Functional Performance Specification with Stochastic Probes

Ashok Argent-Katwala and Jeremy T. Bradley

Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2BZ, United Kingdom
{ashok, jb}@doc.ic.ac.uk

Abstract. In this paper, we introduce FPS, a mechanism to define performance measures for stochastic process algebra models. FPS is a functional performance specification language which describes passage-time, transient, steady-state and continuous state space performance questions. We present a generalisation of stochastic probes, a formalism-independent specification of behaviour in stochastic process algebra models. Stochastic probes select the performance-critical paths for which the measures are required; increasing their expressiveness in turn gives us greater expressive power to represent performance questions. We end by demonstrating these tools on an RSS syndication architecture of up to 1.5×10^{51} states.

1 Introduction

In this paper, we introduce functional performance specification (FPS) over stochastic probes: a mechanism to define performance measures for stochastic process algebra models, with a unified description to capture passage-time, transient, steady-state and continuous state space quantities.

These four kinds of soft performance bounds are an integral part of 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 99% of the time; this would be derived from a passage-time quantile, based on an underlying stochastic model. 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 long-term probability that our software is in a particular failure mode is less than 0.002; this is a steady-state measure. Continuous state space analysis is used to quantify massively parallel agent-based architectures, by providing counts of agent states at particular time points, e.g. there are 5221 copies of a web client component in a queue for the web server at time 150 seconds.

When measuring the performance of a system, we see a need to separate the logic that specifies the performance query from the logic that defines the model; a modelling requirement described in [1]. Without such separation, it is

A. Horváth and M. Telek (Eds.): EPEW 2006, LNCS 4054, pp. 31–46, 2006.
© Springer-Verlag Berlin Heidelberg 2006

common to see many distinct versions of the same system being created explicitly to capture distinct measurement-centred behaviour. Stochastic probes are one method of making this separation of model and query. A stochastic probe [2] is a measurement device that defines arbitrary start and end events for a performance measure over a stochastic process algebra model.

We base the stochastic probe specification on an action-based regular expression syntax. We provide a further separation between the behavioural properties that make up our performance measure (as described by the stochastic probe) and the quantitative questions that we typically need to ask, using the functional performance specification framework.

This work builds on many performance specification methodologies: the NICE performance measurement system [3] of Woodside *et al.*; the regular expression style behavioural specification of asCSL [4] and TIPPtool [5]; the path-based reward structures described by Obal and Sanders [1]. FPS and stochastic probes are, however, unique in offering the combination of functional-style performance questions and a simple regular-expression based behavioural specification.

In this paper, we show how stochastic probes can be used to specify expressive behavioural constraints (Section 2.1), while the functional performance specification layer uses the stochastic probes to define passage-time, transient, steady-state or continuous state-based measures (Section 4). We significantly augment the expressiveness of the stochastic probe language from the introduction presented in [2] and present a new formal semantic translation of probe operators to underlying stochastic process algebra components (Section 6). We conclude by demonstrating the use of functional performance specification and stochastic probes over a PEPA model of the publish–subscribe mechanism, *Really Simple Syndication* or RSS (Section 7).

2 Stochastic Probes

We use a regular expression [6] specification to describe the start and end points of a performance measurement. The atoms of the regular expression are action names in the target system, drawn from the alphabet of the underlying process algebra model. This specification is turned into a fragment of stochastic process algebra, for our purposes PEPA [7], but we could equally apply probes to other stochastic process algebras such as EMPA [8] or SFSP [9] according to our modelling requirements. These probe fragments are composed with the original model to produce a model–probe system from which the performance measure can be easily extracted using tools native to the formalism.

The precise meaning of the probe will depend on the semantics of the underlying process algebra, in particular how choice works. In principle the translation offered in Section 6 will apply for any stochastic process algebra which has choice, a passive cooperation and supports CSP-style multiway synchronisation.

2.1 Stochastic Probe Definition

In this enhanced version of stochastic probes, we add the *without* operator, R/L. This specifies that, for a given path, a set of behaviours R should be observed without seeing any of the actions in the set, L. This is a significant generalisation over [10], where the modeller is only allowed to specify start and stop actions actions with no additional constraints on intermediate behaviour. It also generalises [2] where the modeller is only allowed to specify behaviour that should be seen along a particular path. A stochastic probe definition, R, has the following syntax:

$$\begin{aligned} R &::= A \mid T, T \mid S \\ S &::= T|S \mid T \\ T &::= R \mid R\{n\} \mid R\{m,n\} \mid R^{+} \mid R^{\star} \mid R? \mid R/act \\ A &::= act \mid act:\mathsf{start} \mid act:\mathsf{stop} \end{aligned} \tag{1}$$

act is an action label that matches a label in the system being measured. Any action specified in the probe has to be observed in the model before the probe can advance a state. An action, act, can also be distinguished or tagged 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** operation. R_1 is matched against the system's operation, then R_2 is matched.

