is reflected in the cooperation by the *cooperation set*, $L = \{get, download, rel\}$ 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 contributory 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 system consists of just two browsers, the system is represented as the cooperation of the browsers and the server as follows:

$$WEB \stackrel{\mathit{def}}{=} (Browser \parallel Browser) \underset{L}{\bowtie} Server \qquad L = \{get, download, rel\}$$

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 browsers. This pure parallel combinator can be thought of as cooperation over the empty set: $(Browser \underset{\emptyset}{\bowtie} Browser)$.

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 information system we may wish to hide the access of a browser to its local cache. This might lead to a new representation of the browser:

$$Browser' \stackrel{\mathit{def}}{=} Browser/\{cache\}$$

and a corresponding new representation of the system:

$$WEB' \stackrel{\mathit{def}}{=} (Browser' \parallel Browser') \underset{L}{\bowtie} Server \qquad L = \{get, download, rel\}$$

Use of the hiding combinator in this way has two implications. Firstly, it ensures that no components added to the model at a later stage can interact, or interfere, with this action of the browser. 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 (names) such as $Server$ to associate identifiers with behaviours. Using recursive definitions we are able to describe components with infinite behaviours.

Representing the components of the system as separate components means that we can easily extend our model. Now we may want to consider a system consisting of more than two browsers which act independently of each other but compete for the use of the web server. This extension may be achieved compositionally by combining more instances of the $Browser$ component already described. For example, in the case of four browsers we have:

$$WEB4 \stackrel{def}{=} (Browser \parallel Browser \parallel Browser \parallel Browser) \underset{L}{\bowtie} Server$$

where, as before, $L = \{get, download, rel\}$.

8.3 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 a formal semantics, in the structured operational style, which associates a labelled transition system with every expression in the language. This form of directed graph shows the possible evolutions of the model.

Formally, a labelled transition system $(S, T, \{\xrightarrow{t} \mid t \in T\})$ consists of a set of states S, a set of transition labels T and a transition relation $\xrightarrow{t} \subseteq S \times S$. 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[16].

Based on the transition relation, a transition diagram, called the *derivation graph*, can be associated with each language expression. This graph describes all the possible evolutions of any component. It is completely analogous to the reachability graph of an SPN or GSPN. There the firing rule defined the semantics of the model; here the semantics is given by a set of transition rules, at least one for each combinator. An example derivation graph is shown in Figure 14 where the derivation graph of the PEPA model $WEB1$, consisting of a single browser accessing the web server, is shown. For clarity, in the upper part of the figure the derivatives of the components $Browser$ and $Server$ have been expanded. Inspection of the derivation graph allows us to derive qualitative properties of the model. In this case, for instance, we can see that the PEPA model is free from deadlock. Moreover, the Markov process underlying the PEPA model is obtained directly from the derivation graph: 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 derivation graph. Starting from the derivation graph of Figure 14, the derivation of the corresponding Markov process is straightforward

[16]Note that in contrast to CCS, the number of instances of a transition (action) is significant since it can affect the timing behaviour of a component.

$$
\begin{array}{rcllrcl}
Browser & \stackrel{def}{=} & \multicolumn{5}{l}{(display, p_1\lambda).Browser_1 + (display, p_2\lambda).Browser_2} \\
Browser_1 & \stackrel{def}{=} & (cache, m).Browser & & Browser_2 & \stackrel{def}{=} & (get, g).Browser_3 \\
Browser_3 & \stackrel{def}{=} & (download, \top).Browser_4 & & Browser_4 & \stackrel{def}{=} & (rel, r).Browser \\
Server & \stackrel{def}{=} & (get, \top).Server_1 & & Server_1 & \stackrel{def}{=} & (download, \mu).Server_2 \\
Server_2 & \stackrel{def}{=} & (rel, \top).Server & & & & \\
WEB1 & \stackrel{def}{=} & Browser \underset{L}{\bowtie} Server & & \multicolumn{3}{l}{L = \{get, download, rel\}}
\end{array}
$$

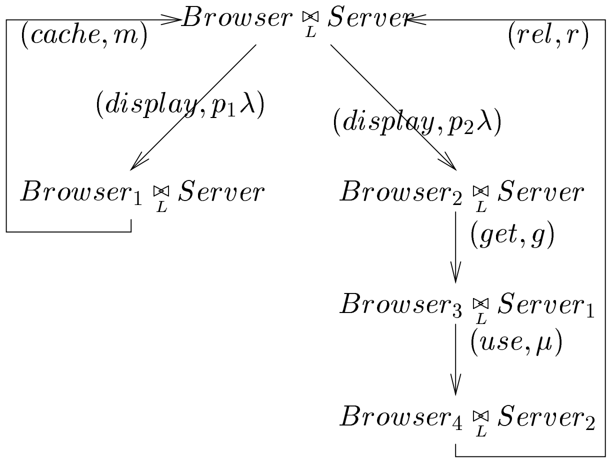

Figure 14: Derivation graph underlying Sys_4.

and results in the following generator matrix.

