Expressing Performance Requirements using Regular Expressions to Specify Stochastic Probes over Process Algebra Models

Ashok Argent-Katwala Jeremy T. Bradley Nicholas J. Dingle

Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2AZ, United Kingdom

{a.argent-katwala,jb,njd200}@doc.ic.ac.uk

ABSTRACT

This paper describes how soft performance bounds can be expressed for software systems using stochastic probes over stochastic process algebra models. These stochastic probes are specified using a regular expression syntax that describes the behaviour that must be observed in a model before a performance measurement can be started or stopped. We demonstrate the use of stochastic probes on a $661,960$ state parallel, redundant web server model to verify its passage-time performance characteristics.

1. INTRODUCTION

In this paper, we unify the description of soft performance bounds in software systems which have been modelled using stochastic process algebras. Soft performance bounds are an integral part of software and system performance validation. For example, we might have a service-level agreement (SLA) that a particular type of SQL query must return a result within 0.35 seconds 98.7% of the time; this would be derived from a passage-time quantile. Alternatively, we might need to assure ourselves that the probability that a just-in-time compiler is running native code exactly 5 seconds after loading a Java applet is at least 0.8; this would be a transient constraint. Finally, we might have to demonstrate that the probability that a particular software failure mode is eventually reached is less than 0.002; this would be a steady-state measure.

In this paper, we unify the specification of all three soft performance criteria using a stochastic probe mechanism. A stochastic probe is a measurement device that defines arbitrary start and end points for such measures. We base the stochastic probe specification on a regular expression syntax (similar to action sequences [1]), which allows the user to describe the behaviour that should be seen in a software model before a performance measure is started or stopped. Unlike [1], however, these stochastic probes are themselves converted into stochastic process algebra components before interrogating the software model through process composition. In this way, we avoid trying to tailor notationally more complex logic specification paradigms, such as Linear-time Temporal Logic (LTL) or Dynamic LTL [2].

Why use stochastic process algebra models at all? Stochastic process algebras or SPAs provide us with a powerful formalism for modelling such software systems. SPAs provide the ability to describe and compose modular software while taking account of potentially many other processes that are concurrently competing for system resources. Additionally, the encapsulation of complex behaviour using random variables provides an abstraction that allows us to obtain soft bounds either through analysis or simulation.

The paper is structured as follows: we will briefly introduce the stochastic probe concept and show how it can be used to describe all the three soft performance criteria (Section 2). For this study, we use the stochastic process algebra PEPA (although we could use any SPA in practice) and so we briefly outline PEPA in Section 2.4. Section 2.5 demonstrates some stochastic probe examples on a simple PEPA model. We then define how the regular expression probe specification can be translated into PEPA in Section 3.1. In Section 4, we outline how the ipc/DNAmaca tool takes a PEPA model, with integrated stochastic probe, through to analysis. Finally, we take a model of a parallel, redundant web server in Section 5 and run several stochastic probe measurements over it to demonstrate its performance behaviour.

2. STOCHASTIC PROBES

2.1 Introduction

Using regular expressions over action names to probe system behaviour is not a new concept. In particular, work with Propositional Dynamic Logic (PDL) introduced by [3, 4], and related dynamic logics has extended from reasoning about the decidability and complexity of systems to dealing with action predicates and fluents—notably in [5] using a PDL-based logic and [6] using LTL. In [5], De Giacoma and Lenzerini use regular expressions as state formulae in the modalities of PDL to describe possible or necessary flows of actions.

DLTL [2] is an extension of LTL to incorporate the dynamic aspects of PDL, which uses regular expressions to express programs. The modalities have been used in [1] to express fluents of actions which can in turn be used to pose purely functional queries of programs.

As an example of regular expressions being used in a very different stochastic context, stochastic regular expressions were introduced in [7] where the expressions themselves encode the stochastic behaviour. These expressions are then probabilistically pattern matched against strings. Our approach differs somewhat, in that our stochastic probes are non-interfering—all of the stochastic behaviour is already in the system and we merely want to define and observe certain measures.

2.2 Stochastic Probe Definition

We represent stochastic probe definitions as regular expressions (as introduced by Kleene in [8]). In this case, however, the atoms of the regular expression are action names drawn from the alphabet of the underlying process algebra model. This is a significant generalisation over [9], where the modeller is only allowed a fixed-structure two state probe. As such, a probe definition, R, has the following elements:

Operator	Description
a	action label
R, R	sequence operator
$R \mid R$	choice operator
$R?$	zero-or-one operator
$R\{n\}$	iterative operator
$R\{m, n\}$	range operator
R^+	positive closure operator
R^*	closure operator

a is an action label that matches a label in the system. Any action specified in the probe has to be observed in the model before the probe can advance a state. An action, a, can also be distinguished as a *start* or *stop* action in a probe and signifies an action which will start or stop a measurement, respectively.

R_1, R_2 is the sequential operator. R_1 is matched against the system's operation before R_2 is matched.

$R_1 \mid R_2$ is the choice operator. Either R_1 or R_2 is matched against the system being probed.

$R\{n\}$ is the iterative operator. n sequential copies of R are matched against the system e.g. $R\{3\}$ is simply shorthand for R, R, R.

$R\{m, n\}$ is the range operator. Between m and n copies of R are matched against the system's operation.

R^* is the closure operator, where zero or more copies of R are matched against the system.

R^+ is the positive closure operator, where one or more copies of R are matched against the system. It is syntactic sugar for R, R^*.

$R?$ is the optional operator, matching zero or one copy of R against the system. It is syntactic sugar for $R\{0, 1\}$.