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

$R^{\star}$ is the **closure** operation, where zero or more copies of R are matched against the system.

$R?$ is the **optional** operation, matching zero or one copy of R against the system.

$R\{n\}$ is the **iterative** operation. A fixed number of 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** operation. Between m and n copies of R are matched against the system's operation. $R\{m,n\}$ is equivalent to $R\{n,m\}$, and we consider the canonical form to have the smaller index first.

R^{+} is the **positive closure** operation, where one or more copies of R are matched against the system. It is syntactic sugar for $R, R^{\star}$.

R/act is the **without** operation. R must begin again whenever the probe sees an act action that is not matched by R.

3 PEPA Stochastic Process Algebra

PEPA is used as the stochastic process algebra of choice for defining the probe semantics of Section 6 and the modelling example of Section 7. PEPA is a parsimonious stochastic process algebra that can describe compositional stochastic

models and has been used for many performance modelling case studies. These models consist of components whose actions incorporate random exponential delays. Full details of the PEPA process algebra can be found in [7]. In brief, the syntax of a PEPA component, P, is represented by:

$$P ::= (a, \lambda).P \;\Big|\; P + P \;\Big|\; P \underset{S}{\bowtie} P \;\Big|\; P/L \;\Big|\; A$$

$(a, \lambda).P$ is an action 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 a $\top$-value, which makes the action passive in a cooperation (see below).

$P_1 + P_2$ is a choice operation between two components. 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 between two components which synchronise over a set of actions, S. 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 of a set of actions, L. 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.

Regarding related performance specification in PEPA, itself, Gilmore and Hillston [11] have developed their own *feature interaction* logic which explores a PEPA model, assigning rewards 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, it 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] for steady-state measure specification.

4 FPS: Functional Performance Specification

A functional performance specification is presented here as a contrast to well-established logical performance specification formalisms that have some from CSL [13]. Logical formalisms reduce all performance questions to a truth value, for instance in a later version of CSL [14], the expression $s \models \mathcal{S}_{<0.3}(\psi)$ means:

> *Is the state s the initial state of a path that ends in the set of states defined by ψ where the total steady-state probability of being in those ψ states is less than* 0.3*?*

This is how a performance modeller might phrase the same question:

> *Is the steady-state probability of the states defined by ψ less than* 0.3*?*

If we were to require the precise value of the steady-state probability in CSL, we would have to ask the more general question $s \models \mathcal{S}_{<p}(\psi)$ and observe the value of p at which the formula moves from being true to false. Of late, this situation has been in part remedied by the support of tools such as PRISM [15], which allow questions such as:

> *Find p such that, given a start state s, $s \models \mathcal{S}_{<p}(\psi)$ is true.*

However, we still feel that the question is not as directly or succinctly stated as it might be. Despite this, logical performance specification offers a very expressive and very powerful environment, especially in being able to construct compositional performance queries, due in a large part to its well-explored CTL pedigree. In developing a functional performance paradigm, we seek to be able to ask the quantitative performance question more directly, as in:

> *What is the steady-state probability of being in a set of states, ψ?*

while maintaining the compositional power of logical performance specification. However in this paper, since we favour a process framework for our underlying model, we use stochastic probes rather than logical atomic propositions of CSL to specify our state sets.

4.1 Performance Specification Syntax

With this motivation, we present a functional specification, which takes an input native to a stochastic process algebra model – i.e. a stochastic probe or component label – and generates a performance function, e.g. a passage time CDF, which can itself be sampled or composed into higher-order performance queries. A functional performance specification, $\mathcal{M}$, over a stochastic probe, R, has the following grammar:

$$\mathcal{M} \quad ::= \quad \mathcal{S}teady(R) \;\big|\; \mathcal{P}assage(R) \;\big|\; \mathcal{T}ransient(R) \;\big|\; \mathcal{N}umber(C)$$

and $\mathcal{S}teady(R)$ represents a steady-state measure; $\mathcal{P}assage(R)$ represents a passage-time cumulative distribution function; $\mathcal{T}ransient(R)$ represents a transient-state distribution function; and $\mathcal{N}umber(R)$ represents a deterministic component counting function. C is a component type from the process model.

4.2 Definitions

Let the joint probe–model system, $R \underset{L}{\bowtie} M$, be a Markov process, $\{Z(t) : t \geq 0\}$, where $Z(t)$ is the state of the system at time t. We can define the counting process, $N(t) = |\{\!|Z(u) : 0 \leq u \leq t|\!\}| - 1$, to be the number of state transitions that have occurred by time t.