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

As usual, the probability distribution of the states of the model is generally not the ultimate goal of performance analysis. In order to define performance measures such as throughput and utilisation the modeller can define a reward structure over the Markov process at the level of the process algebra. Just as in a GSPN, using CSPL, we defined rewards in terms of the marking of places and the enabling of transitions, when using PEPA we associate rewards with actions being enabled[17]. The reward associated with a state is then the total reward attached to the actions it enables. Note that no reward can be attached to internal, τ, actions.

8.4 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 6). Each PC is represented by a separate

[17] There is no equivalent to *the number of tokens in a place* within a PEPA model.

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).

$$
\begin{aligned}
PC_{10} &\stackrel{def}{=} (arrive, \lambda).PC_{11} + (walkon_2, \omega).PC_{10} \\
PC_{11} &\stackrel{def}{=} (serve_1, \mu).PC_{10} \\[1em]
PC_{20} &\stackrel{def}{=} (arrive, \lambda).PC_{21} + (walkon_3, \omega).PC_{20} \\
PC_{21} &\stackrel{def}{=} (serve_2, \mu).PC_{20} \\[1em]
PC_{30} &\stackrel{def}{=} (arrive, \lambda).PC_{31} + (walkon_4, \omega).PC_{30} \\
PC_{31} &\stackrel{def}{=} (serve_3, \mu).PC_{30} \\[1em]
PC_{40} &\stackrel{def}{=} (arrive, \lambda).PC_{41} + (walkon_1, \omega).PC_{40} \\
PC_{41} &\stackrel{def}{=} (serve_4, \mu).PC_{40}
\end{aligned}
$$

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

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

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

8.5 The PEPA Workbench

The PEPA Workbench, is a suite of tools, intended to help a modeller to solve and experiment with PEPA models. It comes in various versions, but the one installed on the system under the alias `pepa` is a Java version which offers a number of different solution techniques. Information about the tool and a user guide (including a *quick start* section) can be found on the module web page.

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

1. In the first stage the state space of the PEPA model is generated using the `Run` command. This results in a file *mymodel*`.states`.

2. In the second state this state space is interpreted as a Markov process and solved for steady state, using the appropriate solver. This results in the file *mymodel*`.steadystate`.

Models to be loaded into the tool must be in a file with extension `.pepa` and in a version of the syntax which can be recreated in ASCII. The most obvious change is that instead of $P \underset{\{a,b\}}{\bowtie} Q$ we would type `P <a, b> Q`. Additionally if symbolic rates (e.g. `r, s` etc.) have been used in the model they must be explicitly assigned values at the start of the file. Similarly probabilities must be explicit values within the `.pepa` file.

```
lambda = 0.05;
m=0.1;
g =0.01;
r=0.001;
mu= 1.0;

Browser= (display,0.4*lambda).(cache,m).Browser +
        (display,0.6*lambda).(get,g).(download,infty).(rel,r).Browser;
Server = (get,infty).(download,mu).(rel,infty).Server;
Browser<get,download,rel>Server
```

Figure 15: `WEB1.pepa` file

An example PEPA file, corresponding to the small *WEB1* example discussed earlier, is shown in Figure 15. Examples of the `.states` and `.steadystate` files, corresponding to this model, are shown in Figures 16 and 17 below[18].

```
1 Browser <{rel, get, download}> Server
2 (cache,m=0.1).Browser <{rel, get, download}> Server
3 (get,g=0.01).(download,1.0*Top).(rel,r=0.0010).Browser
                                              <{rel, get, download}> Server
4 (download,1.0*Top).(rel,r=0.0010).Browser <{rel, get, download}>
                                     (download,mu=1.0).(rel,1.0*Top).Server
5 (rel,r=0.0010).Browser <{rel, get, download}> (rel,1.0*Top).Server
```

Figure 16: `WEB1.states` file

8.6 Efficient Solution

PEPA models are prone to the problem of state space explosion: even moderately sized system descriptions can generate a huge state space making analysis of the system infeasible. It is an area of on-going research to investigate when the component structure of the model can be used in order to gain efficient solution of the underlying Markov process, i.e. when can we consider the components of the model separately and yet still derive answers about the model as a whole.

[18]More information about the PEPA Workbench and associated tools is available via the WWW—`http://www.dcs.ed.ac.uk/pepa/`.

```
1  0.0292141396435875
2  0.005842827928717507
3  0.08764241893076259
4  8.764241893076253E-4
5  0.8764241893076249
```

Figure 17: `WEB1.steadystate` file

As we will see in a later lecture note efficient *product form solution* is one of the major attractions of queueing networks for performance modelling purposes. These models rely on a form of interaction between nodes in a network which allows them to be solved in isolation, since they behave as if independent, up to normalisation. In some circumstances PEPA models (and GSPN models) can also be solved in this way—each component is solved in isolation and the steady state probability vector of the complete state space is formed from the product of the individual steady state distributions of the components.

8.7 Assumptions

The assumptions that we need to make about our models are the same ones that we needed to make about Markov processes in general. But 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.