2.3 Performance Analysis with Stochastic Probes

A stochastic probe is a fragment of stochastic process algebra, for our purposes PEPA, which describes the start and end points of a measurement that a modeller wishes to make of a system. Typically a modeller will wish to make one of the following types of measurement of a given system and will need to specify the start and end points for such a measurement:

steady-state measures are used to calculate the probability that the system ends up in the given set of states S eventually (independent of start state). In this case, a modeller would use a single probe to partition the state space into S and $\neg S$. The start actions, from the probe regular expression, define the entering transitions to S and the stop transitions, the leaving transition from S.

transient measures give the probability that a system is in a given set of states S_2 at time t, having started from one of the start states in S_1. In this case, we use the probe in slightly a slightly different way from when we were specifying a steady-state quantity. Here S_1 is taken to be the set of states that the system (and probe) are in after a start transition, and S_2 is similarly defined to be the set of states that occur after the stop transitions.

passage-time measures depict the probability that the system completes a passage from a given set of states S_1 to states in a given set S_2 in time t. As with transient measures, we take the intuitive approach of letting S_1 be defined by the start actions and S_2 by the stop actions of the probe. Passage-time analysis can produce either the probability density or cumulative distribution function of the amount of time taken. These are used most frequently for soft bounds on event-duration.

2.4 PEPA

We use the stochastic process algebra PEPA to represent our stochastic probes and models. PEPA [10] is a parsimonious stochastic process algebra that can describe compositional stochastic models. These models consist of components whose actions incorporate random exponential delays.

In fact, stochastic probes are SPA-independent and we could have tailored them to any suitable SPA. PEPA suited our needs here as it has an uncomplicated syntax which lends itself to describing the underlying ideas. As we will see, the probe is expressed as a single PEPA component, so that it can then be combined with the model being queried.

The syntax of a PEPA component, P, is represented by:

$$\mathrm{P} \quad ::= \quad (\mathrm{a}, \lambda).\mathrm{P} \;\big|\; \mathrm{P} + \mathrm{P} \;\big|\; \mathrm{P} \underset{S}{\bowtie} \mathrm{P} \;\big|\; \mathrm{P}/\mathrm{L} \;\big|\; \mathrm{A} \qquad (1)$$

$(\mathrm{a}, \lambda).\mathrm{P}$ is a prefix operation. It represents a process which does an action, a, and then becomes a new process, P. The time taken to perform a is described by an exponentially distributed random variable with parameter

λ. The rate parameter may also take the value $\top$, which makes the action passive in a cooperation (see below).

$P_1 + P_2$ is a choice operation. A race is entered into between components P_1 and P_2. If P_1 evolves first then any behaviour of P_2 is discarded and vice-versa.

$P_1 \underset{S}{\bowtie} P_2$ is the cooperation operator. P_1 and P_2 run in parallel and synchronise over the set of actions in the set S. If P_1 is to evolve with an action $a \in S$, then it must first wait for P_2 to reach a point where it is also capable of producing an a-action, and vice-versa. In an active cooperation, the two components then jointly produce an a-action with a rate that reflects the slower of the two components (usually the minimum of the two individual a-rates). In a passive cooperation, where P_1, say, can evolve with an $(a, \top)$-transition, the joint a-action inherits its rate from the P_2 component alone.

P/L is a hiding operator where actions in the set L that emanate from the component P are rewritten as silent τ-actions (with the same appropriate delays). The actions in L can no longer be used in cooperation with other components.

A is a constant label and allows, amongst other things, recursive definitions to be constructed.

While on the subject of PEPA, it is worth noting that Gilmore and Hillston [11] have developed their own *feature interaction* logic which explores a PEPA model, assigning reward to component states for use in steady-state and transient-state analysis. This is an alternative technique to the one we are trying to achieve here. Instead of using a logic to interrogate a model, we use the language's own cooperation operator to observe the key events that we wish to measure. By selectively sampling a model's behaviour in this way, we can simplify the task of picking the states that are relevant to our measure.

Our method has the benefit of not requiring the user to learn an entirely different paradigm (being based on the process algebra in which the model is described). At the moment, it has the downside that, being observationally-based, in cannot distinguish actions that are generated by different copies of the same component. This is possible in a logic setting such as that set up by Gilmore et al [12] and Clark [13] for steady-state measure specification.

2.5 Stochastic Probe Examples

We give a few examples of stochastic probes as specified by regular expressions over a simple PEPA model.

$$\begin{aligned}
\mathrm{Light}_1 &\overset{\mathit{def}}{=} (\mathrm{red}, r_1).(\mathrm{green}, r_2).(\mathrm{blue}, r_3).\mathrm{Light} \\
&\quad + (\mathrm{fail}, r_4).\mathrm{Light}_2 \\
\mathrm{Light}_2 &\overset{\mathit{def}}{=} (\mathrm{repair}, r_1).\mathrm{Light}_1
\end{aligned}$$

This describes a simple oscillating light that can periodically fail and then repair itself. We look at the following probe definitions:

Example 1: (red:start, fail:stop) creates a probe which is designed to wait for a red action, then absorbs any other behaviour before changing state again on seeing a fail. It could be used to measure the time to failure (passage-time). It could be used to identify all the states that make up the *correct* functioning of the light for a steady-state measure. It could be used to ascertain the probability of failing at a particular time, t, if used in a transient measure.

Example 2: ((fail, red:start, red:stop) | (red:start, red:stop)) creates a probe which could be used to analyse the time between successive red-actions after a fail or in normal operation.

Example 3: (green:start, blue$\{3\}$, blue:stop) generates a probe which looks for the occurrence of a green action followed by four blue actions.

It is important to realise that the probe will never block the behaviour of the model it is synchronising with. As described in the next section, the probe will absorb behaviour (without altering state) which it sees that is not part of its next specified action. In LTL terms, a modeller can only ask a question, "will I ever see this behaviour?" and not "will I see exactly this behaviour next?". Asking the latter question with a probe could change the functional behaviour of the model, and as we have said, probes are meant to be neutral in the effect they have on the model.

3. STOCHASTIC PROBE GENERATION

3.1 Syntax

Formally, a stochastic probe has the following syntax:

$$\begin{aligned}
R &::= a \mid T, T \mid (S) \\
S &::= T \mid T|S \\
T &::= R \mid R\{n\} \mid R\{m,n\} \mid R^+ \mid R^* \mid R? \\
a &::= act \mid act : start \mid act : stop
\end{aligned} \tag{2}$$

In diagrammatic terms, Fig. 1 shows the conversion of the individual regular expression elements to process algebra components.