In order to define the performance measure operators, $\mathcal{P}assage$ and $\mathcal{T}ransient$, we will need to specify sets of source states, F, and target states, G, based on the instants after probe start and probe stop actions have occurred respectively. So we define:

$$F(R) = \{R'' \underset{L}{\bowtie} M'' : R' \underset{L}{\bowtie} M' \xrightarrow{(a:\mathsf{start},\cdot)} R'' \underset{L}{\bowtie} M''\} \tag{2}$$

$$G(R) = \{R'' \underset{L}{\bowtie} M'' : R' \underset{L}{\bowtie} M' \xrightarrow{(a:\mathsf{stop},\cdot)} R'' \underset{L}{\bowtie} M''\} \tag{3}$$

where R' and R'' are derivative or successor states of the probe, R. M represents the model being measured and M' and M'' are derivative states of M. It is worth noting, that although we have used PEPA notation to highlight the joint probe–model process, these definitions could easily be expressed in other process algebras.

In the following descriptions $\mathsf{prob} \equiv [0, 1]$, the set of probability values and C is a component type from the process model.

Steady-state, $\mathcal{S}teady(R) : \mathsf{prob}$. Applying the steady-state operator to a probe, R, generates the steady-state probability for being in one of the states reachable by a probe stop action. For irreducible state spaces, this can be expressed as:

$$\mathcal{S}teady(R) = \sum_{x \in G(R)} \pi(x) \tag{4}$$

where $\pi(x)$ is the steady-state probability of being in the state x in the process $Z(t)$.

Passage-time CDF, $\mathcal{P}assage(R) : \mathbb{R}^+ \to \mathsf{prob}$. A passage-time measure over a probe R returns a cumulative distribution function for the passage-time that starts from a state reachable by a probe start action and finishes at a state reachable by a probe stop action. More precisely, using λ-notation to define the cumulative distribution function, we can say:

$$\mathcal{P}assage(R) = \lambda t \;.\; \sum_{x \in F(R)} \pi(x)\, \mathbb{P}(P_{xG(R)} \leq t) \tag{5}$$

where $P_{i\mathcal{J}}$ is the random variable representing the passage-time starting from a state i and terminating in one of the states in $\mathcal{J}$, as given by:

$$P_{i\mathcal{J}} = \inf\{u > 0 : Z(u) \in \mathcal{J}, N(u) > 0, Z(0) = i\} \tag{6}$$

In Eq. (5), we weight the passage with the steady-state probabilities of starting in any of the start states, as defined by the start actions in the probe. It

is our intention to generalise this in future versions, so that we can specify a time-point from which we can generate a transient distribution to weight the passage with.

Clearly, the associated density function, $f(t)$, for this passage-time measurement can be obtained by differentiating the CDF, $f(t) = \mathcal{Passage}(R)'(t)$.

Transient-state function, $\mathcal{Transient}(R) : \mathbb{R}^+ \rightarrow \mathsf{prob}$. A transient-state measure over a probe R returns the transient-state distribution function for having just completed the probe stop action at time t, having completed a probe start action at time, $t = 0$. It is defined as follows, again using the steady-state vector to weight the multiple start states that might arise from the stochastic probe:

$$\mathcal{Transient}(R) = \lambda t \; . \sum_{x \in F(R)} \pi(x) \, \mathbb{P}(Z(t) \in G(R) \mid Z(0) = x) \tag{7}$$

Component count function, $\mathcal{Number}(C) : \mathbb{R}^+ \rightarrow \mathbb{R}^+$. Applying the component count function to a component C in the model yields a function which counts the number of that components in the system in the state C at time t. It relates to the recent innovations in continuous state space approximation of stochastic process algebra models [16], which solve systems of coupled of ODEs for systems with huge and otherwise computationally infeasible state spaces.

The model M is regarded as consisting of a cooperation of n classes of component, C_i, $1 \leq i \leq n$ and with each component class having m_i derivative states. At any time, there may be many components of the same class, but in a different state in the system. We let $v_{ij}(t)$ represent the number of components of type C_i in state j at time t for $1 \leq j \leq m_i$. This is found by solving a set of coupled ODEs of the form $v'_{ij}(t) = g(v_{11}(t), \ldots, v_{nm_n}(t))$. In effect:

$$\mathcal{Number}(C_{ij}) = \lambda t \; . \; v_{ij}(t) \tag{8}$$

5 Stochastic Probe Examples

We give a few examples of stochastic probes as specified by regular expressions over a simple PEPA model. Consider a Bartender and a few customers, specified in PEPA:

$$\begin{aligned}
\text{Bartender} &\stackrel{\text{def}}{=} (\mathit{serve}, r_s).\text{Bartender} + (\mathit{restock}, \mathrm{r_r}).\text{Bartender} \\
\text{Person} &\stackrel{\text{def}}{=} (\mathit{life}, r).\text{Person} + (\mathit{thirst}, \mathrm{s}).\text{Thirsty} \\
\text{Thirsty} &\stackrel{\text{def}}{=} (\mathit{serve}, \top).\text{Drinking} \\
\text{Drinking} &\stackrel{\text{def}}{=} (\mathit{drink}, r_0).(\mathit{resume}, r_1).\text{Person} + (\mathit{drink}, \mathrm{r_0}).\text{Thirsty} \\
\text{Sys} &\stackrel{\text{def}}{=} \text{Bartender} \underset{\{serve\}}{\bowtie} (\text{Person} \mid\mid \text{Person} \mid\mid \text{Person})
\end{aligned}$$

Let us ask a few simple questions of this model:

- How long between the first drink is served and the tenth?

$$serve\text{:}\mathsf{start}, serve\{8\}, serve\text{:}\mathsf{stop}$$

- Measure the time from the tenth serving till any of the drinkers returns to their normal life.

$$serve\{9\}, serve\text{:}\mathsf{start}, resume\text{:}\mathsf{stop}$$

- If the bar holds stocks for 100 drinks, and is restocked to back to 100 drinks every time the Bartender performs *restock* action, how long till the bar runs dry?

$$serve\text{:}\mathsf{start}, serve\{99\}/restock, serve\text{:}\mathsf{stop}$$

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. A probe that does not use the *without* construct asks the question "will I ever see this behaviour?". Using *without*, a modeller may also ask "will I see exactly this behaviour next?", if not then skip back to a particular point in the measure.

6 Stochastic Probe Translation

Fig. 1 depicts the conversion of the individual regular expression elements to process algebra components pictorially. The without operator acts at a different level to the other operators, and is concerned with the actions within a particular sub-expression, and not with composing expressions together. Dotted arrows denote that there is an immediate choice (with no prefix action) to continue in the successor state.

Note that the representation for $R\{m, n\}$ is not the same as $(R\{m\}|R\{m + 1\}|\ldots|R\{n\})$. That would be one way to translate it, but would mean committing, through random choice to matching a particular number of repetitions of R. This could be surprising to a modeller who has asked the probe to match any of the range of repetitions. Instead we treat $R\{m, n\}$ as $R\{m\}, R?\{m - n\}$ (or $R?\{n\}$ where $m = 0$).

Before translating the probe at all, we first build a parse tree, during which all the syntactic sugar is removed. We convert: R^+ to $R, R^\star$; $R\{n\}$ to $R, \ldots, R$ to give n explicit copies of R; and $R\{m, n\}$ to $R\{m\}, R?\{m-n\}$ or $R?\{n\}$ where $m = 0$. Now, we are left with probes which fit a smaller syntactic form, which we need to convert to our target process algebra. In particular, the syntax of the T component of the regular expression definition from Eq. (1) is reduced to:

$$T \quad ::= \quad R \;\big|\; R^\star \;\big|\; R? \;\big|\; R/act$$

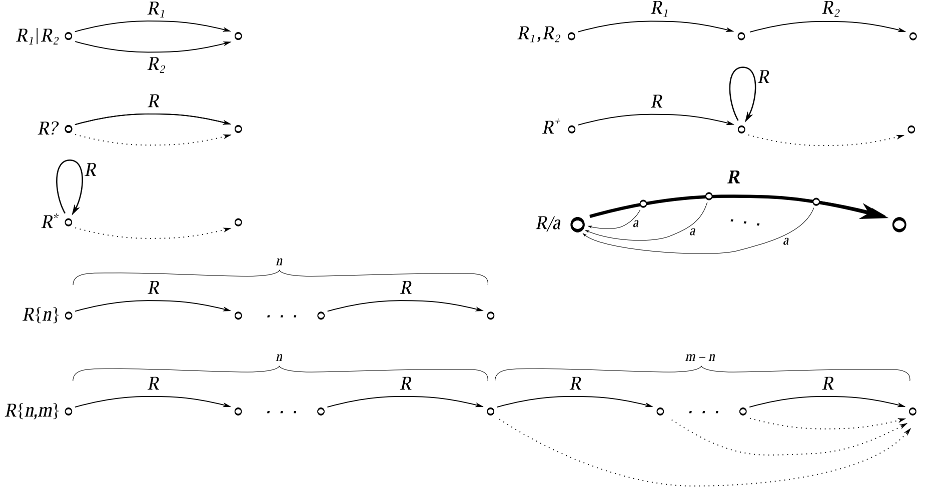

Fig. 1. The representation of the distinct regular expression terms as state-transitions in the underlying process algebra

6.1 Mapping Probes to PEPA

In the definitions below, we take as input the component name, P, we are to define, the component name, Q, we are to end at and a reset-list of action names and the component-label to which we return when we absorb that action. The reset-list is initially empty. We introduce new, intermediate component labels N_i, as required.