Symbolically, we first build a parse tree, during which the syntactic sugar is removed, converting R^+ to R, R^* and $R\{n\}$ to $R, \ldots, R$ to give n explicit copies of R. At each stage of the conversion, only the topmost element in the tree is considered, and the subtrees are handled recursively.

3.2 Probe Mapping to PEPA

In the definitions below, we take as input the PEPA component name, P, we are to define and the component name we are to end at, Q. We introduce intermediate component labels N_i, as required.

Action a: is always a leaf node and translates to:

$$\mathrm{P} \overset{\mathit{def}}{=} (\mathrm{a}, \top).\mathrm{Q}$$

Choice $R_1 \mid R_2$: translates to $\mathrm{P} \overset{\mathit{def}}{=} \mathrm{N}_1 + \mathrm{N}_2$ and the algorithm is repeated for the sub-trees with R_1 running from N_1 to Q and R_2 running from N_2 to Q.

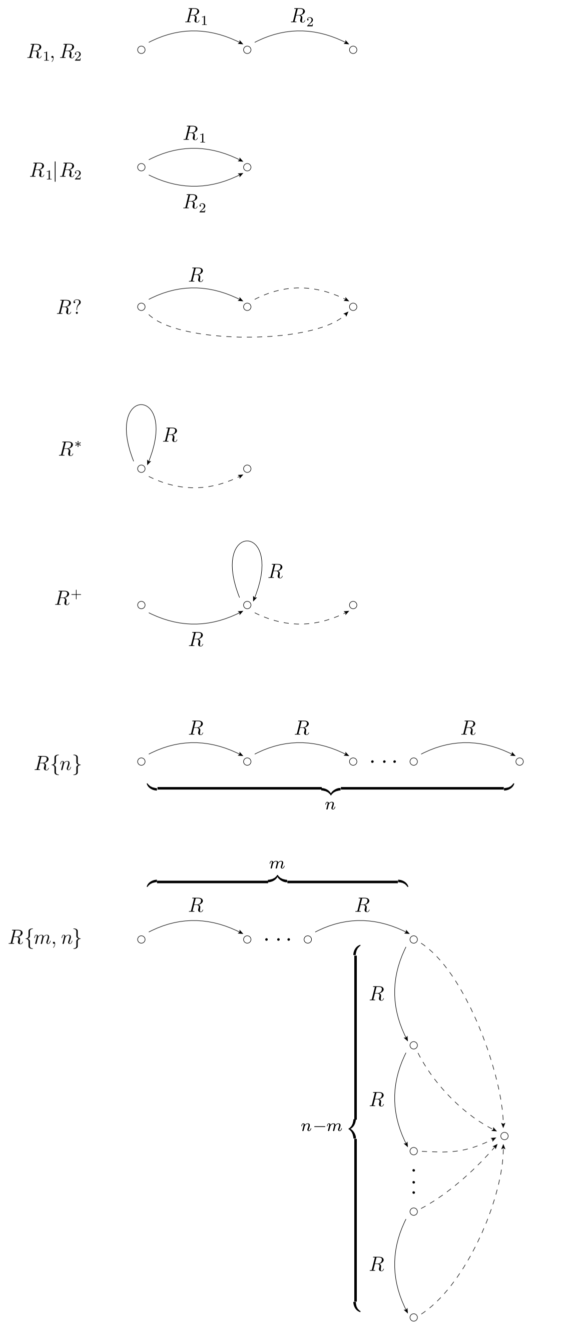

Figure 1: The representation of the distinct regular expression terms as PEPA components

Sequence R_1, R_2**:** translates to R_1 running from P to N_1 and R_2 from N_1 to Q.

Closure R^***:** becomes $\mathrm{P} \stackrel{def}{=} \mathrm{N}_1 + \mathrm{Q}$, where R is translated running from N_1 to P.

Zero-or-one $R?$**:** becomes $\mathrm{P} \stackrel{def}{=} \mathrm{N}_1 + \mathrm{Q}$, where R is translated running from N_1 to Q.

Range $R\{n,m\}$**:** can be rewritten as $(R\{n\}, R\{0, m-n\}) \equiv (R\{n\}, (R, R\{0, m-n-1\})?)$, where $R\{0,1\} \equiv R?$

$$
\begin{aligned}
\mathrm{Probe} &\stackrel{def}{=} (\mathrm{foo}, \top).\mathrm{Probe1} + (\mathrm{bar}, \top).\mathrm{Probe} \\
\mathrm{Probe1} &\stackrel{def}{=} \mathrm{Probe2} + \mathrm{Probe3} \\
\mathrm{Probe2} &\stackrel{def}{=} (\mathrm{foo}{:}\mathsf{start}, \top).\mathrm{Probe4} \\
\mathrm{Probe3} &\stackrel{def}{=} (\mathrm{bar}, \top).\mathrm{Probe4} \\
\mathrm{Probe4} &\stackrel{def}{=} \mathrm{Probe5} + \mathrm{Probe6} \\
\mathrm{Probe5} &\stackrel{def}{=} (\mathrm{foo}{:}\mathsf{stop}, \top).\mathrm{Probe} \\
\mathrm{Probe6} &\stackrel{def}{=} (\mathrm{bar}, \top).\mathrm{Probe}
\end{aligned}
$$

Figure 2: The PEPA description of the probe specification $(\mathrm{foo}, (\mathrm{foo}{:}\mathsf{start} \mid \mathrm{bar}), (\mathrm{foo}{:}\mathsf{stop} \mid \mathrm{bar}))$**.**

After the complete regular expression has been translated, the last labels in the probe description are set to point at the beginning of the probe. Thus, the probe is made cyclic and, for a carefully designed probe, irreducibility is preserved in the whole system.