Throughout the conversion, only the topmost operator in the tree is considered, and the subtrees are handled recursively. First, a few definitions:

Definition 1. $\mathcal{F}(R)$ *denotes the first names of the probe expression R. These are the action names that are explicitly used at the beginning of R. It is defined over the terms for regular expressions.*

The sequence operator needs careful handling; where the first term is optional ($R^\star$ or $R?$ or $R\{0, n\}$) then the first actions of the second term are also immediately available. We take the first (which is also the most specific) definition below that matches the current situation:

$$
\begin{aligned}
\mathcal{F}(a\text{:}\mathsf{start}) &= a \\
\mathcal{F}(a\text{:}\mathsf{stop}) &= a \\
\mathcal{F}(a) &= a \\
\mathcal{F}(R_1^\star, R_2) &= \mathcal{F}(R_1) \cup \mathcal{F}(R_2) \\
\mathcal{F}(R_1?, R_2) &= \mathcal{F}(R_1) \cup \mathcal{F}(R_2) \\
\mathcal{F}(R_1, R_2) &= \mathcal{F}(R_1)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}(R_1 \mid R_2) &= \mathcal{F}(R_1) \cup \mathcal{F}(R_2) \\
\mathcal{F}(R?) &= \mathcal{F}(R) \\
\mathcal{F}(R^+) &= \mathcal{F}(R) \\
\mathcal{F}(R^\star) &= \mathcal{F}(R) \\
\mathcal{F}(R/a) &= \mathcal{F}(R)
\end{aligned}
$$

Definition 2. $\mathcal{N}(R)$ *denotes the names that are not explicitly enabled at the beginning of R. For a probe that acts over the alphabet A,* $\mathcal{N}(R) = A - \mathcal{F}(R)$.

At every state, the probe will offer every action to lead somewhere. If not to move the probe forward, or to reset it to some prior state due to an enclosing *without* operator, then the probe absorbs the action and remains in the same state. To abstract this from the translation that follows, we define the function $\mathcal{S}$, which provides the sum (choice) of all the reset and self-loops for an expression R, being translated running from label P, with the set of pairs for resets X. Note that this procedure adds a transition for every action in $\mathcal{N}(R)$:

$$\mathcal{S}(R,P,X) \quad = \sum_{(b,E)\in X:\ b\in\mathcal{N}(R)} (b,\top).\mathrm{E} \quad + \sum_{c\in\mathcal{N}(R):\ \nexists x:\ (c,x)\in X} (c,\top).\mathrm{P}$$

We now define the full translation, $\mathcal{T}$, which produces a set of PEPA definitions, and is a formal version of the English descriptions above. The first argument to $\mathcal{T}$ is always a probe expression, which is underlined, to avoid confusing the expression's sequence operator for the argument separator. The N_i are new process names for each recursive call.

The initial call to translate a probe is to $\mathcal{T}(\underline{R}, Probe, Probe, \emptyset)$. This creates a cyclic PEPA component, Probe, so when composed with an irreducible system, that may be preserved.

$$\begin{array}{lll}
\text{Action} & \mathcal{T}(\underline{a},P,Q,X) & = \mathrm{P} \stackrel{\text{def}}{=} (a,\top).\mathrm{Q} + \mathcal{S}(\underline{\mathrm{a}},\mathrm{P},\mathrm{X}) \\
\text{Grouping} & \mathcal{T}(\underline{(R)},P,Q,X) & = \mathcal{T}(\underline{R},P,Q,X) \\
\text{Choice} & \mathcal{T}(\underline{R_1|R_2},P,Q,X) & = \mathrm{P} \stackrel{\text{def}}{=} \mathrm{N}_1 + \mathrm{N}_2 + \mathcal{S}(\underline{\mathrm{R}_1|\mathrm{R}_2},\mathrm{P},\mathrm{X}); \\
 & & \quad \mathcal{T}(\underline{R_1},N_1,Q,X);\ \mathcal{T}(\underline{R_2},N_2,Q,X) \\
\text{Sequence} & \mathcal{T}(\underline{R_1,R_2},P,Q,X) & = \mathcal{T}(\underline{R_1},P,N_1,X);\ \mathcal{T}(\underline{R_2},N_1,Q,X) \\
\text{Closure} & \mathcal{T}(\underline{R^\star},P,Q,X) & = \mathrm{P} \stackrel{\text{def}}{=} \mathrm{N}_1 + \mathrm{Q} \\
 & & \quad + \mathcal{S}(\underline{R^\star},P,X);\ \mathcal{T}(\underline{R},N_1,P,X) \\
\text{Optional} & \mathcal{T}(\underline{R?},P,Q,X) & = \mathrm{P} \stackrel{\text{def}}{=} \mathrm{N}_1 + \mathrm{Q} \\
 & & \quad + \mathcal{S}(\underline{R?},P,X);\ \mathcal{T}(\underline{R},N_1,Q,X) \\
\text{Without} & \mathcal{T}(\underline{R/a},P,Q,X) & = \mathcal{T}(\underline{R},P,Q,X') \\
 & & \text{where } X'=\{(c,x)\in X \mid c\neq a\}\cup(a,P)
\end{array}$$

Note that the intention of the ";" operator here is as a separator between definitions. $\mathcal{T}(\ldots);\mathcal{T}(\ldots)$ means the PEPA system contains all the definitions from both calls.