To make the probe non-interfering, when composed with the system under analysis, we ensure all actions in the cooperation set are enabled in every state of the probe. We do this, at each state, by adding null loops which absorb any action not immediately available from that state of the probe. In constructing these loops, we do not discriminate between the untagged and tagged forms of action labels—i.e. a, a:start and a:stop are all considered equal and the actions on the loops only ever use the plain, untagged names. Thus, the probe will never change the observable behaviour of any system it is composed with.

This procedure gives us a valid PEPA fragment which will behave properly as a probe. However, for our analysis with ipc/DNAmaca we also need to be able to tell, purely from the state of the probe, whether it is running or stopped. To achieve this, we need to be able to deal with the case where there are distinct paths between two states containing a mixture of plain and tagged actions (as in Fig. 3, between states Probe1 and Probe4 where, as we will see, it is important to know which of the foo:start or bar actions was performed).

So, at a cost of at most doubling the state space of the probe, we make an observationally equivalent version which has a partition in its state space, Running and Stopped. This allows DNAmaca to make a state-based decision as to whether the probe is active.

3.3 Probe Example

For example, for the expression:

$$(\mathrm{foo}, (\mathrm{foo}{:}\mathsf{start} \mid \mathrm{bar}), (\mathrm{foo}{:}\mathsf{stop} \mid \mathrm{bar}))$$

the approach would generate the process depicted in Fig. 3 with PEPA description of Fig. 2. Plainly, the probe can bypass the tagged foo actions whenever they are presented so we cannot know whether the probe is running or not based on the state alone.

To solve this, we introduce a further two-state tracker process (used originally in [9]) that observes whether the probe measurement has started or not; it silently consumes start actions when running and stop actions when stopped.

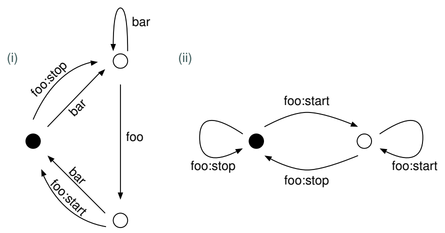

Figure 3: (i) Probe and (ii) two-state running/stopped tracker

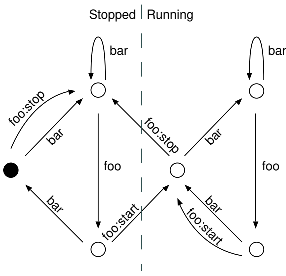

Figure 4: Full probe, made by cooperating the 3-state probe and the tracker over the actions foo:start **and** foo:stop

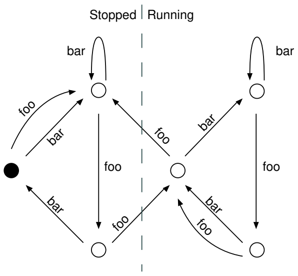

Figure 5: Complete, relabelled probe

Now the full probe, shown in Fig. 4, is made from:

$$\begin{aligned}
\text{Stopped} &\stackrel{def}{=} (\text{foo:}\mathsf{start}, \top).\text{Running} \\
&\quad + (\text{foo:}\mathsf{stop}, \top).\text{Stopped} \\
\text{Running} &\stackrel{def}{=} (\text{foo:}\mathsf{start}, \top).\text{Running} \\
&\quad + (\text{foo:}\mathsf{stop}, \top).\text{Stopped} \\
\text{FullProbe} &\stackrel{def}{=} \text{Stopped} \underset{L_p}{\bowtie} \text{Probe}
\end{aligned}$$

where $L_p = \{\text{foo:}\mathsf{start}, \text{foo:}\mathsf{stop}\}$.

Note that at this level we are treating foo:start and foo:stop as ordinary action names that happen to end in ":start" and ":stop", so the plain foo above is not involved in the synchronisation.

If we used relabelling, as in CCS [14], to convert foo:start and foo:stop into plain foo actions so that they interact properly with the model, we would be done. We could use the definition of FullProbe as stated and just observe the tracker component switch from Stopped to Running with DNAmaca to determine when to start and end the measurement.

PEPA, however, does not include relabelling, but we can still build a probe process that would have the same state space and behaviour. By explicitly generating the state space of FullProbe where all the foo:start and foo:stop actions have been relabelled to plain foo actions, as shown in Fig. 5, we can achieve the same effect. Now, instead of letting DNAmaca observe the tracker process switch states, it observes the overall probe transit from one state partition to the other.

4. THE IPC/DNAMACA TOOL CHAIN

4.1 ipc: The Imperial PEPA Compiler

ipc performs a translation from PEPA to a stochastic Petri net formalism (similar to that described in [15]), and also incorporates any extra logic necessary for expressing the passage-time, transient or steady-state query derived from Section 3.1. The ipc compiler consists of:

1. .pepa file parser
2. PEPA normal form translator
3. component state space explorer
4. DNAmaca component linker and .mod file generator
5. PEPA-measurement specifier
6. command line parser for passage-time and steady-state queries

The .pepa file format allows, for instance, arbitrary numbers of sequential prefixes and also arbitrary numbers of summation and cooperation operations attributed to a single constant label. The normal form in question strictly enforces the binary summation and cooperation, insisting on constant labels after each operation; it therefore also has to take care of the unique labelling of all components' states.

The component linker takes the DNAmaca description of the individual PEPA components and creates shared transitions with appropriate preconditions and actions to represent the cooperation over shared actions.

The PEPA-measurement specifier augments the input PEPA model with the process algebra probe (see Section 3.1) and adds the command for the requisite measurement specification (passage-time, transient or steady-state) to the DNAmaca file.

4.2 DNAmaca: Markovian Analyser

DNAmaca [16] is a modelling language for Markov and semi-Markov chains. Previous publications [17, 18, 19, 20, 21, 22] describe the mathematical foundation for the calculation of steady-state, transient and passage-time distributions in such models, so we do not dwell on the details here. The DNAmaca interface language, to which ipc compiles, is described in [16], with the passage-time specification syntax in [19]. ipc uses this to describe the performance measurement to be carried out on the stochastic probe, and thus on the model itself.