Or in words (omitting the passive loops and resets):

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

$$\mathrm{P} \stackrel{\text{def}}{=} (a,\top).\mathrm{Q}$$

Choice $R_1 \mid R_2$**:** translates to $\mathrm{P} \stackrel{\text{def}}{=} \mathrm{N}_1 + \mathrm{N}_2$ 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.

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

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

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

Without R/a**:** R is translated running from P to Q and at every stage between, offers a choice of $(a, \top).P$ wherever R does not explicitly offer one. We do this by adding the pair (a,P) to the reset-list and removing any other a-reset pair. This ensures we use the most specific reset action that the modeller has chosen, should they choose to exclude the same action more than once.

The procedure above gives us a valid PEPA fragment which will behave properly as a probe. However, for our analysis with ipc/DNAmaca [10] 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 create an observationally equivalent probe which has a partition in its state space to enable ipc to specify the start and stop states for DNAmaca. The details of this procedure can be found in Argent-Katwala *et al.* [2].

7 Worked Example: An RSS Publish–Subscribe System

To demonstrate our functional performance specification framework, we introduce a simplified PEPA model of an RSS publish–subscribe system.

The RSS system under consideration consists of N_C RSS clients and a single RSS server comprising N_S virtual servers running in parallel. The clients and server are connected by a network capable of sustaining N_N concurrent network connections. This is described by the top-level system equation below:

$$\begin{aligned} \mathit{RSS_System} \stackrel{\mathrm{def}}{=} (\mathit{RSS_Client}[N_C] \underset{L}{\bowtie} \mathit{RSS_Server}[N_S, M]) \\ \underset{L}{\bowtie} \mathit{RSS_Network}[N_N] \end{aligned}$$

where:

$$\begin{aligned} L &= \{\mathit{subscribe}, \mathit{unsubscribe}, \mathit{rss_poll}, \mathit{rss_update}, \mathit{rss_cache_hit}\} \\ M &= \{\mathit{new_feed}, \mathit{rss_refresh}\} \end{aligned}$$

describes the set of actions that the RSS client and server cooperate over via the network. Note that in the above description, the $A[N]$ construction is shorthand for $A[N, \emptyset]$ and $A[N, M]$ represents N components of type A cooperating over the set of actions M, as in:

$$A[N, M] \equiv \underbrace{A \underset{M}{\bowtie} A \underset{M}{\bowtie} \cdots \underset{M}{\bowtie} A}_{N}$$

The RSS client can subscribe to a server feed, after which it can poll the RSS server for the current feed information. With some rate, λ_2, a client can withdraw from the system by unsubscribing. After polling, the client is given

a newer version of the RSS feed or told that the cached version that the client has is current and no update is necessary (achieved through a choice between *rss_update* and *rss_cache_hit* actions in component RSS_Client_1). At this stage, the client will poll at different rates according to whether it has just been handed a new version of the feed or not. If an older cached version exists, it will poll more frequently, with $\lambda_4 > \lambda_3$.

$$\begin{aligned}
RSS_Client &\stackrel{\text{def}}{=} (subscribe, \lambda_1).RSS_Client_n \\
RSS_Client_n &\stackrel{\text{def}}{=} (rss_poll, \lambda_3).RSS_Client_1 + (unsubscribe, \lambda_2).RSS_Client \\
RSS_Client_c &\stackrel{\text{def}}{=} (rss_poll, \lambda_4).RSS_Client_1 + (unsubscribe, \lambda_2).RSS_Client \\
RSS_Client_1 &\stackrel{\text{def}}{=} (rss_update, \top).RSS_Client_n + (rss_cache_hit, \top).RSS_Client_c \\
&\quad + (unsubscribe, \lambda_2).RSS_Client
\end{aligned}$$

The RSS virtual servers, of which there will be several working in parallel to update the feed information, keep track of the subscription list (not explicitly represented in this model). A current feed has a lifetime determined by the *new_feed* action at rate μ_1, which, together with the *rss_refresh* action, represent a feed content change on a shared disk, say. A client polling one of the servers receives either an *rss_update* or an *rss_cache_hit* with probabilities $\frac{\mu_u}{\mu_u+\mu_c}$ and $\frac{\mu_c}{\mu_u+\mu_c}$ respectively, where $\mu_c > \mu_u$ representing that sending a message that the feed has not modified is quicker than sending the whole body of the feed. This represents a type of HTTP conditional request, without greatly increasing the size of the model.