4.3 Numerical Calculations

A PEPA model reduces to an underlying continuous-time Markov chain (CTMC), so we consider an n state CTMC with $n \times n$ generator matrix $Q = q_{ij}$. Solving the linear system $\pi Q = 0$ subject to $\sum \pi_i = 1$ gives us the steady-state vector, π. For the steady-state measure, the stochastic probe specifies a set of states, $\vec{k}$, (between start and stop actions) that we wish to know the steady-state probability for, where $\pi_{\vec{k}} = \sum_{i \in \vec{k}} \pi_i$.

We calculate passage-time densities $P_{\vec{i}\vec{j}}$ and transient measures $T_{\vec{i}\vec{j}}(t)$ from many source states $\vec{i}$ to many target states $\vec{j}$. In these cases, the sets of states, $\vec{i}$ and $\vec{j}$, are specified by the stochastic probe being started and being stopped respectively. The calculation is then performed by means of efficient uniformisation-based analysis [23, 24]. In the passage-time calculation, uniformisation is applied to a modified Markov chain, where all the target states, $\vec{j}$, have been made absorbing. This technique is described in detail in [19, 25].

5. EXAMPLE: HIGH-AVAILABILITY WEB SERVER

In this section, we demonstrate performance and reliability analysis of a high-availability distributed web server. Using stochastic probes to pick out key system passage-times, we make soft performance and reliability statements about the number of requests that the system can handle within a given time, as well as the probability of complete failure within the first minutes or hours of operation.

A typical application scenario for such a system is a web-based current events news feed which must meet strict QoS or quality-of-service requirements on system availability and response-time. Regardless of whether the underlying technology is web-based or not, such systems require careful performance engineering to achieve peak efficiency [26]. In part, the strict QoS requirements are met by skewing the prioritisation for *fast* reads over writes so that writes are buffered and only processed at times of low read load. The consequence of this is that there is no guarantee that a reader will see the latest version of the site although high availability is maintained.

The system is made up of a cluster of servers, each of which can fail and be repaired independently. If all of the servers fail then a special recovery mechanism can restart them all. We now proceed to describe the components of the system.

5.1 The Server Model

The server receives read requests, each of which incurs a read lookup before the server is available to serve the next request (states Server and Server_read below). A successful write request requires that none of the servers are in the process of performing a read access, so that all servers can be simultaneously updated (therefore no s_write action is allowed in the Server_read state). The server may fail and while failed (the Server_fail state) read requests are not intercepted (no s_read_request activities) whereas write requests (s_write) are absorbed without action. It is assumed that server resynchronisation occurs during recovery. The write costs are different in the functioning and failure states (the costs are quantified by variables r_{s_w} and $r_{s_{fsw}}$ respectively). Failed servers may be repaired individually (s_fail_recover) or collectively (s_fail_recover_all).

$$
\begin{aligned}
\mathrm{Server} &\stackrel{def}{=} (\mathrm{s\_read\_request}, \top).\mathrm{Server\_read} \\
&+ (\mathrm{s\_fail}, r_{s_f}).\mathrm{Server\_fail} \\
&+ (\mathrm{s\_write}, r_{s_w}).\mathrm{Server} \\
\mathrm{Server\_read} &\stackrel{def}{=} (\mathrm{s\_read\_lookup}, r_{s_{rl}}).\mathrm{Server} \\
\mathrm{Server\_fail} &\stackrel{def}{=} (\mathrm{s\_fail\_recover}, r_{s_{fr}}).\mathrm{Server} \\
&+ (\mathrm{s\_fail\_recover\_all}, \top).\mathrm{Server} \\
&+ (\mathrm{s\_write}, r_{s_{fsw}}).\mathrm{Server\_fail}
\end{aligned}
$$

5.2 Server Groups

The recovery of the servers is co-ordinated by a server group manager. The responsibility of this component is to witness server failures and recoveries from failures. When the server number, S, is reached and all servers have failed, the server group is restarted with all failures recovered simultaneously (as a result of a high-priority repair).

$$
\begin{aligned}
\mathrm{Server\_group_0} &\stackrel{def}{=} (\mathrm{s\_fail}, \top).\mathrm{Server\_group_1} \\
\mathrm{Server\_group_i} &\stackrel{def}{=} (\mathrm{s\_fail}, \top).\mathrm{Server\_group_{i+1}} \\
&+ (\mathrm{s\_fail\_recover}, \top) \\
&\quad .\mathrm{Server\_group_{i-1}} : 1 \leq \mathrm{i} < \mathrm{S} \\
\mathrm{Server\_group_S} &\stackrel{def}{=} (\mathrm{s\_fail\_recover\_all}, r_{s_{gsfra}}) \\
&\quad .\mathrm{Server\_group_0}
\end{aligned}
$$

Taking, for example, $S = 4$, the process instantiation expression for the server cluster is as shown below:

$$
\begin{aligned}
\mathrm{Servers} \stackrel{def}{=} &(\mathrm{Server} \underset{\mathcal{L}}{\bowtie} \mathrm{Server} \underset{\mathcal{L}}{\bowtie} \mathrm{Server} \underset{\mathcal{L}}{\bowtie} \mathrm{Server}) \\
&\underset{\mathcal{L}'}{\bowtie} \mathrm{Server\_group_0}
\end{aligned}
$$

where:

$$
\begin{aligned}
\mathcal{L} &= \{\mathrm{s\_write}, \mathrm{s\_fail\_recover\_all}\} \\
\mathcal{L}' &= \mathcal{L} \cup \{\mathrm{s\_fail\_recover}\}
\end{aligned}
$$

5.3 Buffered Writes

The write buffer manages the promotion of buffered writes to server write actions. To ameliorate the relative infrequency of all the servers being available simultaneously to perform an s_write, write requests are necessarily buffered until the buffer capacity, B, is reached. When an s_write action does occur, the entire write buffer is executed and emptied.

$$
\begin{aligned}
\mathrm{Write\_buffer_0} &\stackrel{def}{=} (\mathrm{b\_write}, \top).\mathrm{Write\_buffer_1} \\
\mathrm{Write\_buffer_i} &\stackrel{def}{=} (\mathrm{b\_write}, \top).\mathrm{Write\_buffer_{i+1}} \\
&+ (\mathrm{s\_write}, \top).\mathrm{Write\_buffer_0} \\
&\qquad : 1 \leq i < B \\
\mathrm{Write\_buffer_B} &\stackrel{def}{=} (\mathrm{s\_write}, \top).\mathrm{Write\_buffer_0}
\end{aligned}
$$

5.4 Web Authors and Browsers

Web authors (writers) issue web content to the system. Web browsers are the readers in our system. It is assumed that there are multiple writers and, comparatively, a much larger number of web browsers (readers). This accounts for the prioritisation of read access over writes.