$$\begin{aligned}
RSS_Server &\stackrel{\text{def}}{=} (subscribe, \top).RSS_Server + (unsubscribe, \top).RSS_Server \\
&\quad + (rss_poll, \top).RSS_Server_2 + (new_feed, \mu_1).RSS_Server_1 \\
RSS_Server_1 &\stackrel{\text{def}}{=} (rss_refresh, \mu_2).RSS_Server \\
RSS_Server_2 &\stackrel{\text{def}}{=} (rss_update, \mu_u).RSS_Server + (rss_cache_hit, \mu_c).RSS_Server
\end{aligned}$$

Finally, a simple network model keeps track of limited bandwidth by capping the total number of connections within a given network window to N_N. The duration of the network window is governed by the *net_recover* action and the γ parameter.

$$\begin{aligned}
RSS_Network &\stackrel{\text{def}}{=} (subscribe, \top).RSS_Network_1 \\
&\quad + (unsubscribe.\top).RSS_Network_1 \\
&\quad + (rss_poll, \top).RSS_Network_1 \\
&\quad + (rss_update, \top).RSS_Network_1 \\
&\quad + (rss_cache_hit, \top).RSS_Network_1 \\
RSS_Network_1 &\stackrel{\text{def}}{=} (net_recover, \gamma).RSS_Network
\end{aligned}$$

7.1 Functional Performance Queries

We present some questions and analysis of the RSS model in the functional performance specification style:

Steady-state query. What is the steady-state probability of seeing 4 consecutive *rss_poll* actions in the RSS model without having an *rss_update* action? This is translates into the formal functional performance query:

$$\mathcal{S}teady(\,(rss_poll{:}\mathsf{start}, rss_poll\{2\}, rss_poll{:}\mathsf{stop})/rss_update) \tag{9}$$

We solved this query for a 1,494,288 state system of 7 clients, 3 servers and 2 network connections. $\mathcal{S}(R_1) = 0.19273$, for R_1 taken to be the probe of Eq. (9).

Passage-time query. What is the probability that the time between consecutive *rss_update* actions is between 2 and 4 time units? This translates into the formal functional performance query on the cumulative distribution function of the equivalent passage time, $\mathcal{P}assage(R_2)(4) - \mathcal{P}assage(R_2)(2)$ where $R_2 = rss_update{:}\mathsf{start}, rss_update{:}\mathsf{stop}$.

We solved this for a 11,232 state system of 4 clients, 2 servers and 2 network connections. Fig. 2 shows a probability density function of the appropriate passage $f_p(t) = \mathcal{P}assage(R_2)'(t)$. Fig. 3 shows the cumulative distribution function for the appropriate passage $F_P(t) = \mathcal{P}assage(R_2)(t)$. From this second plot we can calculate the required probability $F_P(4) - F_P(2) = 0.1072793$.

Component counting. How many *RSS_Client* components are there at time 60, for a system with $N_C = 100$, $N_S = 3$, $N_N = 2$? This translates into the formal functional performance query on the component counting function $\mathcal{N}umber(RSS_Client)(60)$. Although this query is not based on a stochastic probe, it is of fundamental interest as a quantitative measure to a performance modeller.

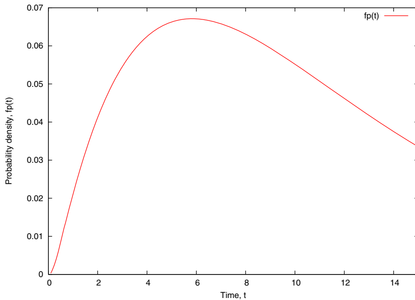

Fig. 2. The PDF for the passage time, $f_p(t) = \mathcal{P}assage(R_2)'(t)$, with $N_C = 4, N_S = 2, N_N = 2$

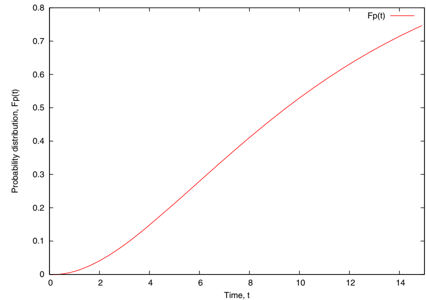

Fig. 3. The CDF for the passage time, $F_P(t) = \mathcal{P}assage(R_2)(t)$, with $N_C = 4, N_S = 2, N_N = 2$

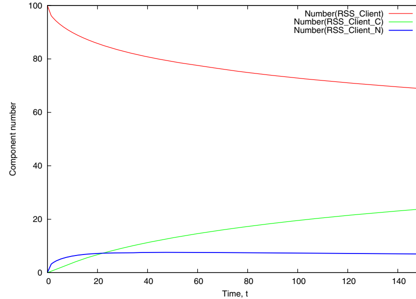

Fig. 4. The component counting functions, (top) $\mathcal{N}umber(RSS_Client)(t)$, (middle) $\mathcal{N}umber(RSS_Client_C)(t)$ and (bottom) $\mathcal{N}umber(RSS_Client_N)(t)$, for RSS model $N_C = 100, N_S = 3, N_N = 2$