Writers may perform only buffered writes. They have no capacity to perform a server write directly.

$$\begin{aligned} \text{Writer} &\stackrel{def}{=} (\text{b\_write}, r_{w_{bw}}).\text{Writer\_writ} \\ \text{Writer\_writ} &\stackrel{def}{=} (\text{w\_reset}, r_{w_r}).\text{Writer} \end{aligned}$$

With three writers ($W = 3$) the process instantiation expression for the writers is as shown below:

$$\text{Writers} \stackrel{def}{=} \text{Writer} \underset{\emptyset}{\bowtie} \text{Writer} \underset{\emptyset}{\bowtie} \text{Writer}$$

Readers send read requests and then await the response from the server.

$$\begin{aligned} \text{Reader} &\stackrel{def}{=} (\text{s\_read\_request}, r_{r_{srr}}) \\ &\quad .(\text{s\_read\_lookup}, \top).\text{Reader\_read} \\ \text{Reader\_read} &\stackrel{def}{=} (\text{r\_reset}, r_{r_r}).\text{Reader} \end{aligned}$$

If we model three readers ($R = 3$), for instance, we get:

$$\text{Readers} \stackrel{def}{=} \text{Reader} \underset{\emptyset}{\bowtie} \text{Reader} \underset{\emptyset}{\bowtie} \text{Reader}$$

5.5 The System Equation

The system is built by composing the behaviours of the simpler component to form the behaviour of the model as a whole:

$$\begin{aligned} \text{Environment} &\stackrel{def}{=} \text{Writers} \underset{\emptyset}{\bowtie} \text{Readers} \\ \text{Web\_cluster} &\stackrel{def}{=} \text{Servers} \underset{\{\text{s\_write}\}}{\bowtie} \text{Write\_buffer}_0 \\ \text{System} &\stackrel{def}{=} \text{Environment} \underset{\mathcal{N}}{\bowtie} \text{Web\_cluster} \end{aligned}$$

where $\mathcal{N} = \{\text{b\_write}, \text{s\_read\_request}, \text{s\_read\_lookup}\}$.

6. RESULTS

In this section, we describe three distinct stochastic probes which we use to extract passage-time distributions from the underlying Markov model via the ipc/DNAmaca tool. We focus on passage-time analysis as this is still a relatively rare piece of analysis within stochastic process algebra modelling.

Probe	S	B	W	R	States	Transitions
A	5	4	3	3	362,240	4,061,440
B	5	4	3	3	581,200	6,516,356
C	5	4	3	5	661,960	9,440,988

Table 1: Size of analysed model for given probes and model configurations

Tab. 1 shows the size of the Markovian model to be analysed once the stochastic probe query has been incorporated. The probes used are:

Probe A: measures the length of time to process at least 4 read events fully, i.e. observe 4 s_read_lookup actions.

$$\begin{aligned} \Pi_A &= \text{s\_read\_request:}\mathsf{start}, (\text{s\_read\_lookup}\{4\}), \\ &\quad \text{r\_reset:}\mathsf{stop} \end{aligned}$$

The measurement is started on seeing the first start action, s_read_request, and terminates, after the passage of at least 4 s_read_lookup actions, upon completion of the next r_reset. The probability density function of this passage is show in Fig. 6. From this we can see that there is a high probability that all 4 read actions will have occurred by time $t = 3$.

Probe B: observes the length of time it takes either to observe 4 completed reads or 2 completed writes.

$$\begin{aligned} \Pi_B &= (\text{s\_read\_request:}\mathsf{start}, (\text{s\_read\_lookup}\{4\}), \\ &\quad \text{r\_reset:}\mathsf{stop}) \mid (\text{b\_write:}\mathsf{start}, \text{b\_write}\{2\}, \\ &\quad \text{w\_reset:}\mathsf{stop}) \end{aligned}$$

The probe is started by either seeing a s_read_request or a b_write. It then either looks for 4 s_read_lookup actions or 2 further b_write actions, before terminating on an appropriate read or write reset. Fig. 7 and Fig. 8 show the probability density function and cumulative distribution function for the passage, respectively. From the cumulative function, we can deduce the soft bound, that the system will complete either 4 reads or 2 writes within 4 time units with probability 0.98856.

Probe C: extracts the passage-time to first full system failure.

$$\begin{aligned} \Pi_C &= (\text{s\_read\_request:}\mathsf{start} \mid \text{b\_write:}\mathsf{start}), \\ &\quad \text{s\_fail\_recover:}\mathsf{stop} \end{aligned}$$

A full system failure is said to have occurred if all the servers have independently failed and none has managed to recover. The probe starts the measurement on encountering a read request from a reader or a buffer write event from a writer. The measurement is only stopped when a full fail event (s_fail_recover) has occurred. This is a good example of a rare event which would be hard to simulate as the absolute number of complete failures is going to be very small for a given execution trace. Fig. 9 and Fig. 10 show the density function and cumulative distribution for this passage, respectively, and demonstrate how infrequent this occurrence is. From the cumulative distribution we can ascertain the soft reliability bound that the system will fail within the first 90 time units is 0.00559.