For parameters $N_C = 100$, $N_S = 3$, $N_N = 2$, the RSS model has approximately 1.5×10^{51} states. For this magnitude of calculation, the only practical option is to resort to the continuous state space techniques of Hillston [16]. The result of $\mathcal{N}umber(RSS_Client)(60) = 77.4$ is derived from the appropriate counting function in Fig. 4

8 Conclusion

In this paper, we have developed FPS, a functional performance specification language which allows the modeller to derive quantitative performance functions using stochastic probes. We have also extended stochastic probes as a means of measuring soft performance characteristics of systems. We demonstrated a regular expression language which specifies the stochastic probe and is itself converted into a stochastic process algebra component. The probe is composed with the target system for the purposes of extracting the performance measurement.

We showed how this joint FPS/stochastic probe environment could be used to specify quantitative performance and reliability bounds on stochastic process algebra based systems. Finally, we applied these techniques to a model of an RSS system where we analysed PEPA models of 11 thousand, 1.5 million and 1.5×10^{51} states in size for quantitative performance measures.

Future improvements include allowing the specification of initial state distributions for passage-time and transient measures. We would also like to find an intuitive but unrestrictive way of using probes to define the continuous state space measure, in addition to or maybe as an alternative to using the SPA component type.

Acknowledgements

The authors would like to thank anonymous referees for their insightful comments and suggestions. AAK is supported by EPSRC under the PerformDB

grant EP/D054087/1. JB is supported in part by EPSRC under the GRAIL grant EP/D505933/1 and the PerformDB grant EP/D054087/1.

References

1. W. D. Obal and W. H. Sanders, "State-space support for path-based reward variables," *Performance Evaluation*, vol. 35, pp. 233–251, May 1999.
2. A. Argent-Katwala, J. T. Bradley, and N. J. Dingle, "Expressing performance requirements using regular expressions to specify stochastic probes over process algebra models," in *WOSP 2004, Proceedings of the 4th International Workshop on Software and Performance* (V. Almeida and D. Lea, eds.), (Redwood City, California), pp. 49–58, ACM, January 2004.
3. C. M. Woodside and C. Shramm, "Complex performance measurements with NICE (notation for interval combinations and events)," *Software—Practice and Experience*, vol. 24, pp. 1121–1144, December 1994.
4. C. Baier, L. Cloth, B. R. Haverkort, M. Kuntz, and M. Siegle, "Model checking action- and state-labelled Markov chains," *DSN'04, Proceedings of International Conference on Dependable Systems and Networks*, pp. 701–710, June 2004.
5. H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle, "Compositional performance modelling with the TIPPtool," *Performance Evaluation*, vol. 39, no. 1–4, pp. 5–35, 2000.
6. 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.
7. J. Hillston, *A Compositional Approach to Performance Modelling*, vol. 12 of *Distinguished Dissertations in Computer Science*. Cambridge University Press, 1996.
8. M. Bernardo and R. Gorrieri, "Extended Markovian Process Algebra," in *CONCUR'96, Proceedings of the 7th International Conference on Concurrency Theory* (U. Montanari and V. Sassone, eds.), vol. 1119 of *Lecture Notes in Computer Science*, pp. 315–330, Springer-Verlag, Pisa, August 1996.
9. T. Ayles, A. J. Field, and J. N. Magee, "Adding performance evaluation to the LTSA tool," in *Proceedings of 13th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, 2003.
10. 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.
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. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton, "Verifying continuous-time Markov chains," in *Computer-Aided Verification*, vol. 1102 of *Lecture Notes in Computer Science*, pp. 269–276, Springer-Verlag, 1996.

14. C. Baier, J.-P. Katoen, and H. Hermanns, “Approximate symbolic model checking of continuous-time Markov chains,” in *CONCUR’99, Proceedings of the 10th International Conference on Concurrency Theory*, vol. 1664 of *Lecture Notes in Computer Science*, pp. 146–162, Springer-Verlag, 1999.
15. M. Kwiatkowska, G. Norman, and D. Parker, “PRISM: Probabilistic symbolic model checker,” in *TOOLS’02, Proceedings of the 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (A. J. Field et al., ed.), vol. 2324 of *Lecture Notes in Computer Science*, (London), pp. 200–204, Springer-Verlag, 2002.
16. J. Hillston, “Fluid flow approximation of PEPA models,” in *QEST’05, Proceedings of the 2nd International Conference on Quantitative Evaluation of Systems*, (Torino), pp. 33–42, IEEE Computer Society Press, September 2005.