7. CONCLUSION

In this paper, we have introduced stochastic probes as a means of measuring soft performance characteristics over software systems. We presented a regular expression language which specifies the stochastic probe and is then itself converted into a stochastic process algebra component. We showed how this could be combined with the original SPA model and analysed to provide soft performance and reliability bounds. Probes in effect partition the model state space into states where the probe measure has been started, and states where it is stopped. This convenient partition allows three types of performance analysis, transient, passage-time and steady-state, to be expressed in a unified manner.

We finished by analysing a 661,960 state PEPA model of a parallel web server for a number of soft performance and reliability bounds.

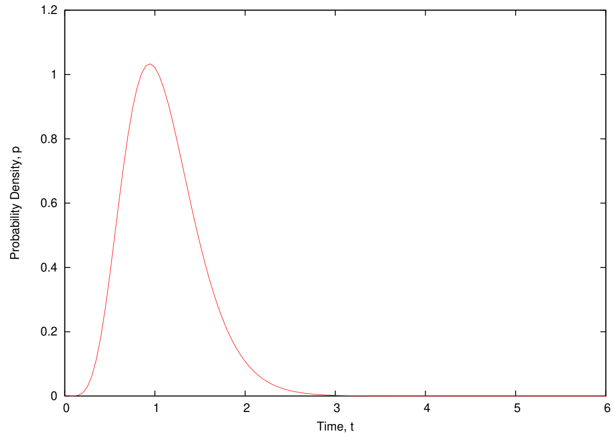

Figure 6: Probability density function for Probe A: a $362,240$ state system

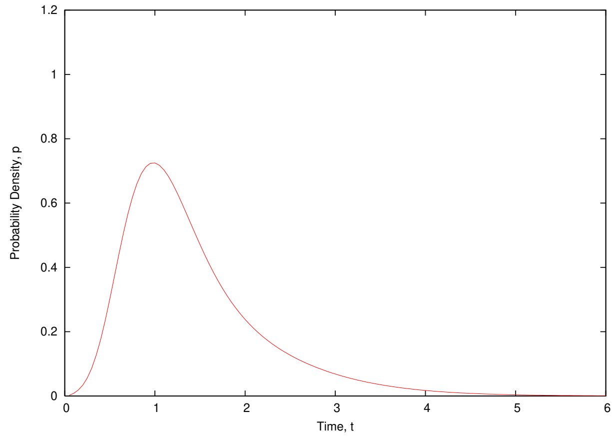

Figure 7: Probability density function for Probe B: a $581,200$ state system

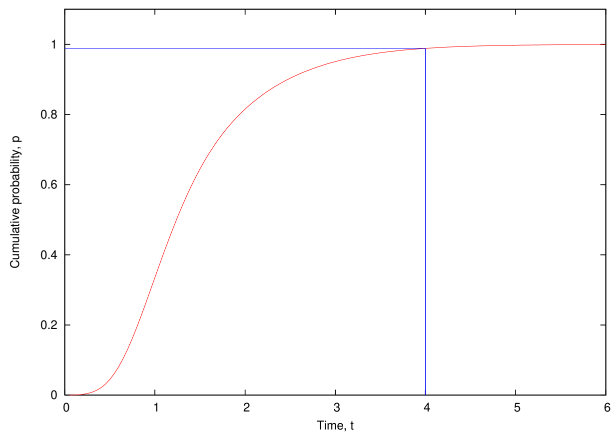

Figure 8: Cumulative distribution function for Probe B: a $581,200$ state system. Example quantile: probability that the specified read/write sequence takes less than time 4 is 0.989.

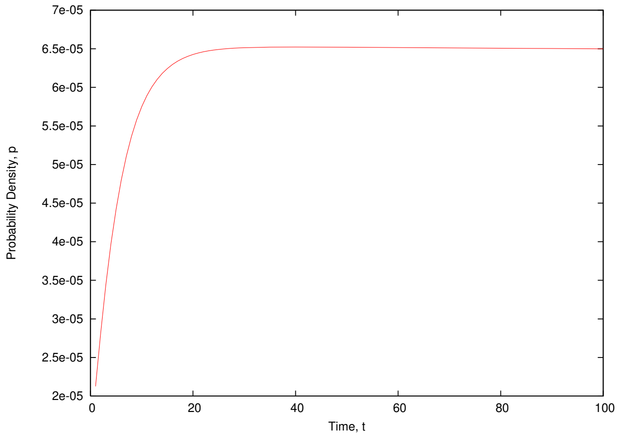

Figure 9: Probability density function for Probe C: a $661,960$ state system

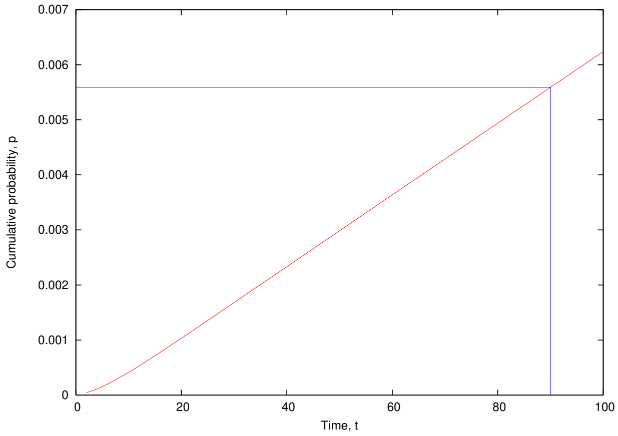

Figure 10: Cumulative distribution function for Probe C: a $661,960$ state system. Example quantile: probability that the specified failure mode occurs in the first 90 time units is 0.00559.

Acknowledgements

The authors would like to thank Stephen Gilmore, Tony Field and Helen Wilson for helpful remarks on the ideas behind this paper.

8. REFERENCES

[1] L. Giordano, A. Martelli, and C. Schwind, "Reasoning about actions in dynamic linear time temporal logic," *Logic Journal of the IGPL*, vol. 9, no. 2, pp. 273–287, 2001.

[2] J. G. Henriksen and P. S. Thiagarajan, "Dynamic linear time temporal logic," *Annals of Pure and Applied Logic*, vol. 96, no. 1–3, pp. 187–207, 1999.

[3] M. J. Fischer and R. E. Ladner, "Propositional dynamic logic of regular programs," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 194–211, 1979.

[4] D. Harel, A. Pnueli, and J. Stavi, "Propositional dynamic logic of non-regular programs," *Journal of*

Computer and System Sciences, vol. 26, no. 2, pp. 222–243, 1983.

[5] G. De Giacomo and M. Lenzerini, "PDL-based framework for reasoning about actions," in *AI*IA'95, Proceedings of the 4th Conference of the Italian Association for Artificial Intelligence*, vol. 992 of *Lecture Notes in Computer Science*, pp. 103–114, Springer-Verlag, 1995.

[6] D. Calvanese, G. De Giacomo, and M. Y. Vardi, "Reasoning about actions and planning in LTL action theories," in *KR'02, Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning* (D. Fensel, F. Giunchiglia, D. L. McGuinness, and M.-A. Williams, eds.), (Toulouse), pp. 593–602, Morgan Kaufmann, April 2002.

[7] B. J. Ross, "Probabilistic pattern matching and the evolution of stochastic regular expressions," *International Journal of Applied Intelligence*, vol. 13, pp. 285–300, Nov–Dec 2000.

[8] S. C. Kleene, "Representation of events in nerve nets and finite automata," in *Automata Studies* (C. E. Shannon and J. McCarthy, eds.), pp. 3–41, Princeton, New Jersey: Princeton University Press, 1956.

[9] J. T. Bradley, N. J. Dingle, S. T. Gilmore, and W. J. Knottenbelt, "Derivation of passage-time densities in PEPA models using ipc: the Imperial PEPA Compiler," in *MASCOTS'03, Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems* (G. Kotsis, ed.), (University of Central Florida), pp. 344–351, IEEE Computer Society Press, October 2003.

[10] J. Hillston, *A Compositional Approach to Performance Modelling*, vol. 12 of *Distinguished Dissertations in Computer Science*. Cambridge University Press, 1996. ISBN 0 521 57189 8.

[11] S. Gilmore and J. Hillston, "Feature interaction in PEPA," in *Process Algebra and Performance Modelling Workshop* (C. Priami, ed.), pp. 17–26, Università Degli Studi di Verona, Nice, September 1998.

[12] S. Gilmore, J. Hillston, and G. Clark, "Specifying performance measures for PEPA," in *Proceedings of the 5th International AMAST Workshop on Real-Time and Probabilistic Systems*, vol. 1601 of *Lecture Notes in Computer Science*, (Bamberg), pp. 211–227, Springer-Verlag, 1999.

[13] G. Clark, *Techniques for the Construction and Analysis of Algebraic Performance Models*. PhD thesis, Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, UK, 1994. CST–107–94.

[14] R. Milner, *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[15] M. Ribaudo, "Stochastic Petri net semantics for stochastic process algebras," in *PNPM'95, Proceedings of 6th Int. Workshop on Petri Nets and Performance Models*, (Durham, North Carolina), pp. 148–157, October 1995.

[16] W. J. Knottenbelt, "Generalised Markovian analysis of timed transitions systems," MSc thesis, University of Cape Town, South Africa, July 1996.

[17] J. T. Bradley, N. J. Dingle, P. G. Harrison, and W. J. Knottenbelt, "Distributed computation of passage time quantiles and transient state distributions in large semi-Markov models," in *Performance Modelling, Evaluation and Optimization of Parallel and Distributed Systems*, (Nice), IEEE Computer Society Press, April 2003.

[18] J. T. Bradley, N. J. Dingle, W. J. Knottenbelt, and P. G. Harrison, "Performance queries on semi-Markov stochastic Petri nets with an extended Continuous Stochastic Logic," in *PNPM'03, Proceedings of Petri Nets and Performance Models* (G. Ciardo and W. Sanders, eds.), (University of Illinois at Urbana-Champaign), pp. 62–71, IEEE Computer Society, September 2003.

[19] N. J. Dingle, P. G. Harrison, and W. J. Knottenbelt, "Response time densities in Generalised Stochastic Petri Net models," in *Proceedings of the 3rd International Workshop on Software and Performance (WOSP'2002)*, (Rome), pp. 46–54, July 2002.

[20] P. G. Harrison and W. J. Knottenbelt, "Passage-time distributions in large Markov chains," in *Proceedings of ACM SIGMETRICS 2002* (M. Martonosi and E. d. S. e Silva, eds.), pp. 77–85, Marina Del Rey, USA, June 2002.

[21] W. J. Knottenbelt, *Parallel Performance Analysis of Large Markov Models*. PhD thesis, Imperial College, London, United Kingdom, February 2000.

[22] W. J. Knottenbelt and P. G. Harrison, "Distributed disk-based solution techniques for large Markov models," in *NSMC'99, Proceedings of the 3rd Intl. Conference on the Numerical Solution of Markov Chains*, (Zaragoza), pp. 58–75, September 1999.

[23] W. Grassman, "Means and variances of time averages in Markovian environments," *European Journal of Operational Research*, vol. 31, no. 1, pp. 132–139, 1987.

[24] A. Reibman and K. Trivedi, "Numerical transient analysis of Markov models," *Computers and Operations Research*, vol. 15, no. 1, pp. 19–36, 1988.

[25] N. J. Dingle, W. J. Knottenbelt, and P. G. Harrison, "HYDRA: HYpergraph-based Distributed Response-time Analyser," in *PDPTA'03, Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications* (H. R. Arabnia and Y. Man, eds.), vol. 1, (Las Vegas, NV), pp. 215–219, June 2003.

[26] A. Carmona, L. Domingo, R. Macau, R. Puigjaner, and F. Rojo, "Performance experiences of the Barcelona Olympic games computer system," in *Computer Performance Evaluation, Modeling Techniques and Tools* (G. Haring and G. Kotsis, eds.), vol. 794 of *Lecture Notes in Computer Science*, pp. 52–75, Springer, 1994.